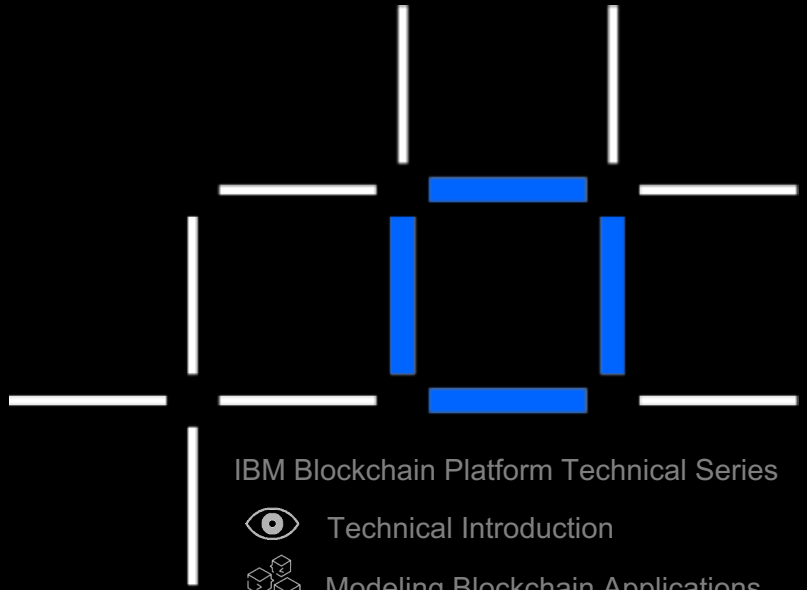
Under the Covers

A Technical Deep-Dive on Hyperledger Fabric and IBM Blockchain Platform

Jeff Tennenbaum
Tennenjl@us.ibm.com
@JeffTennenbaum1



IBM Blockchain Platform Technical Series

- Technical Introduction
- Modeling Blockchain Applications
- **Under the Covers**
- Architectural Good Practices
- What's New in Technology

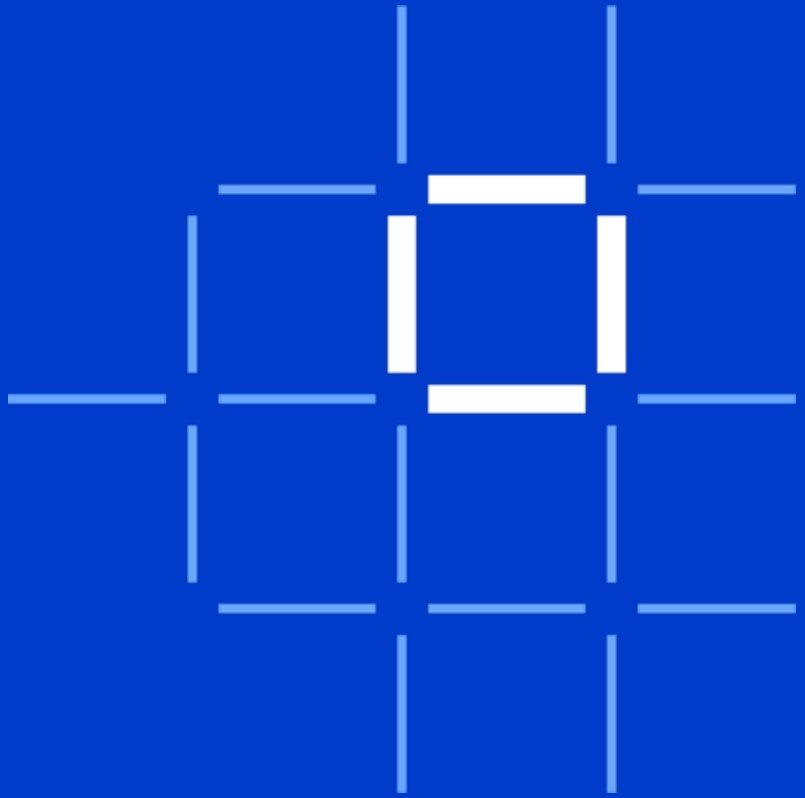
V4.08, 1 Jan 2020

IBM **Blockchain**

IBM

Project Status and Roadmap

Hyperledger Fabric Technical Dive

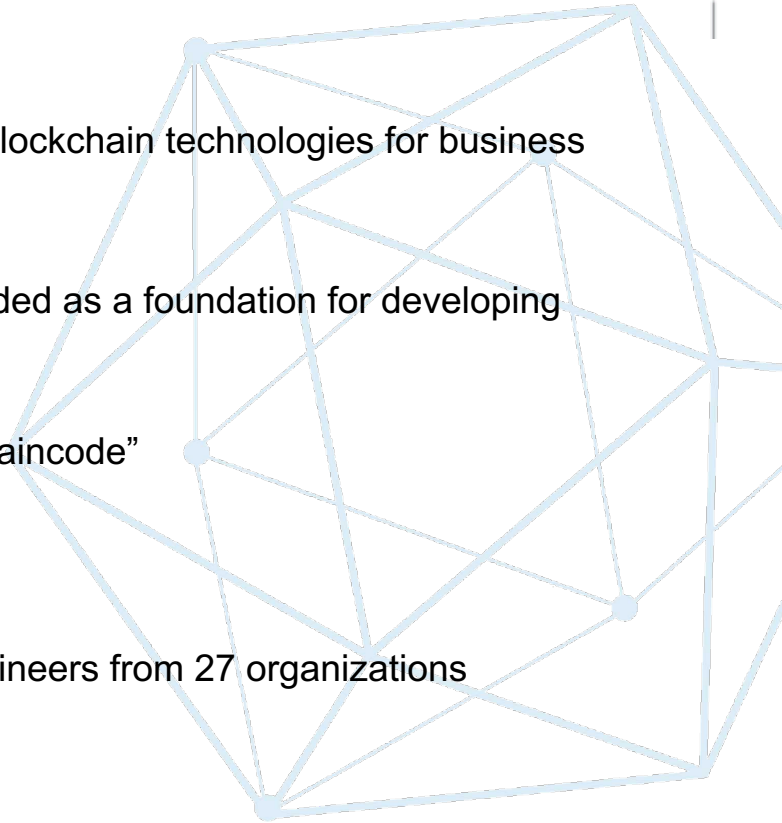


Blockchain concepts

- Consider the business concepts that comprise blockchain solutions:
  - **Assets** modeled as data structures
  - **Contracts** modeled as algorithms
  - **Transactions** modeled as invocations of algorithms
  - **Business Networks** modeled as peer-to-peer networks
  - **Participants** modeled as nodes on these networks

- IBM Blockchain Platform is based around **Hyperledger Fabric**
  - We will now look at how Hyperledger Fabric exposes these business concepts to the blockchain developer
  - We will also look at the IBM Blockchain Platform tools that help developers implement these concepts in a solution

What is Hyperledger Fabric



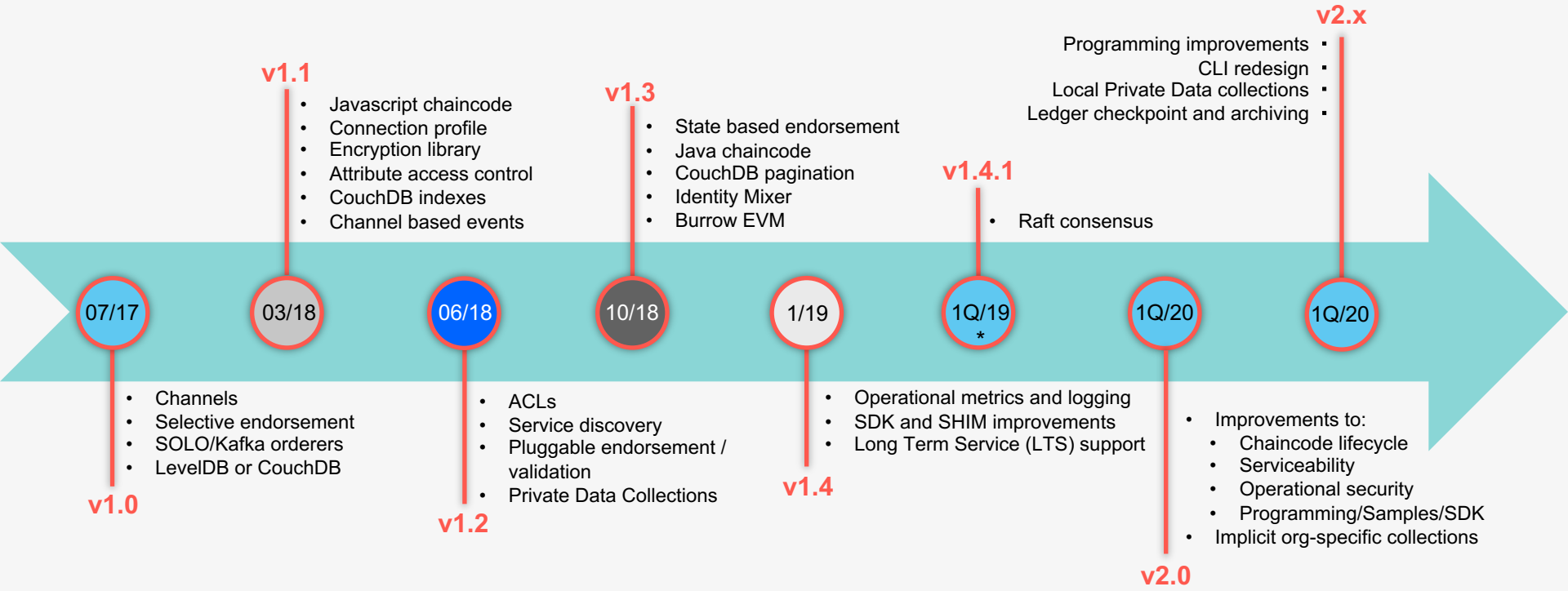
- Linux Foundation Hyperledger
  - A collaborative effort created to advance cross-industry blockchain technologies for business

- Hyperledger Fabric
  - An implementation of blockchain technology that is intended as a foundation for developing blockchain applications
  - Key technical features:
  - A shared ledger and smart contracts implemented as “chaincode”
  - Privacy and permissioning through membership services
  - Modular architecture and flexible hosting options

- First major version released July 2017: contributions by 159 engineers from 27 organizations
  - IBM is one of the contributors to Hyperledger Fabric

HYPERLEDGER FABRIC

Roadmap



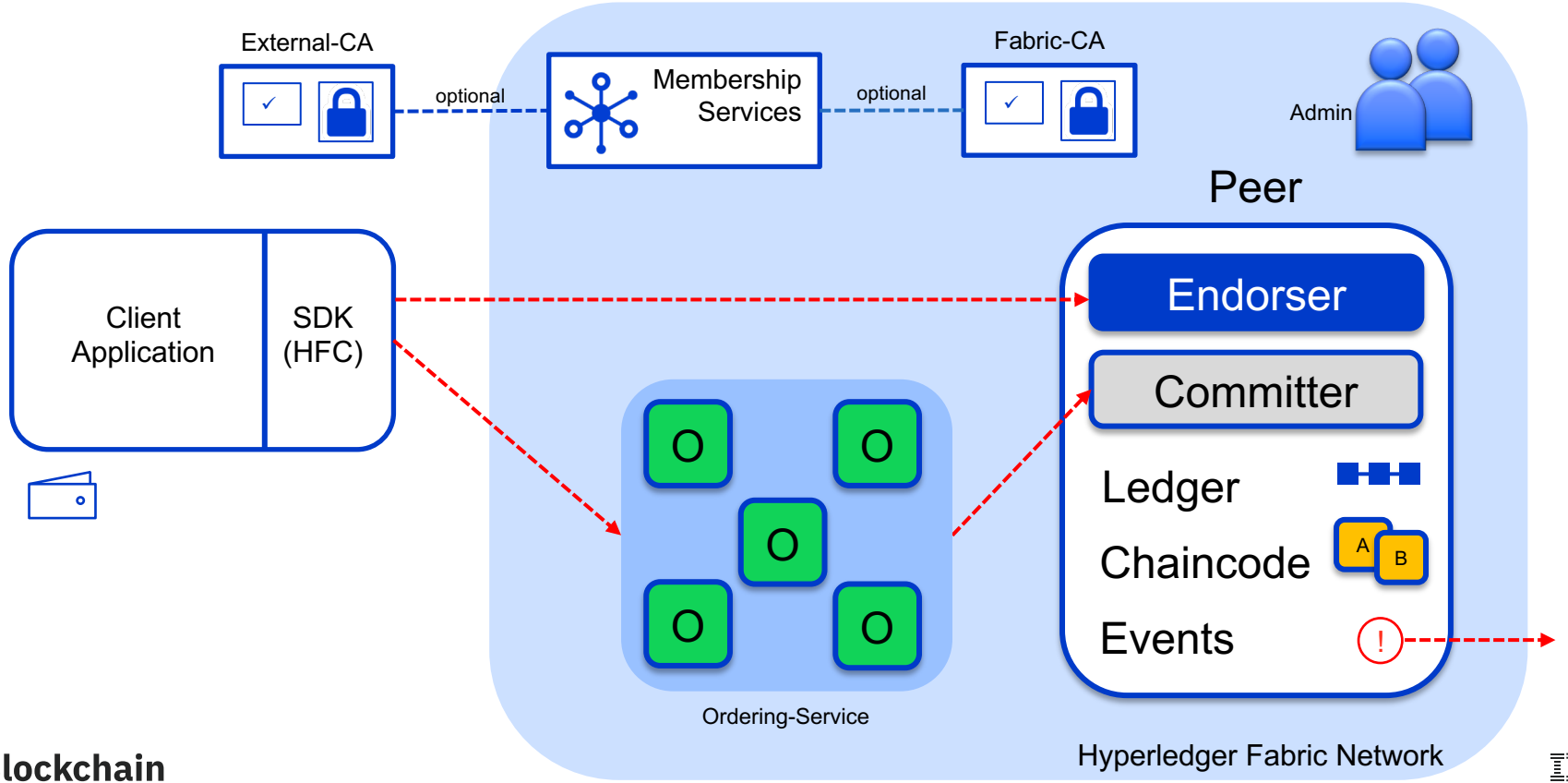
HYPERLEDGER
BLOCKCHAIN TECHNOLOGIES FOR BUSINESS

Based on https://wiki.hyperledger.org/projects/fabric/roadmap - Dates determined by the Hyperledger community - (*) Subject to change

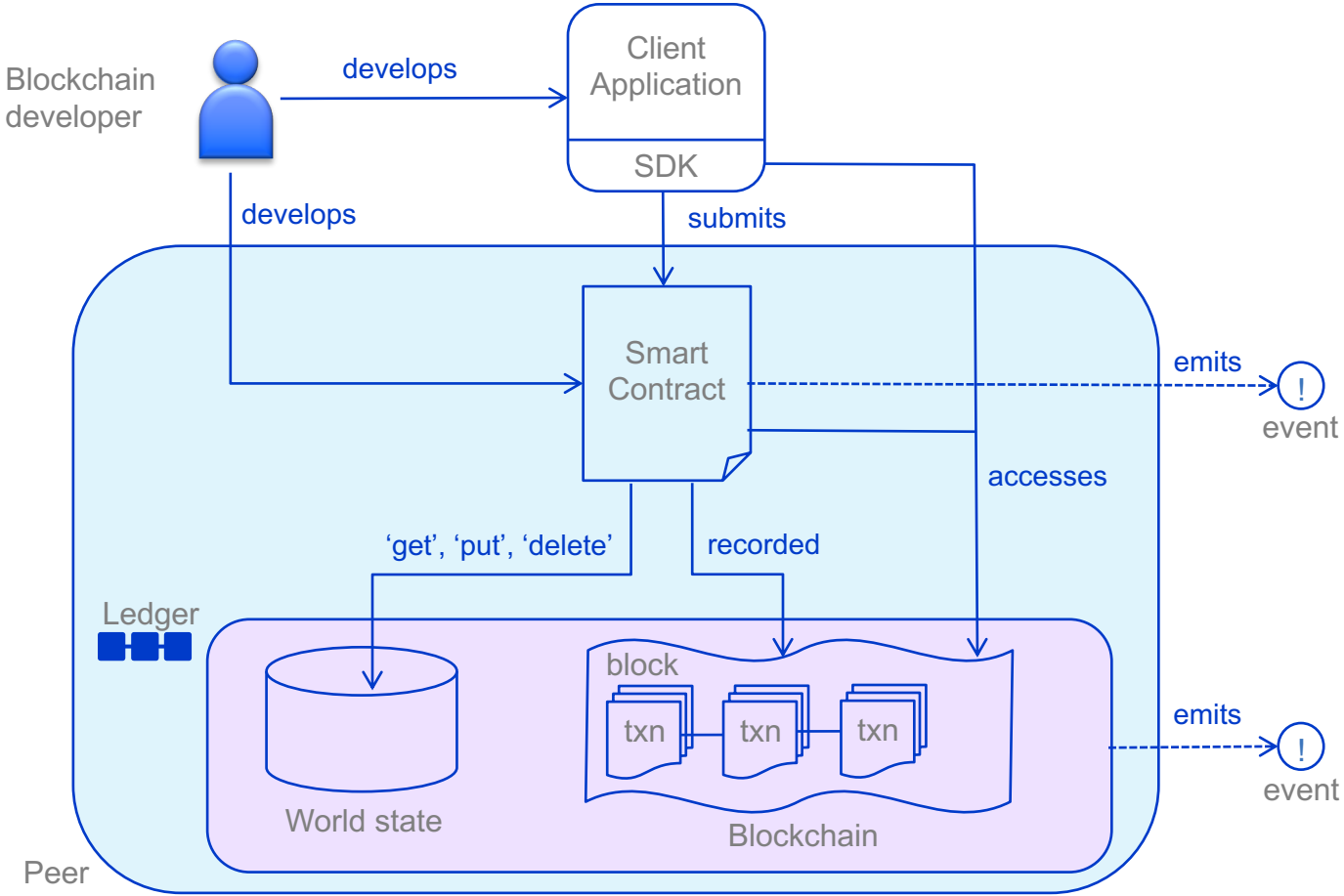
Hyperledger Fabric Technical Dive

- [Architectural Overview]
- Network Consensus
- Channels and Ordering Service
- Components
- Network setup
- Endorsement Policies
- Membership Services

Hyperledger Fabric V1.x Architecture

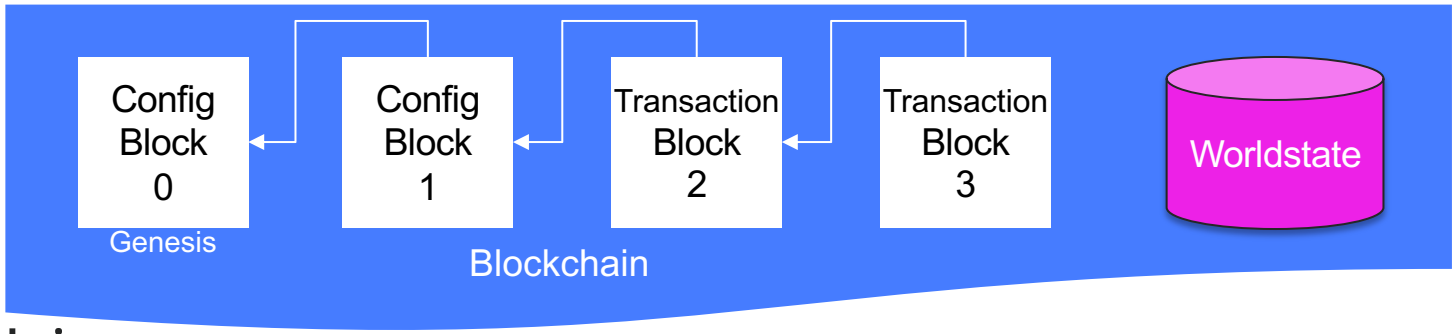


How applications interact with the ledger



Fabric Ledger

- The Fabric ledger is maintained by each peer and includes the blockchain and worldstate
- A separate ledger is maintained for each channel the peer joins
- Transaction read/write sets are written to the blockchain
- Channel configurations are also written to the blockchain
- The worldstate can be either LevelDB (default) or CouchDB
  - LevelDB is a simple key/value store
  - CouchDB is a document store that allows complex queries
- The smart contact Contract decides what is written to the worldstate



Hyperledger Fabric Technical Dive

- Architectural Overview
- [Network Consensus]
- Channels and Ordering Service
- Components
- Network setup
- Endorsement Policies
- Membership Services

A simple **consensus** example (1/4) – a puzzle!

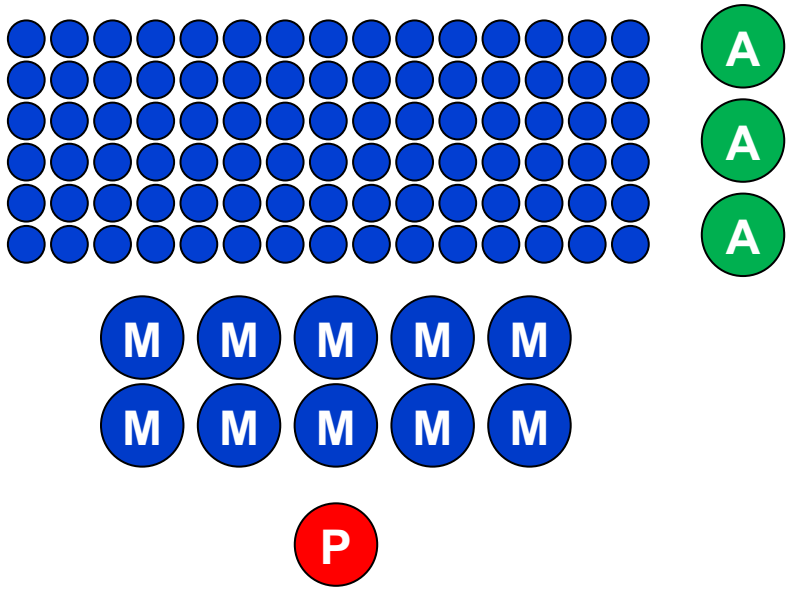
- **Situation:** A presenter in a room with 100 people, 10 of whom are mathematicians!
- **Problem:** Answer the question: **What is the square root of 2401?**
- **Question:** Can everyone agree on an answer?

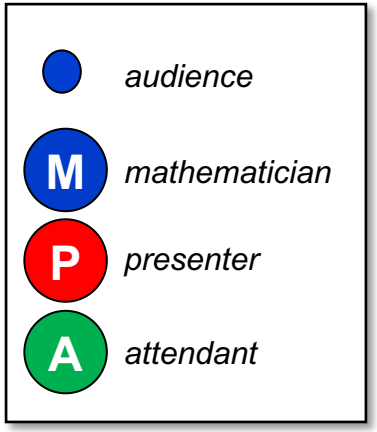
a puzzle
a process
a problem
a policy

You are here! →

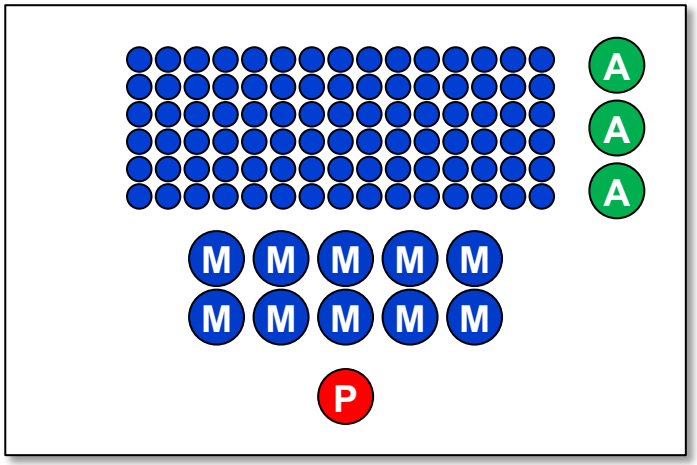
Or here! →

But not here ;) →





A simple **consensus** example (2/4) – a process

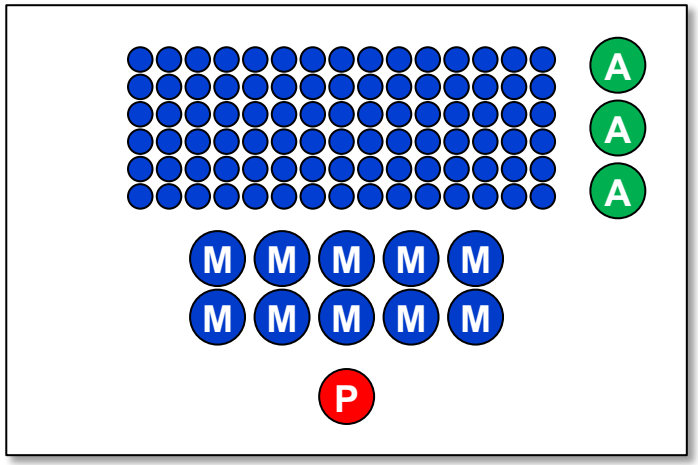


a puzzle
a process
a problem
a policy

Solution approach

1. The presenter asks every mathematician to calculate √2401
2. Every mathematician writes their answer on a piece of paper, signs with their (digital) signature
3. The presenter collects every mathematician's signed response
4. The room attendants distribute copies of the signed answer to every member of the audience
5. Every audience member checks that every mathematician agrees the answer, or not!
6. Everyone is happy; there is **consensus!**

A simple **consensus** example (3/4) – a problem



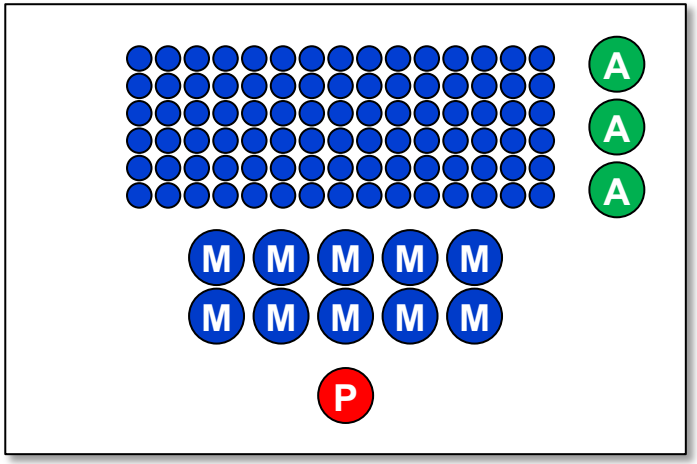
a puzzle
a process
a problem
a policy

Issues

1. How many mathematicians should the presenter ask?
2. How many mathematicians need to agree for an answer to be correct?
3. What happens if the mathematics disagree on the answer, or the presenter selects a bad answer?

→ Everyone needs to agree what constitutes a **good** answer

A simple **consensus** example (4/4) – a policy



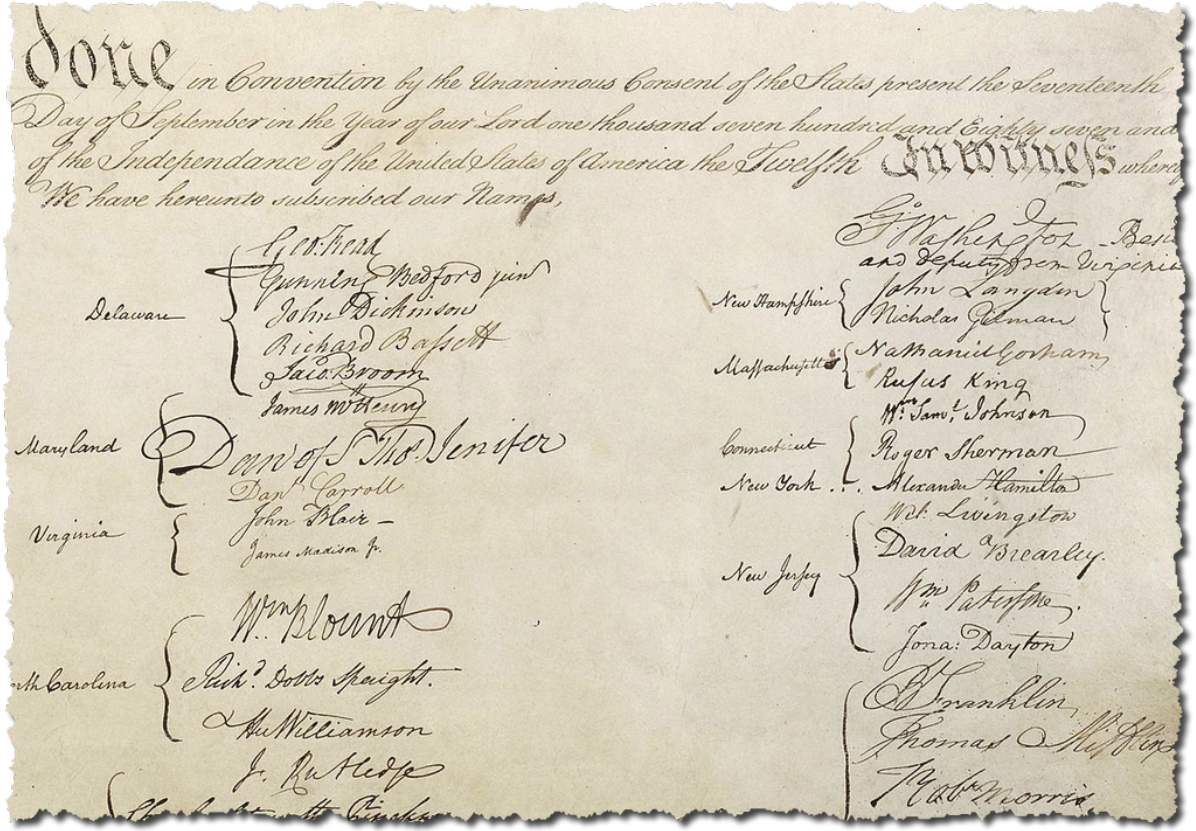
a puzzle
a process
a problem
a policy

The importance of policy

1. The audience agrees a **policy for good answers**
2. Example policies:
   - **all mathematicians must agree**
   - **the majority of mathematicians must agree**

→ Using this **policy** everyone can now agree what makes a **good** answer

Transaction Endorsement



Hyperledger Fabric **multi-party transactions**

- A transaction describes a **change to a system**

  Examples:
  - a change in bank balance
  - a student receives an academic qualification
  - a parcel is delivered

- Traditionally transactions are signed by a **single organization**
  - a bank signs payment transactions
  - a university signs graduation transactions
  - a logistics company signs parcel receipt transactions

- **Multi-party transactions** are **the heart** of Hyperledger Fabric
  - e.g. buyer and seller both sign car transfer transaction
  - e.g. logistics and delivery companies both sign parcel transaction

- **Understand multi-party transactions and you will understand Fabric!**

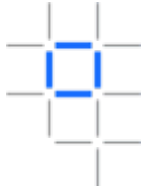
```
car transfer transaction:

 identifier: 1234567890

 proposal:
  input: {CAR1, seller, buyer}
  signature: input*seller

 response:
  output:
    {CAR1.oldOwner=seller,
     CAR1.newOwner=buyer}
  signatures:
    output*seller
    output*buyer
```

*A transaction to transfer a car is signed by both the **buyer** and the **seller***

Nodes and roles

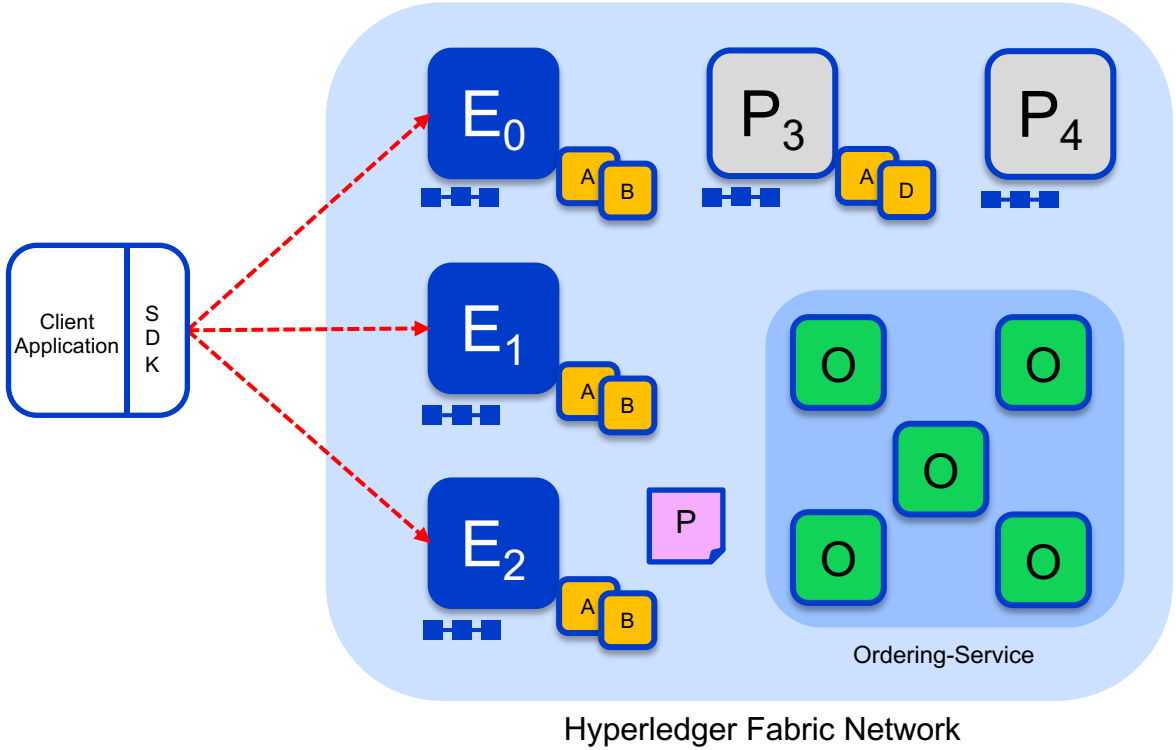
	**Peer:** Maintains ledger and state. Commits transactions. May hold smart contract (chaincode).
	**Endorsing Peer:** Specialized peer also endorses transactions by receiving a transaction proposal and responds by granting or denying endorsement. Must hold smart contract.
	**Ordering Node:** Approves the inclusion of transaction blocks into the ledger and communicates with committing and endorsing peer nodes. Does not hold smart contract. Does not hold ledger.

Hyperledger Fabric Consensus



Consensus is achieved using the following transaction flow:

Sample transaction: Step 1/7 – Propose transaction

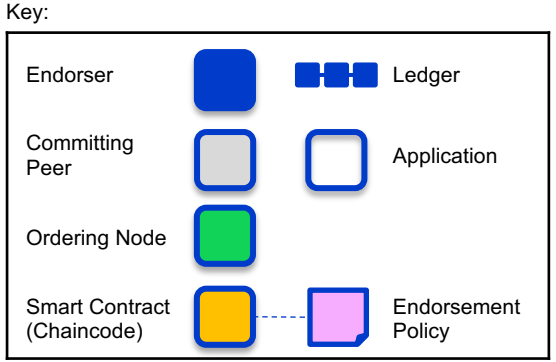


Hyperledger Fabric Network

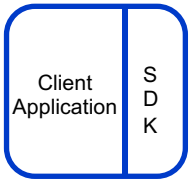
Application proposes transaction

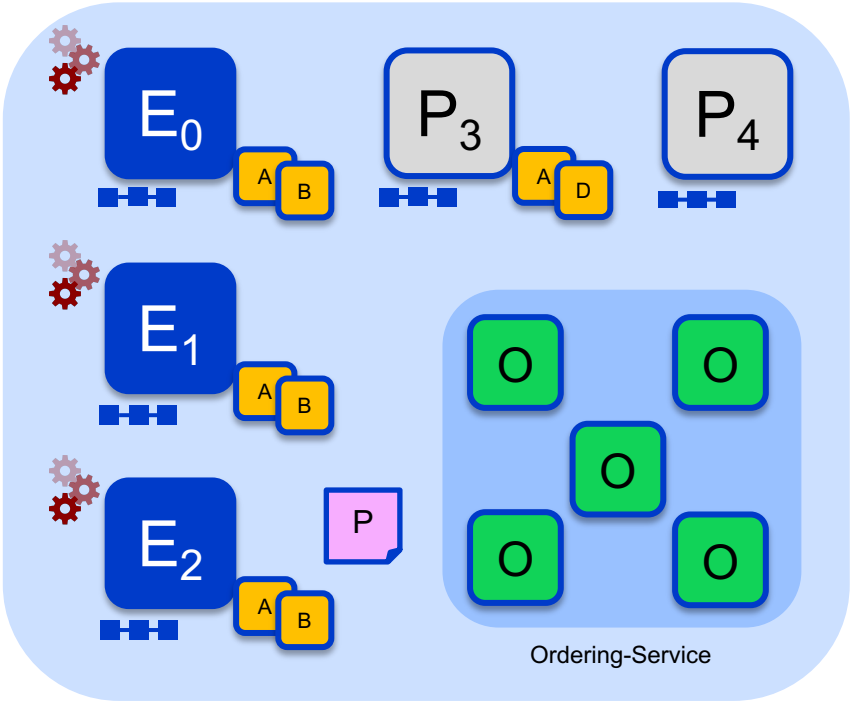
Endorsement policy:
- "E_0, E_1 and E_2 must sign"
- (P_3, P_4 are not part of the policy)

Client application submits a transaction proposal for Smart Contract A. It must target the required peers {E_0, E_1, E_2}



Sample transaction: Step 2/7 – Execute proposal





Hyperledger Fabric Network

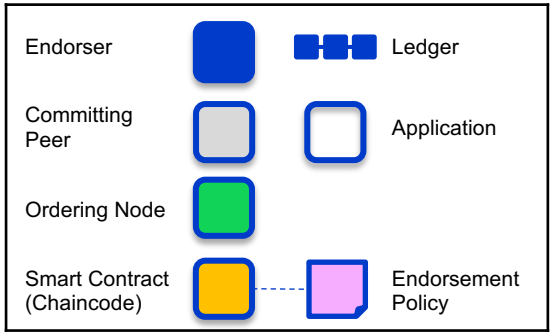
Endorsers Execute Proposals

E_0, E_1 & E_2 will each execute the proposed transaction. None of these executions will update the ledger

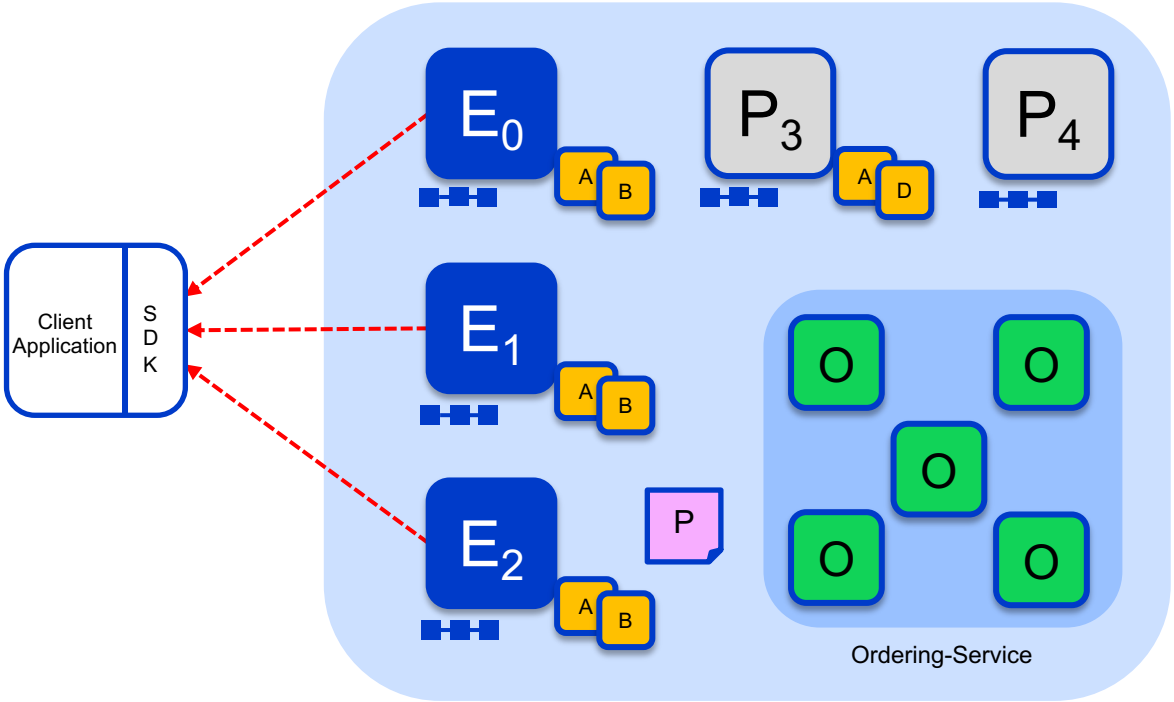
Each execution will capture the set of Read and Written data, called RW sets, which will now flow in the fabric.

Transactions can be signed & encrypted

Key:



Sample transaction: Step 3/7 – Proposal Response



Hyperledger Fabric Network

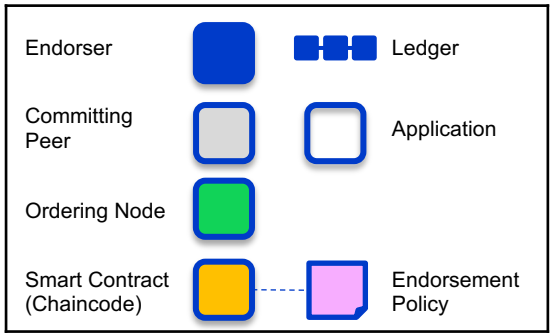
Application receives responses

RW sets are asynchronously returned to application

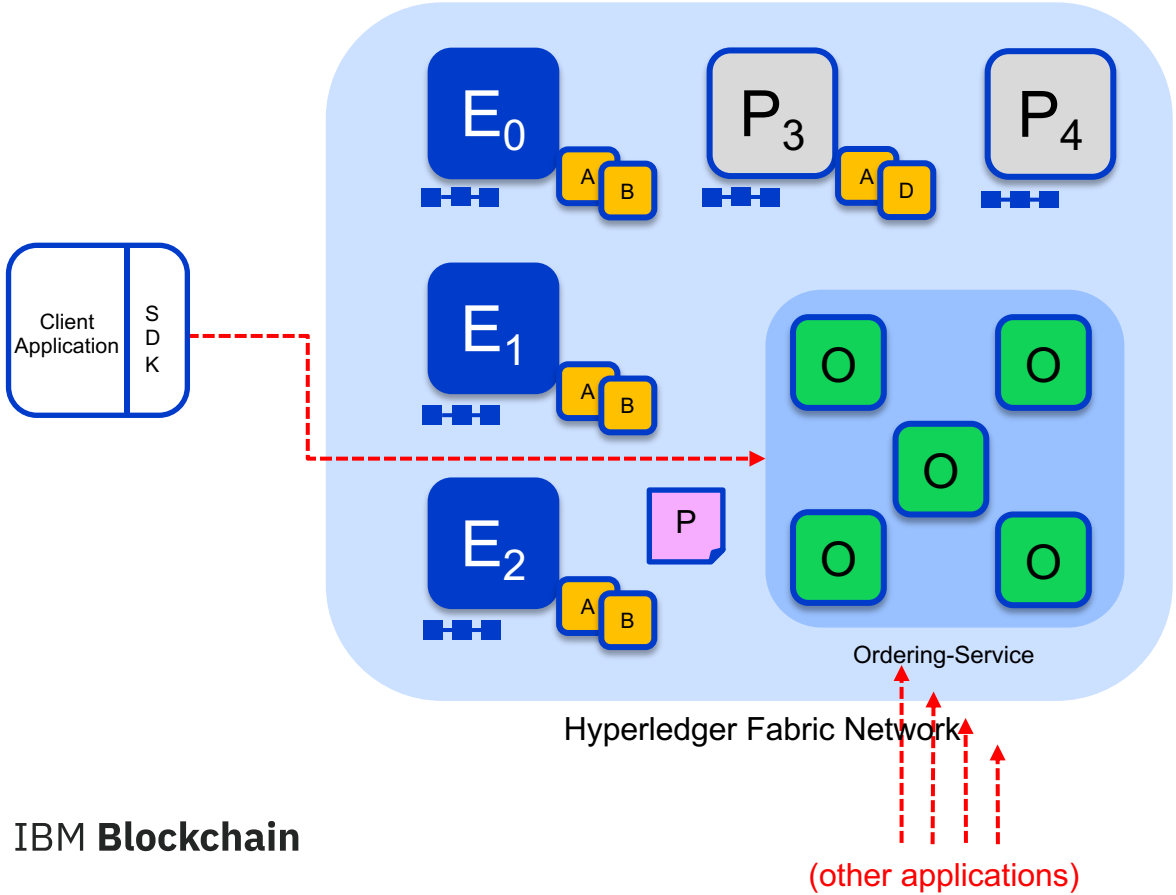
The RW sets are signed by each endorser, and also includes each record version number

(This information will be checked much later in the consensus process)

Key:



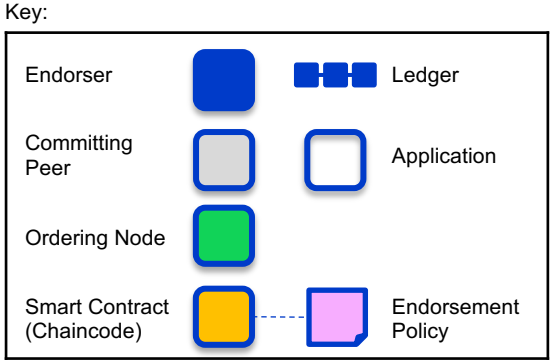
Sample transaction: Step 4/7 – Order Transaction



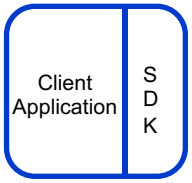
Responses submitted for ordering

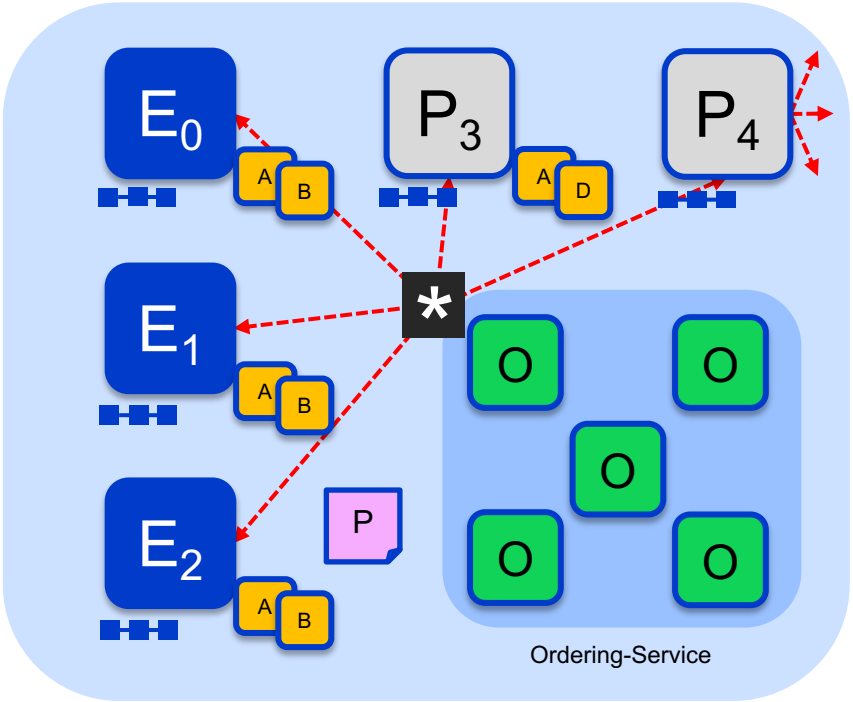
Application submits responses as a transaction to be ordered.

Ordering happens across the fabric in parallel with transactions submitted by other applications



Sample transaction: Step 5/7 – Deliver Transaction





Hyperledger Fabric Network

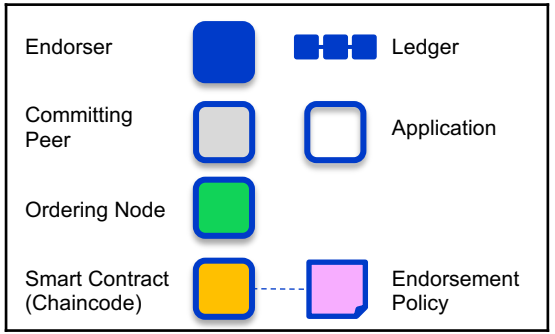
Orderer delivers to committing peers

Ordering service collects transactions into proposed blocks for distribution to committing peers. Peers can deliver to other peers in a hierarchy (not shown)

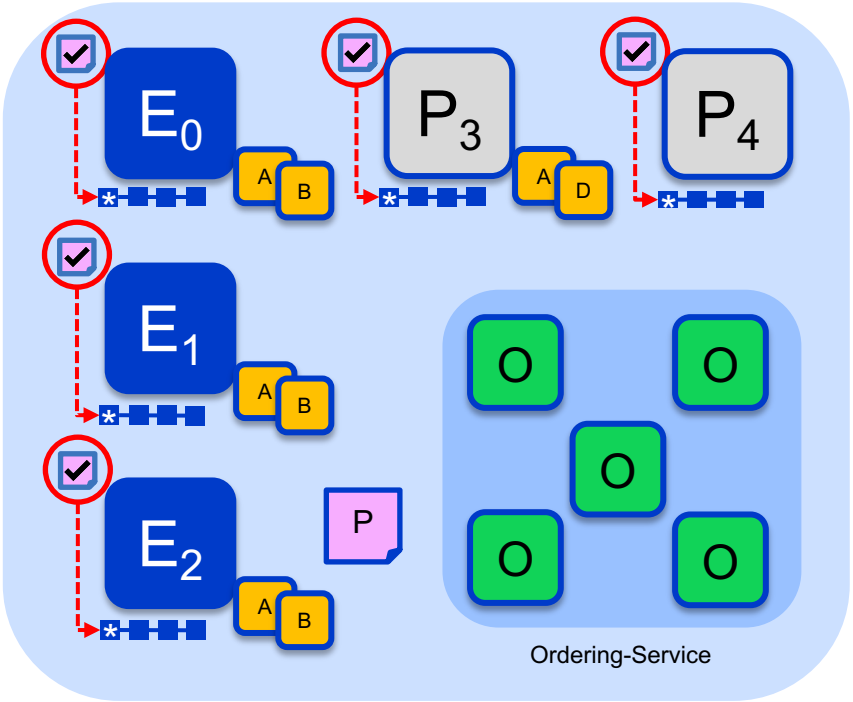
Different ordering algorithms available:

- SOLO (Single node, development)
- Kafka (Crash fault tolerance)

Key:



Sample transaction: Step 6/7 – Validate Transaction



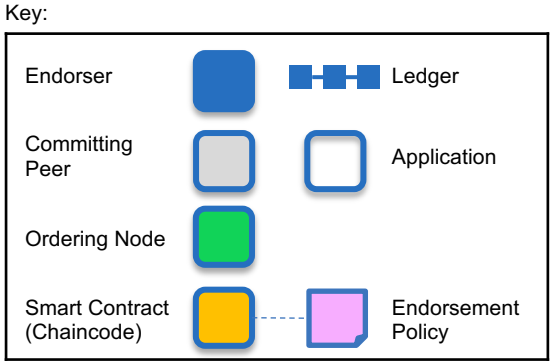
Hyperledger Fabric Network

Committing peers validate transactions

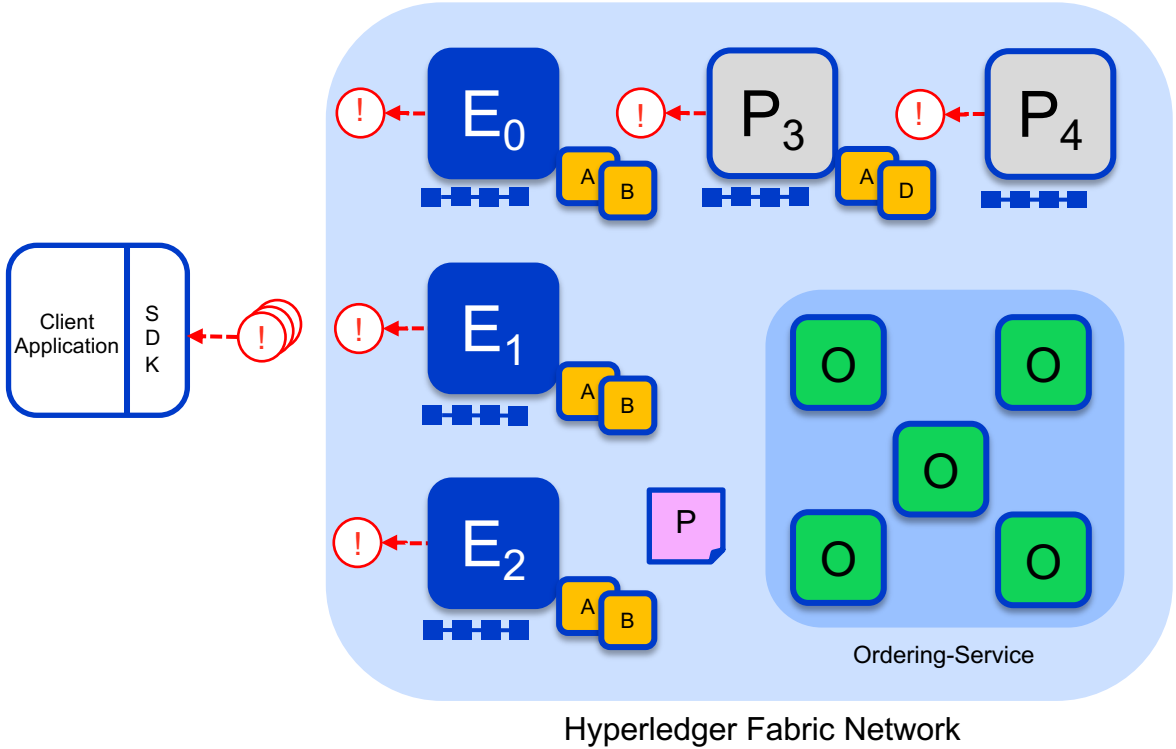
Every committing peer validates against the endorsement policy. Also check RW sets are still valid for current world state

Validated transactions are applied to the world state and retained on the ledger

Invalid transactions are also retained on the ledger but do not update world state



Sample transaction: Step 7/7 – Notify Transaction

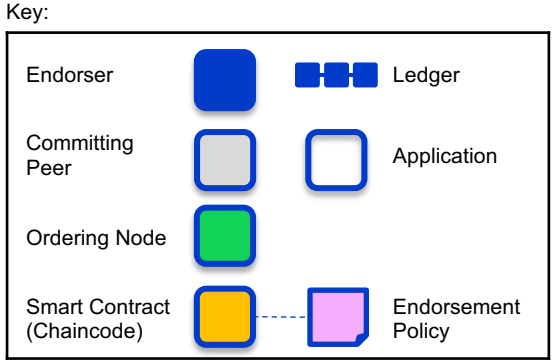


Hyperledger Fabric Network

Committing peers notify applications

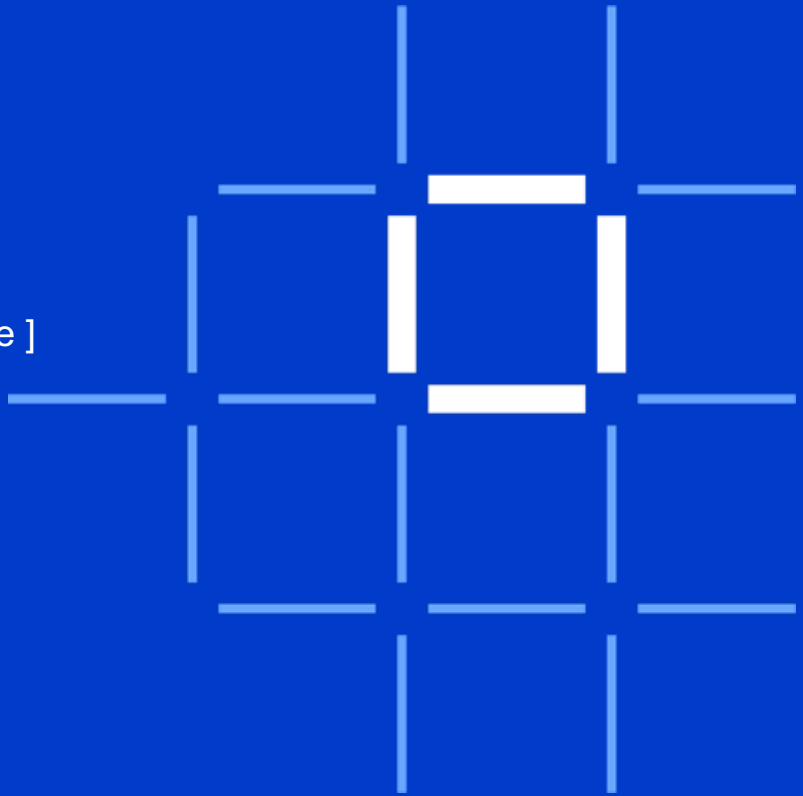
Applications can register to be notified when transactions succeed or fail, and when blocks are added to the ledger

Applications will be notified by each peer to which they are connected



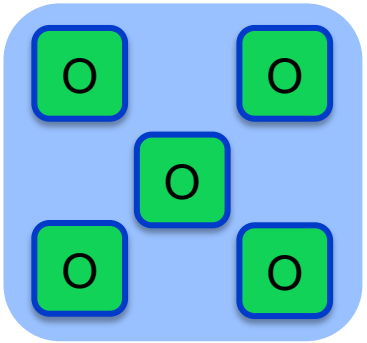
Hyperledger Fabric Technical Dive

- Architectural Overview
- Network Consensus
- [Channels and Ordering Service]
- Components
- Network setup
- Endorsement Policies
- Membership Services



Ordering Service - Raft

The ordering service packages transactions into blocks to be delivered to peers. Communication with the service is via channels.



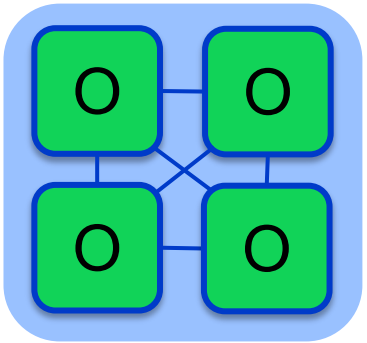
Ordering-Service

Raft

- **Crash fault tolerant** (5 nodes)
- **No additional dependencies** required
- Can be run as single node for development
- Works more efficiently than Kafka over geographically diverse networks, allowing a **distributed ordering service**

Ordering Service – Kafka/Solo

The ordering service packages transactions into blocks to be delivered to peers. Communication with the service is via channels.

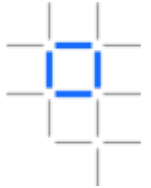


Ordering-Service

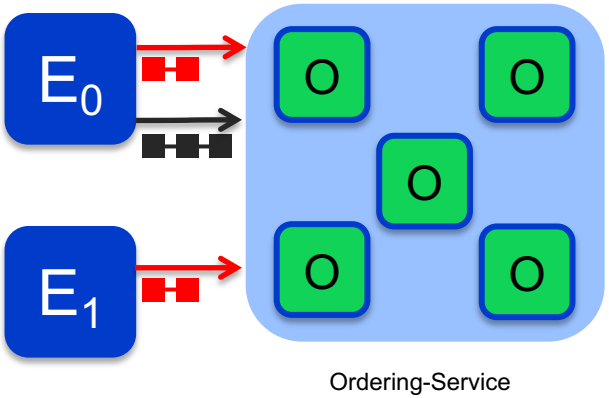
Different configuration options for the ordering service include:

- SOLO
  - Single node for development
- Kafka : Crash fault tolerant consensus
  - 3 nodes minimum
  - Odd number of nodes recommended

Channels

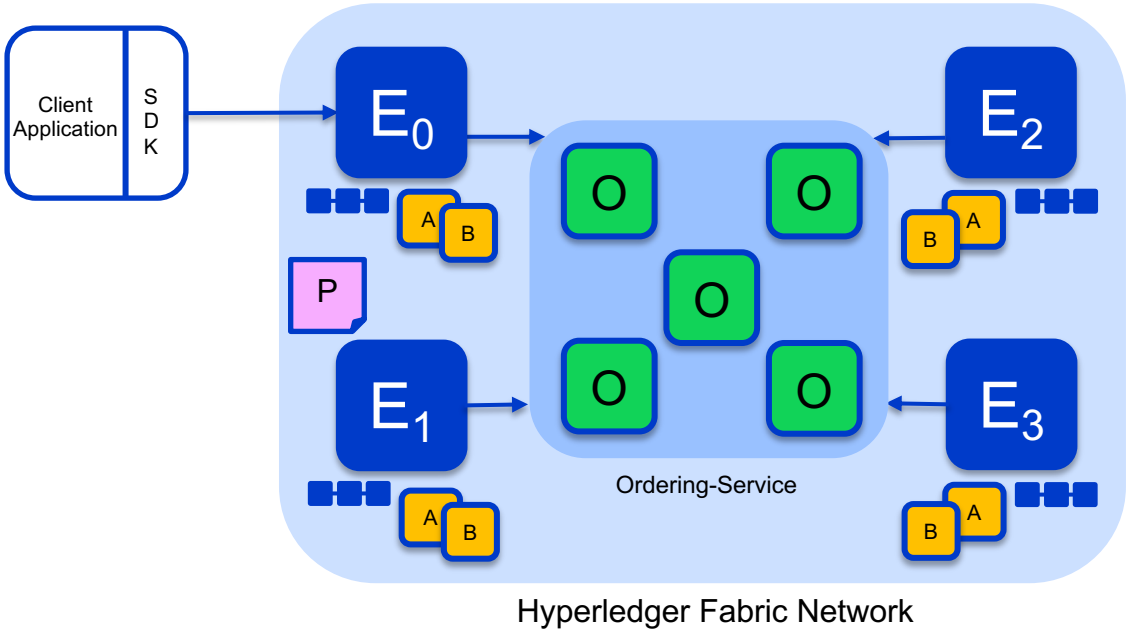


Channels provide privacy between different ledgers



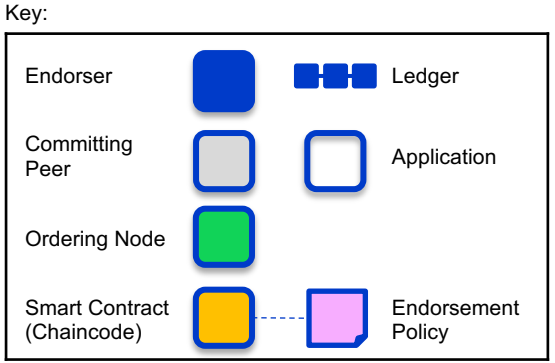
- Ledgers exist in the scope of a channel
  - Channels can be shared across an entire network of peers
  - Channels can be permissioned for a specific set of participants
- Chaincode is installed on peers to access the worldstate
- Chaincode is instantiated on specific channels
- Peers can participate in multiple channels
- Concurrent execution for performance and scalability

Single Channel Network

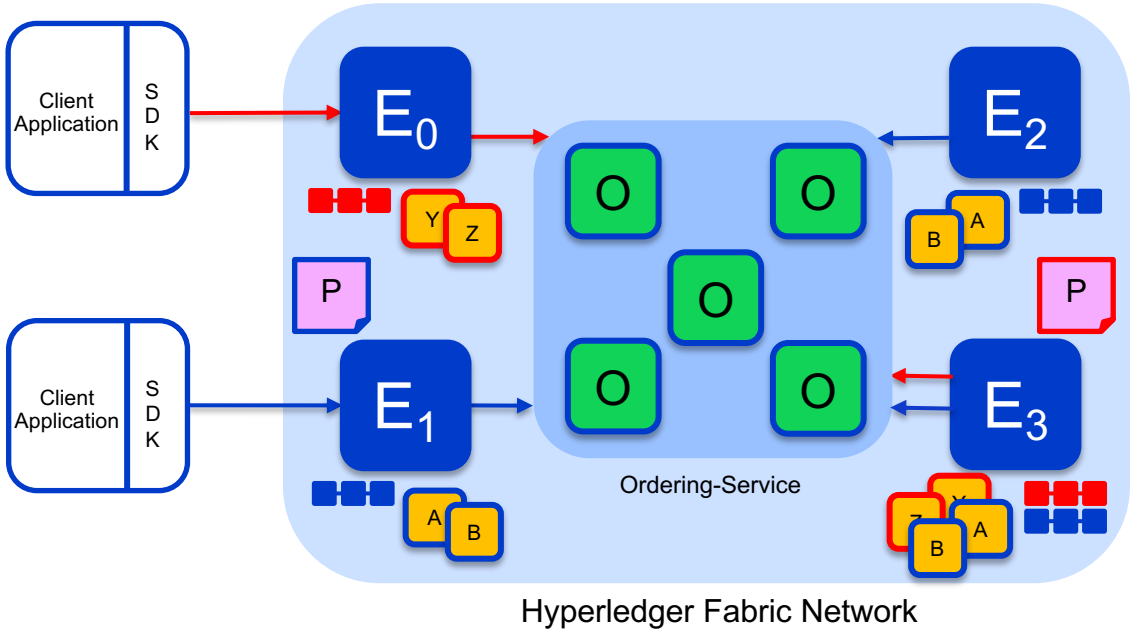


Hyperledger Fabric Network

- Similar to v0.6 PBFT model
- All peers connect to the same system channel (blue).
- All peers have the same chaincode and maintain the same ledger
- Endorsement by peers E_0, E_1, E_2 and E_3

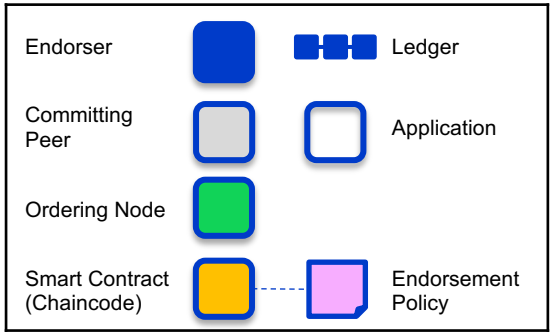


Multi Channel Network

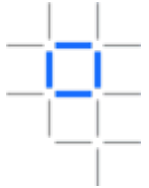


- Peers E_0 and E_3 connect to the red channel for chaincodes Y and Z
- E_1, E_2 and E_3 connect to the blue channel for chaincodes A and B

Key:



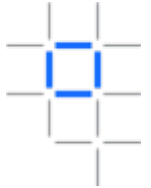
Private Data Collections (FAB-1151)



Allows data to be private to only a set of authorized peers

Fabric 1.0	Fabric 1.1
• Data privacy across channels only	• Data privacy within a channel
• Transaction proposal and worldstate read/write sets visible to all peers connected to a channel	• Transaction proposal and worldstate read/write sets available to only permissioned peers
• Ordering service has access to transactions including the read/write sets	• Ordering service has only evidence of transactions (hashes)
	• Complements Fabric 1.0 channel architecture
	• Policy defines which peers have private data

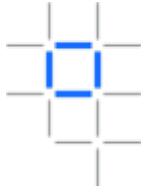
Private Data Collections - Explained



1. Private data:
    1. Excluded from transactions by being sent as ‘transient data’ to endorsing peers.
    2. Shared peer-to-peer with only peers defined in the collection policy.
2. Hashes of private data included in transaction proposal for evidence and validation.
    1. Peers/Orderers not in the collection policy have only hashes.
3. Peers maintain both a public worldstate and private worldstate.
4. Private data held in a transient store between endorsement and validation.

https://jira.hyperledger.org/browse/FAB-1151

Private Data Collections – Marble Scenario



Privacy Requirements:

- No marble data should go through ordering service as part of a transaction
- All peers have access to general marble information
  - *Name, Size, Color, Owner*
- Only a subset of peers have access to marble *pricing* information

Transaction
- Primary read/write set (if exists)
- Hashed private read/write set (hashed keys/values)

Transaction
- Public channel data
- Goes to all orderers/peers

Collection: Marbles
- Private Write Set
- Name, Size, Color, Owner

Policy: Org1, Org2

```
"requiredPeerCount": 1,
"maxPeerCount":2,
"blockToLive":1000000
```

Collection: Marbles
- Private data for channel peers
- Goes to all peers but not orderers

Collection: Marble Private Details
- Private Write Set
- Price

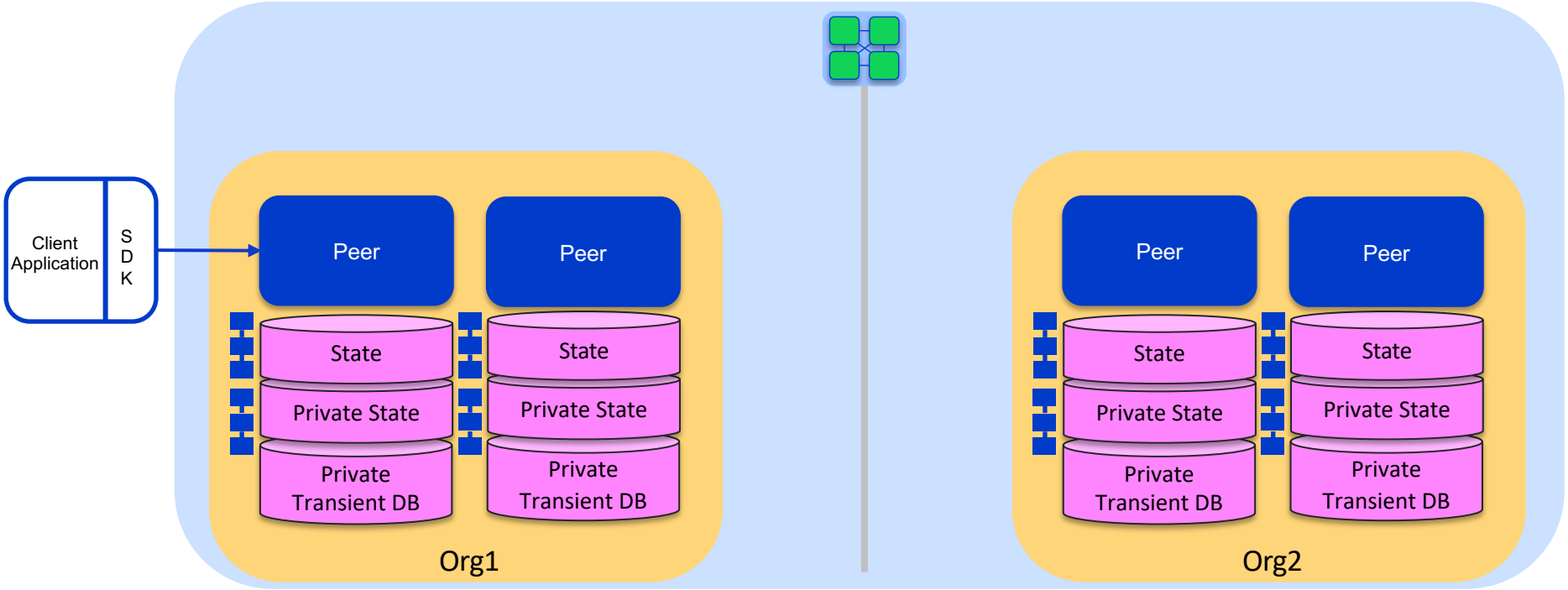
Policy: Org1

```
"requiredPeerCount": 1,
"maxPeerCount": 1,
"blockToLive":3
```

Collection: Marbles Private Details
- Private data for subset of channel peers
- Goes to subset of peers only

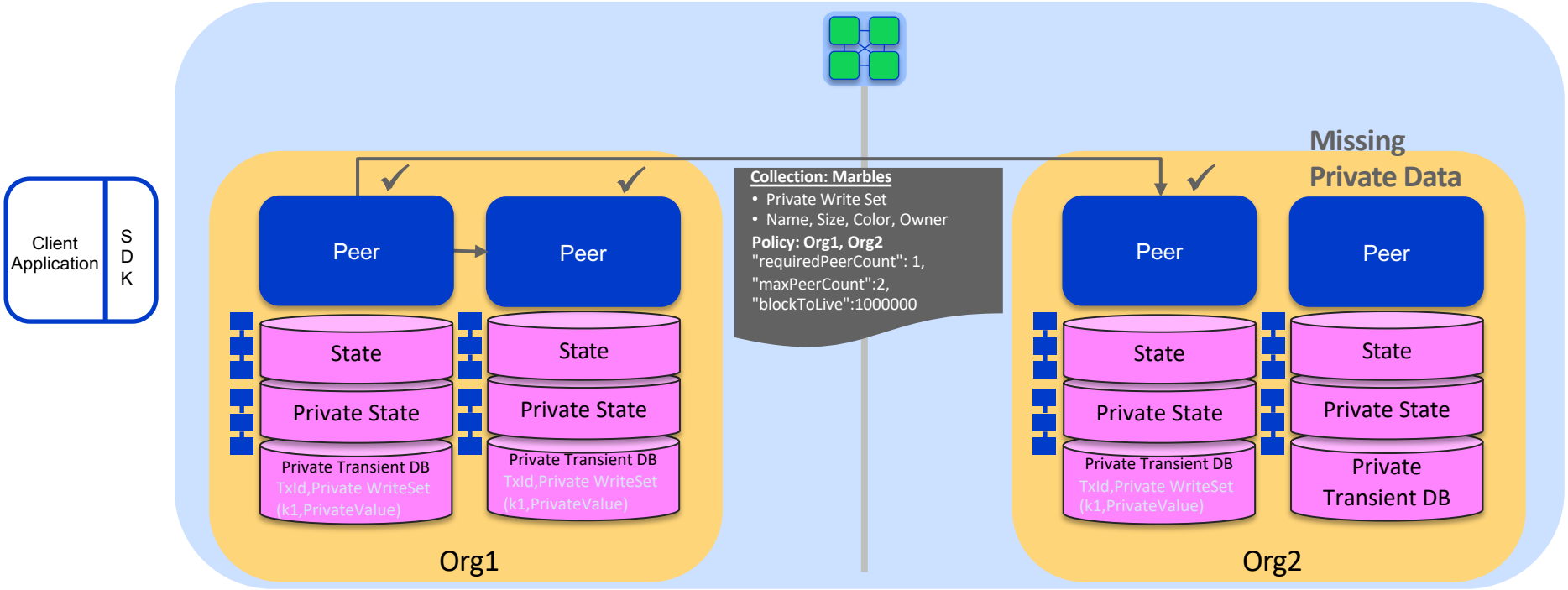
Step 1: Propose Transaction

Client sends proposal to endorsing peer(s)



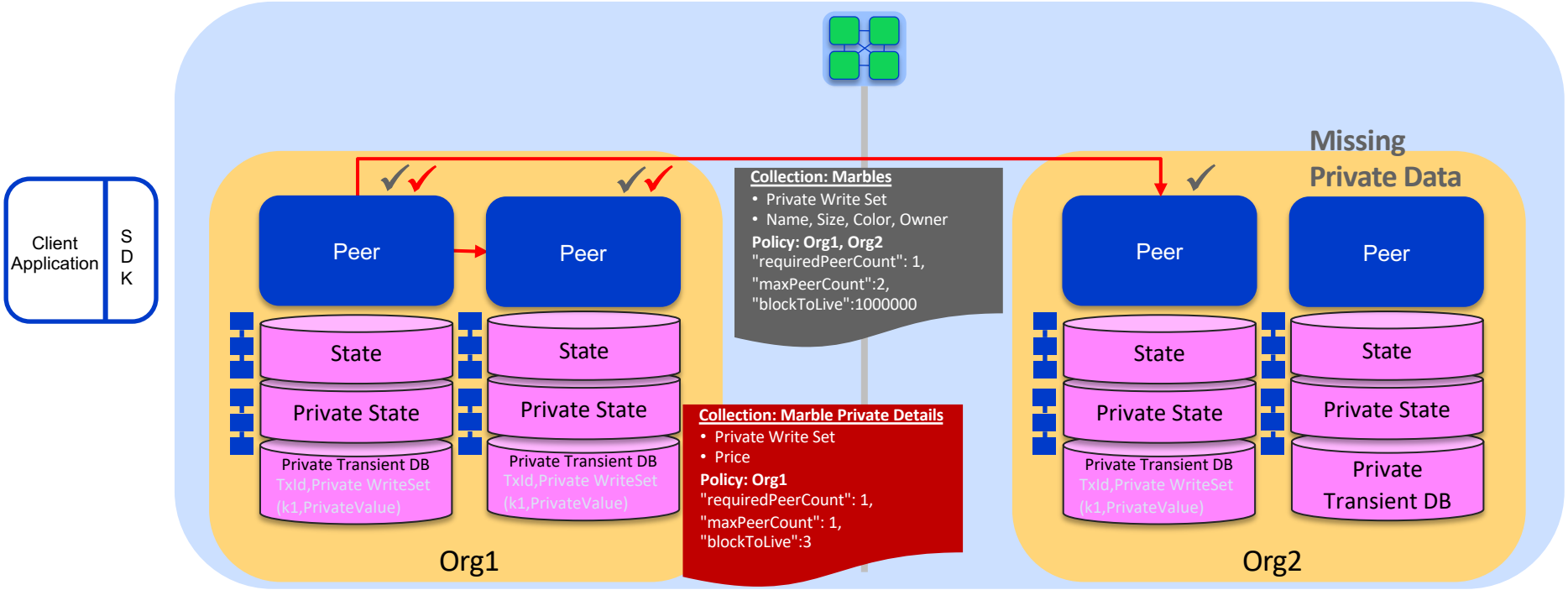
Step 2a: Execute Proposal and Distribute 1st Collection

Endorsing peer simulates transaction and distributes **marbles collection** data based on policy



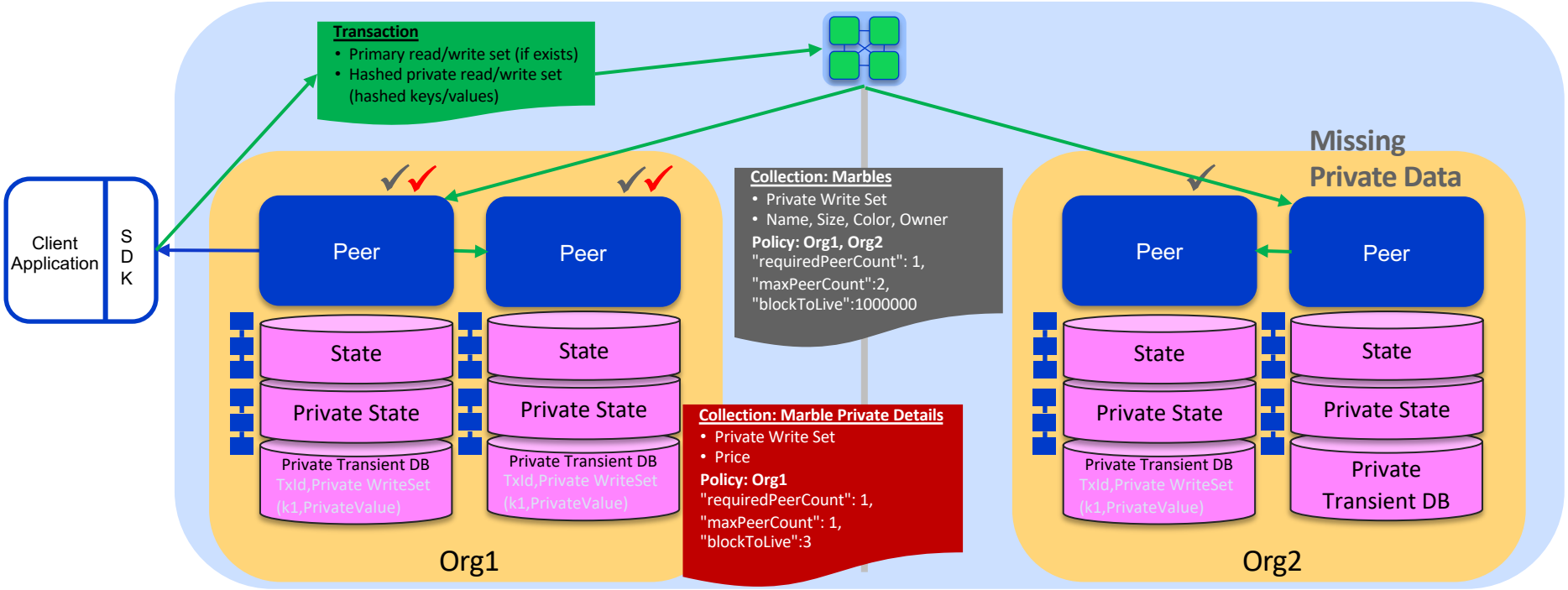
Step 2b: Distribute 2nd Collection

Endorsing peer distributes **marbles private details collection** data based on policy



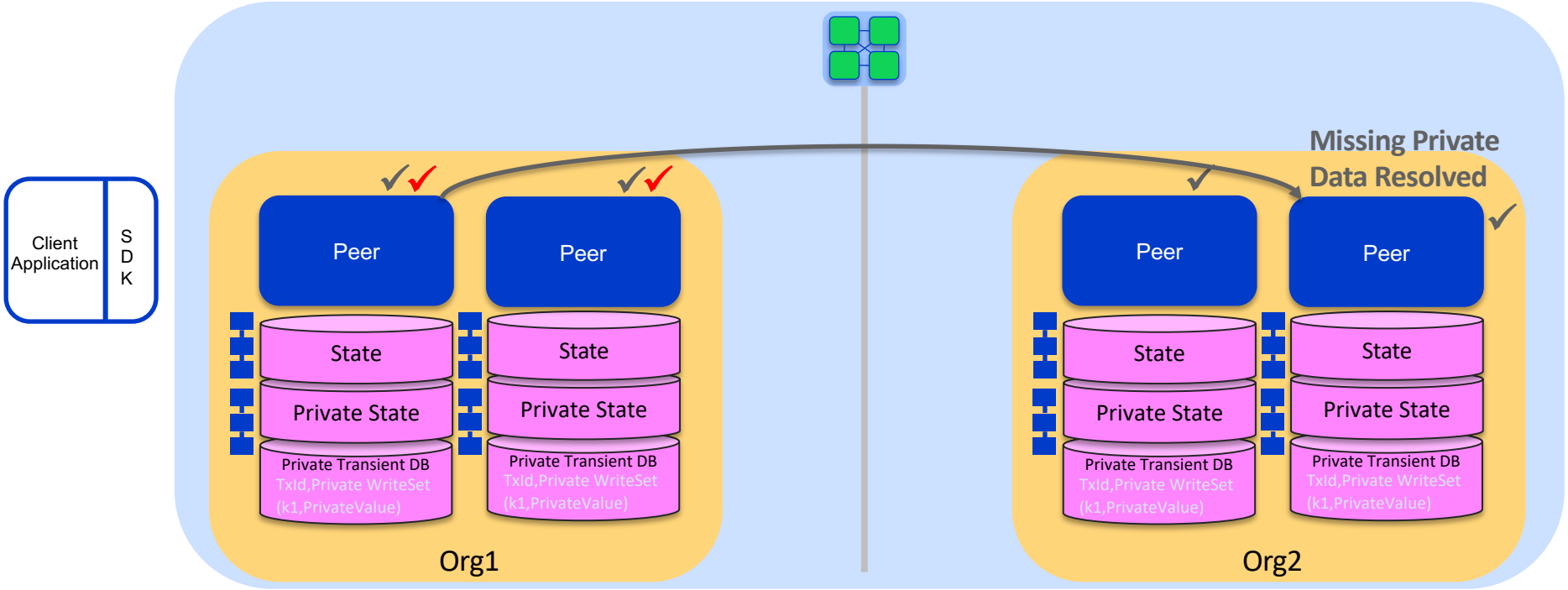
Step 3: Proposal Response / Order / Deliver

Proposal response sent back to client, which then sends the proposal to the ordering service for delivery to all peers



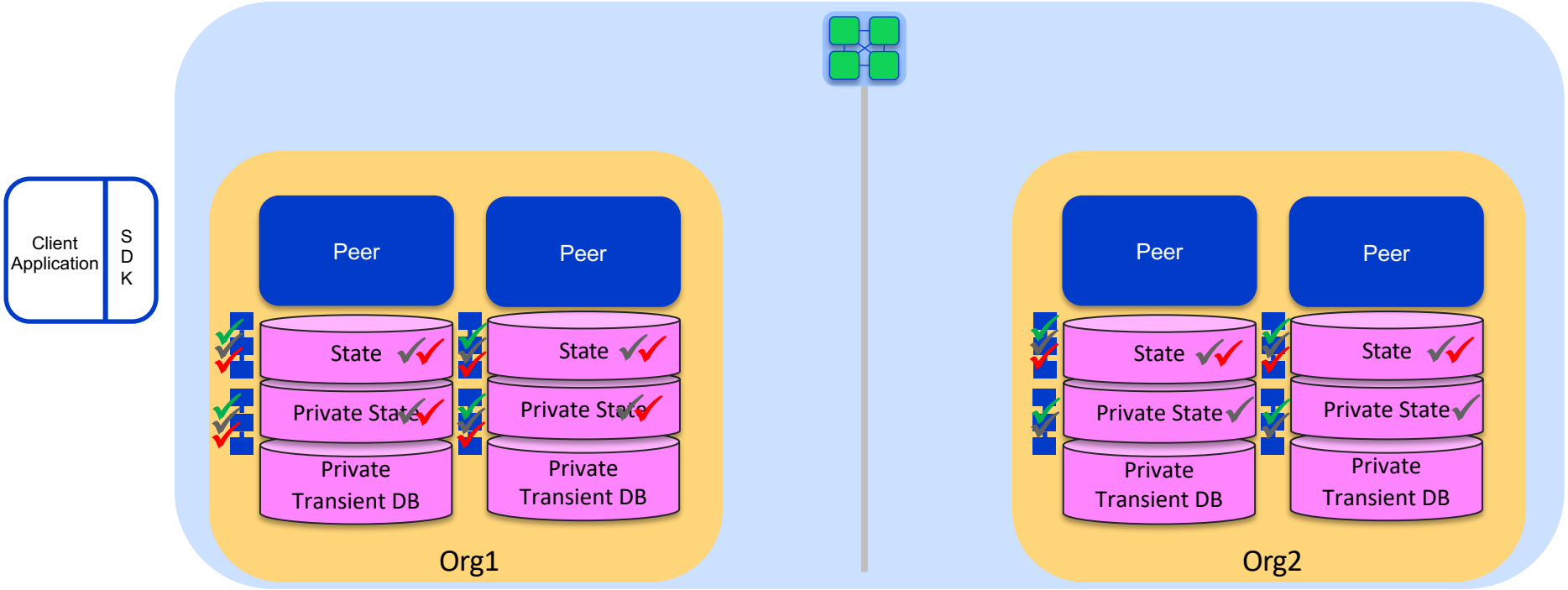
Step 4: Validate Transaction

Peers validate transactions. Private data validated against hashes. Missing private data resolved with pull requests from other peers.



Step 5: Commit

1) Commit private data to private state db. 2) Commit hashes to public state db.
3) Commit public block and private write set storage. 4) Delete transient data.

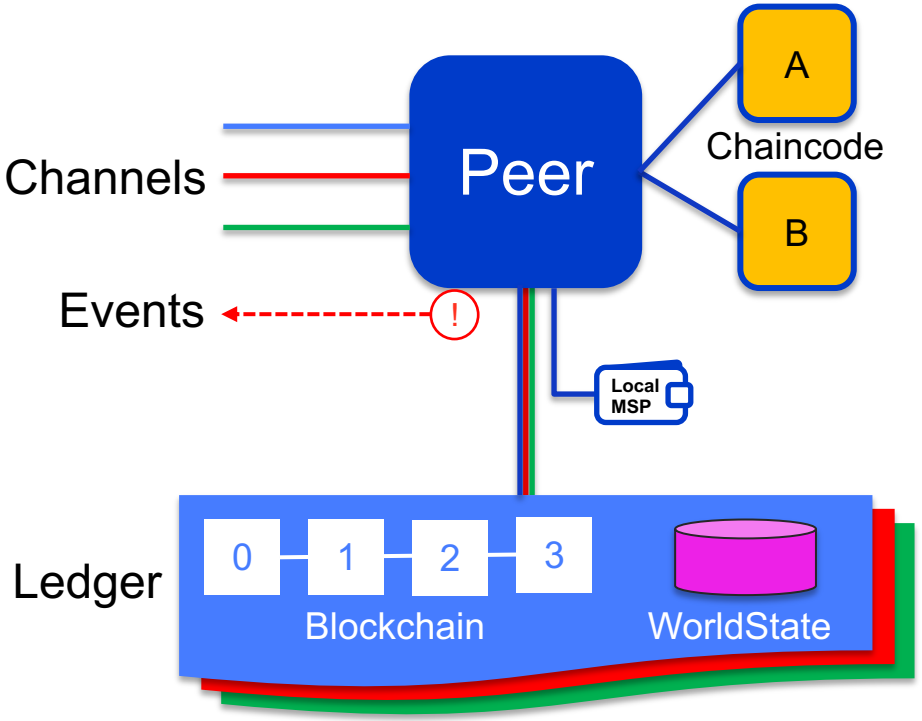


Hyperledger Fabric Technical Dive

- Architectural Overview
- Network Consensus
- Channels and Ordering Service
- [Components]
- Network setup
- Endorsement Policies
- Membership Services

Fabric Peer

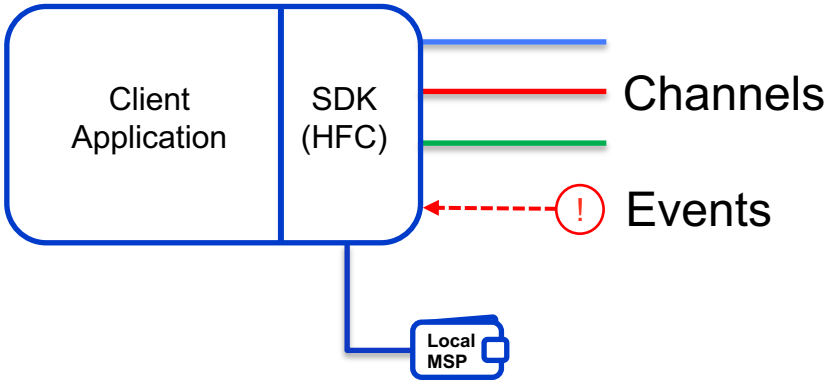
- Each peer:
  - Connects to one or more channels
  - Maintains one or more ledgers per channel
  - Maintains installed chaincode
  - Manages runtime docker containers for instantiated chaincode
    - Chaincode is instantiated on a channel
    - Runtime docker container shared by channels with same chaincode instantiated (no state stored in container)
  - Has a local MSP (Membership Services Provider) that provides crypto material
  - Emits events to the client application



Client Application

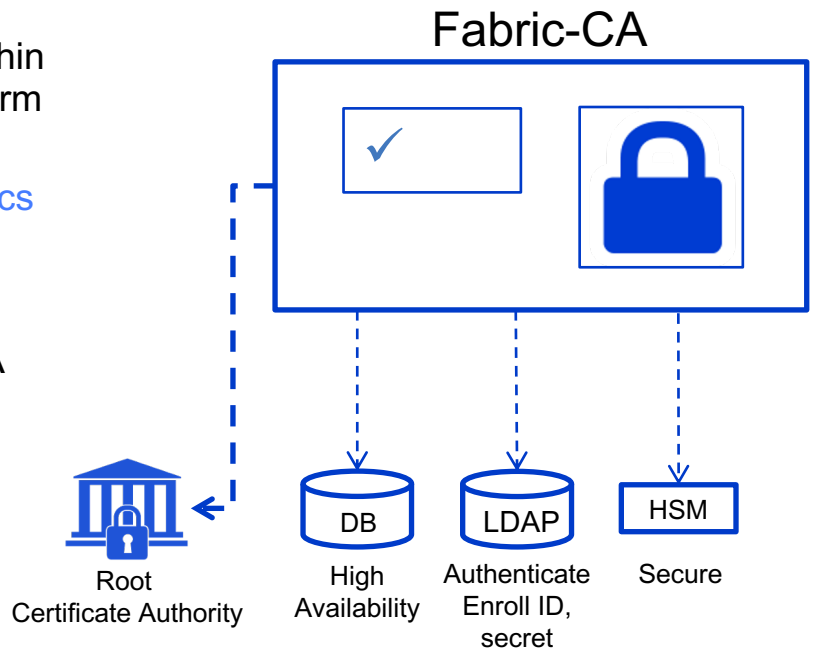
- Each client application uses Fabric SDK to:
  - Connects over channels to one or more peers
  - Connects over channels to one or more orderer nodes
  - Receives events from peers
  - Local MSP provides client crypto material

  - Client can be written in different languages (for example Node.js, Java)



Fabric-CA

- Default (optional) Certificate Authority within Fabric network for issuing Ecerts (long-term identity)
- Supports clustering(*) for HA characteristics
- Supports LDAP(*) for user authentication
- Supports HSM(*) for security
- Can be configured as an intermediate CA

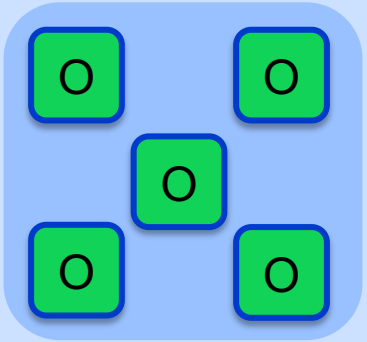


(*) Not all features currently supported by IBM Blockchain Platform

Hyperledger Fabric Technical Dive

- Architectural Overview
- Network Consensus
- Channels and Ordering Service
- Components
- [Network setup]
- Endorsement Policies
- Membership Services

Bootstrap Network (1/6) - Configure & Start Ordering Service

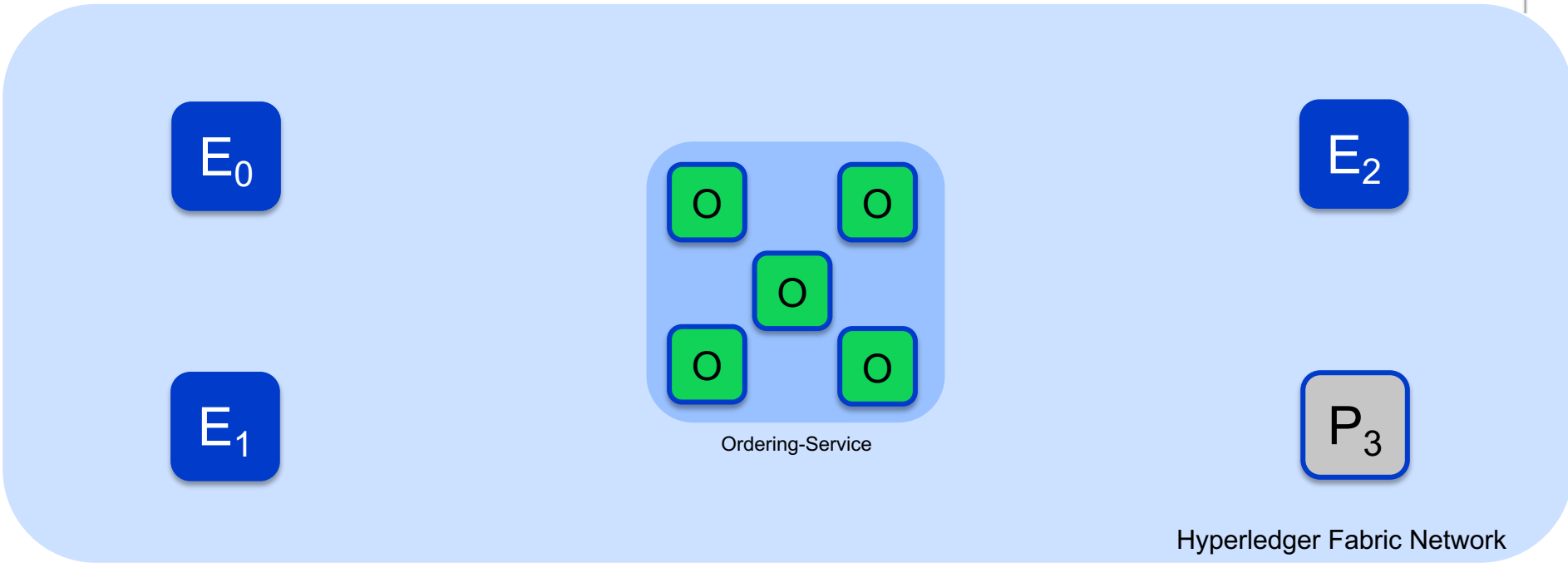


Hyperledger Fabric Network

An Ordering Service is configured and started for the network:

$ docker-compose [-f orderer.yml] ...

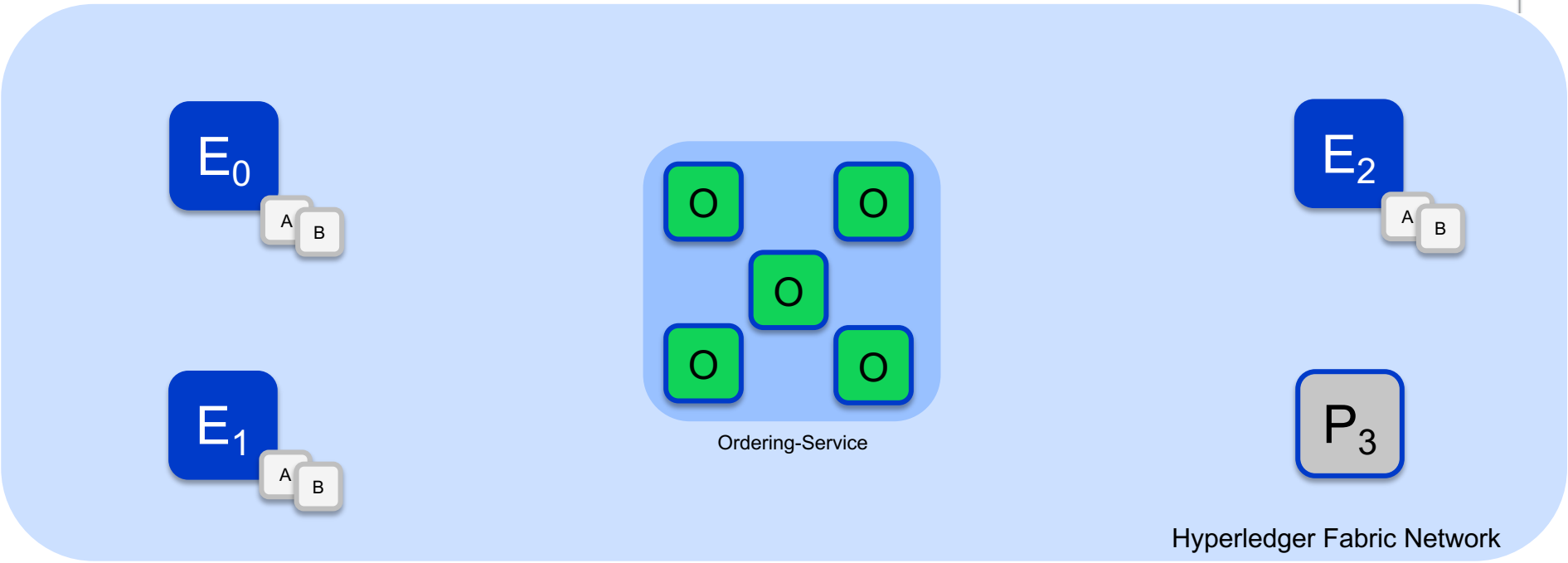
Bootstrap Network (2/6) - Configure and Start Peer Nodes



A peer is configured and started for each Endorser or Committer in the network:

$ peer node start ...

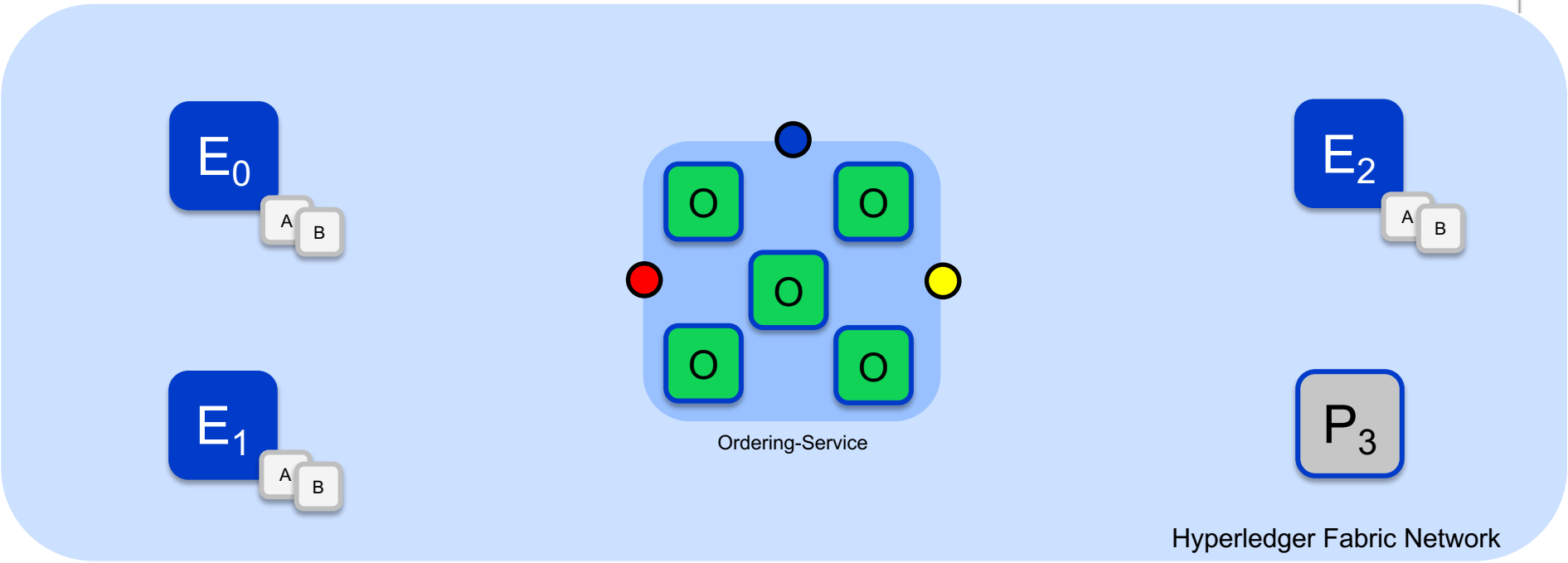
Bootstrap Network (3/6) - Install Chaincode



Chaincode is installed onto each Endorsing Peer that needs to execute it:

$ peer chaincode install ...

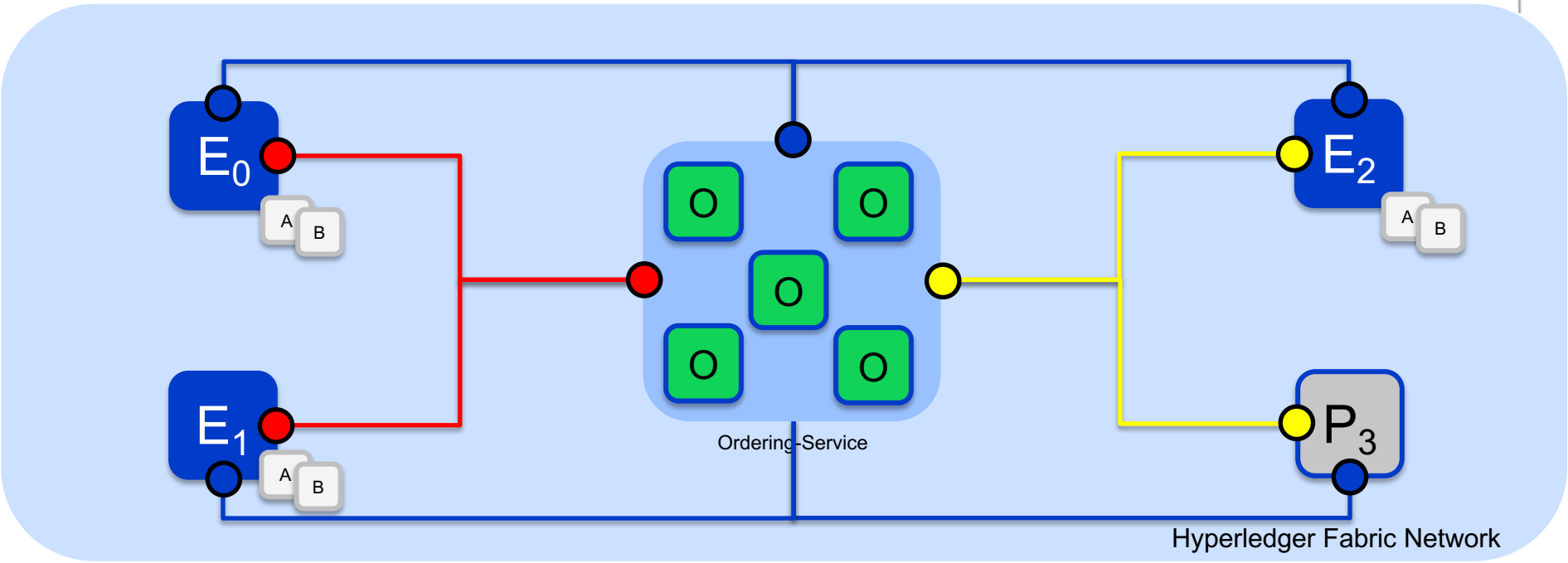
Bootstrap Network (4/6) – Create Channels



Channels are created on the ordering service:

$ peer channel create –o [orderer] ...

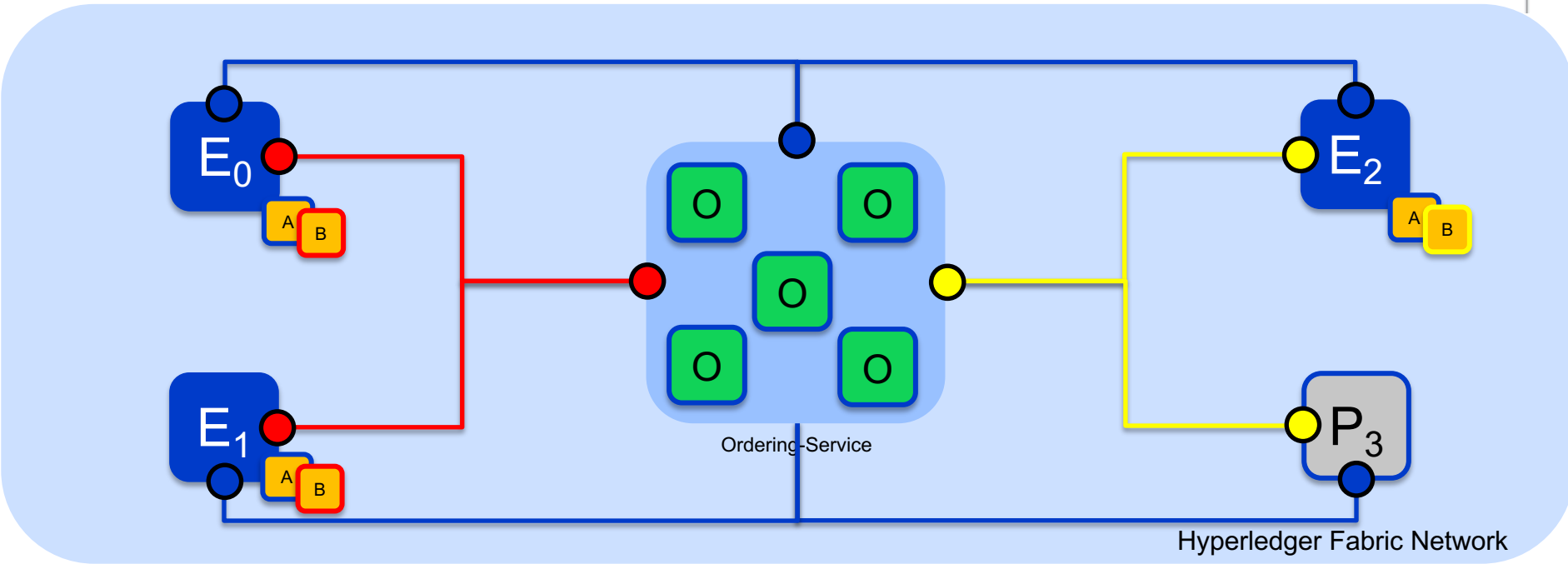
Bootstrap Network (5/6) – Join Channels



Peers that are permissioned can then join the channels they want to transact on:

$ peer channel join ...

Bootstrap Network (6/6) – Instantiate Chaincode



Peers finally instantiate the Chaincode on the channels they want to transact on:

$ peer chaincode instantiate ... –P ‘policy’

An Endorsement Policy is specified and once instantiated chaincode can process transactions.

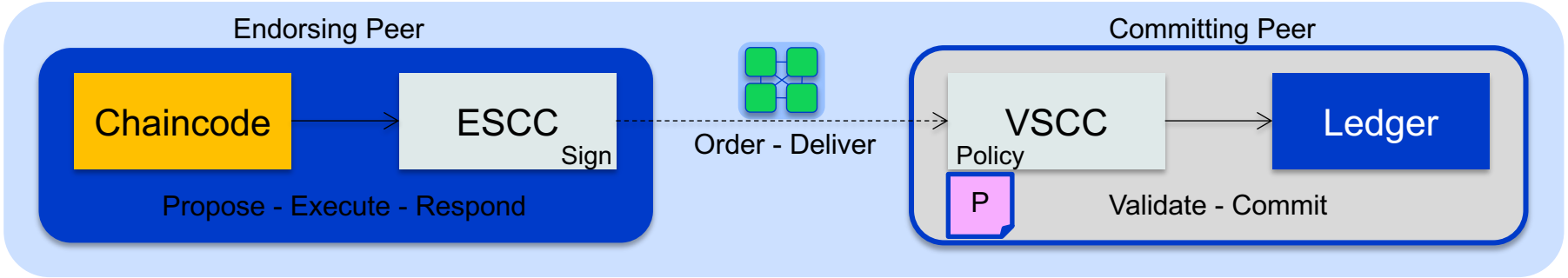
Hyperledger Fabric Technical Dive

- Architectural Overview
- Network Consensus
- Channels and Ordering Service
- Components
- Network setup
- [Endorsement Policies]
- Membership Services

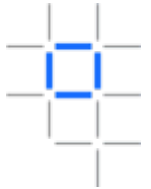
Endorsement Policies

An endorsement policy describes the conditions by which a transaction can be endorsed. A transaction can only be considered valid if it has been endorsed according to its policy.

- Each chaincode is deployed with an Endorsement Policy
- **ESCC** (Endorsement System ChainCode) signs the proposal response on the endorsing peer
- **VSCC** (Validation System ChainCode) validates the endorsements



Endorsement Policy Syntax



```
$ peer chaincode instantiate
-C mychannel
-n mycc
-v 1.0
-p chaincode_example02
-c '{"Args":["init","a", "100", "b","200"]}'
-P "AND('Org1MSP.member')"
```

Instantiate the chaincode mycc on channel mychannel with the policy AND('Org1MSP.member')

Policy Syntax: EXPR(E[, E...])

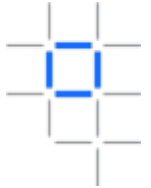
Where EXPR is either AND or OR and E is either a principal or nested EXPR

Principal Syntax: MSP.ROLE

Supported roles are: member and admin

Where MSP is the MSP ID, and ROLE is either “member” or “admin”

Endorsement Policy Examples



Examples of policies:

- Request 1 signature from all three principals
  - AND('Org1.member', 'Org2.member', 'Org3.member')
- Request 1 signature from either one of the two principals
  - OR('Org1.member', 'Org2.member')
- Request 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)
  - OR('Org1.member', AND('Org2.member', 'Org3.member'))

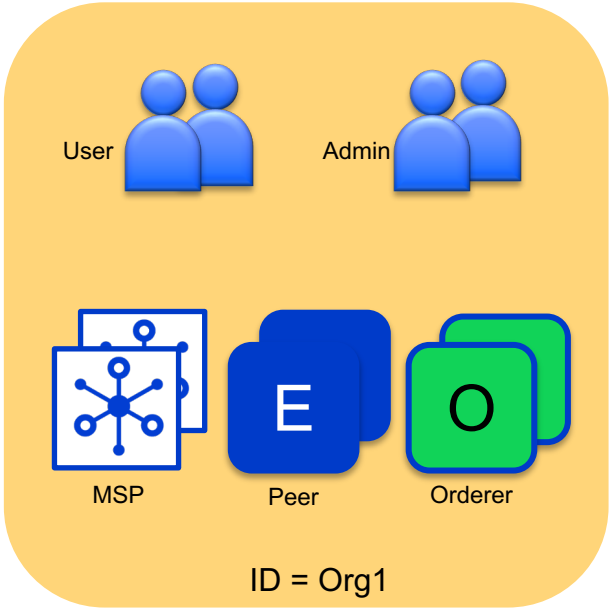
Hyperledger Fabric Technical Dive

- Architectural Overview
- Network Consensus
- Channels and Ordering Service
- Components
- Network setup
- Endorsement Policies
- [Membership Services]

Organisations

Organisations define boundaries within a Fabric Blockchain Network

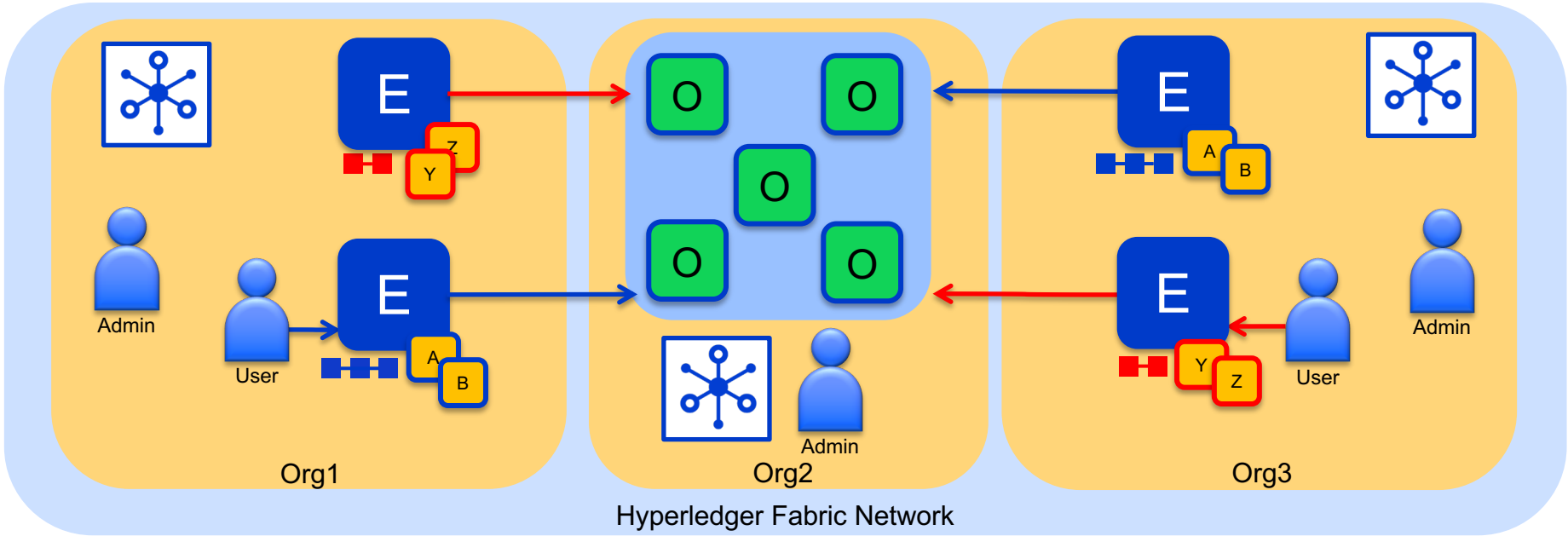
- Each organisation defines:
  - Membership Services Provider (MSP) for identities
  - Administrator(s)
  - Users
  - Peers
  - Orderers (optional)
- A network can include many organisations representing a consortium
- Each organisation has an ID



Consortium Network

An example consortium network of 3 organisations

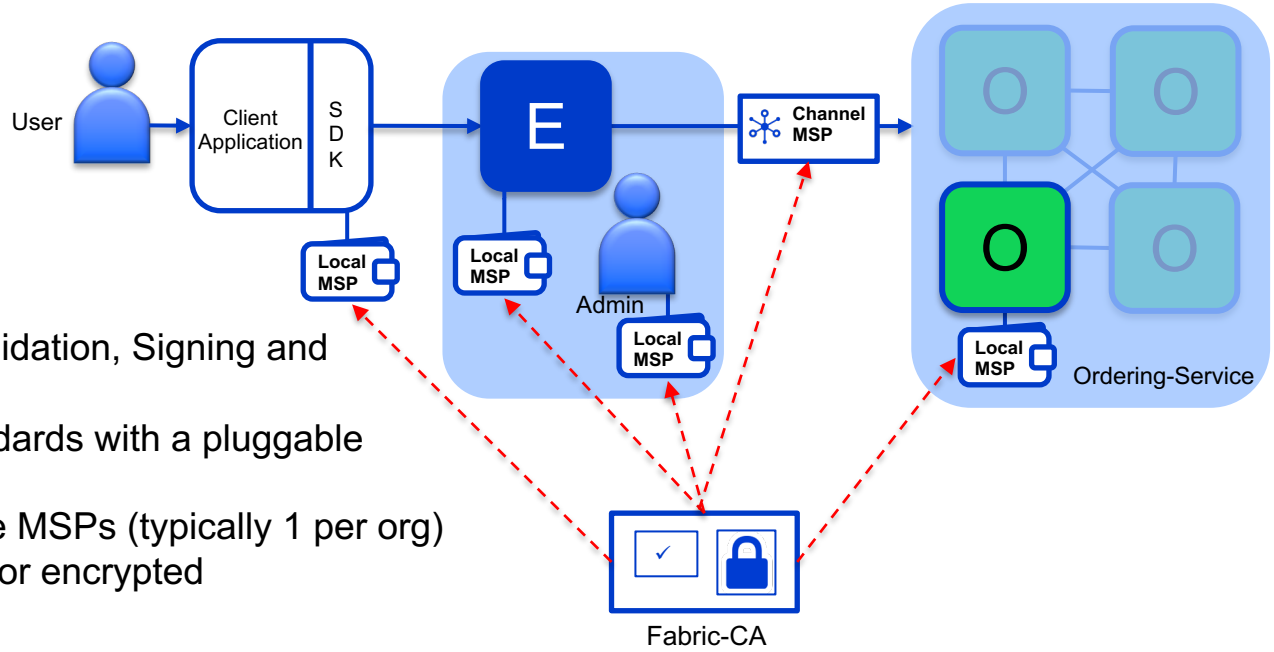
- Orgs 1 and 3 run peers
- Org 2 provides the ordering service only



Membership Services Provider - Overview

A MSP manages a set of identities within a distributed Fabric network

- Provides identity for:
  - Peers and Orderers
  - Client Applications
  - Administrators
- Identities can be issued by:
  - Fabric-CA
  - An external CA
- Provides: Authentication, Validation, Signing and Issuance
- Supports different crypto standards with a pluggable interface
- A network can include multiple MSPs (typically 1 per org)
- Includes TLS crypto material for encrypted communications

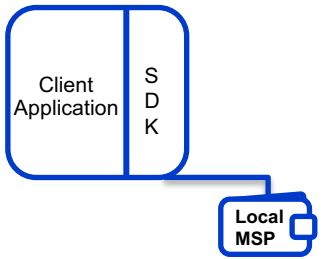


User Identities

Each client application has a local MSP (wallet) to store user identities

- Each local MSP includes:
  - **Keystore**
    - **Private key** for signing transactions
  - **Signcert**
    - **Public x.509 certificate**

- May also include TLS credentials
- Can be backed by a Hardware Security Module (HSM)



user@org1.example.com	
keystore	<private key>
signcert	user@org1.example.com-cert.pem

Admin Identities

Each Administrator has a local MSP (wallet) to store their identity

- Each local MSP includes:
  - **Keystore**
    - **Private key** for signing transactions
  - **Signcert**
    - **Public x.509 certificate**
- May also include TLS credentials
- Can be backed by a Hardware Security Module (HSM)

admin@org1.example.com	
keystore	<private key>
signcert	admin@org1.example.com-cert.pem

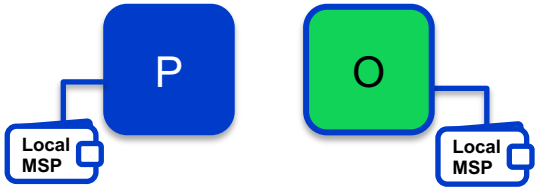
Peer and Orderer Identities

Each peer and orderer has a local MSP

- Each local MSP includes:
  - **keystore**
    - **Private key** for signing transactions
  - **signcert**
    - **Public x.509 certificate**

- In addition Peer/Orderer MSPs identify authorized administrators:
  - **admincerts**
    - List of **administrator certificates**
  - **cacerts**
    - The **CA public cert** for verification
  - **crls**
    - List of **revoked certificates**

- Peers and Orderers also receive channel MSP info
- Can be backed by a Hardware Security Module (HSM) (*)



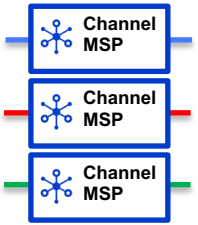
peer@org1.example.com	
admincerts	admin@org1.example.com-cert.pem
cacerts	ca.org1.example.com-cert.pem
keystore	<private key>
signcert	peer@org1.example.com-cert.pem
crls	<list of revoked admin certificates>

(*) Not all features currently supported by IBM Blockchain Platform

Channel MSP information

Channels include additional organisational MSP information

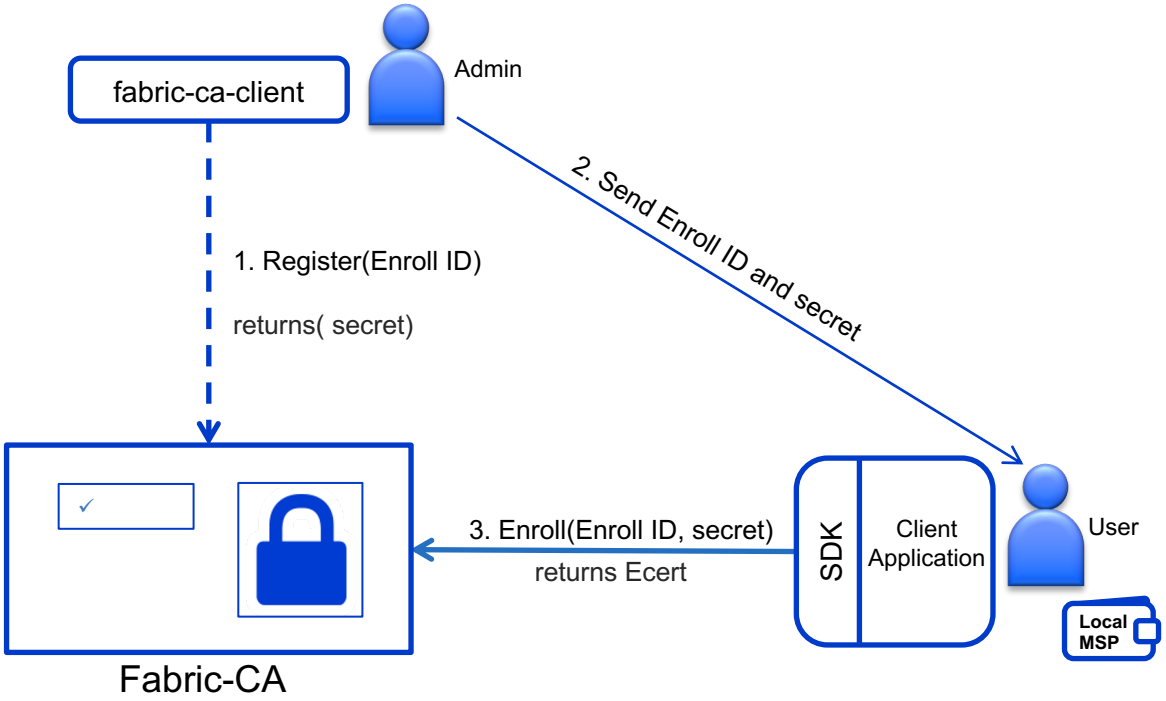
- Determines which orderers or peers can join the channel
- Determines client applications read or write access to the channel
- Stored in configuration blocks in the ledger
- Each channel MSP includes:
  - **admincerts**
    - Any public certificates for administrators
  - **cacerts**
    - The CA public certificate for this MSP
  - **crls**
    - List of revoked certificates
- Does not include any private keys for identity



Channels

ID = MSP1	
admincerts	admin.org1.example.com-cert.pem
cacerts	ca.org1.example.com-cert.pem
crls	<list of revoked admin certificates>

New User Registration and Enrollment



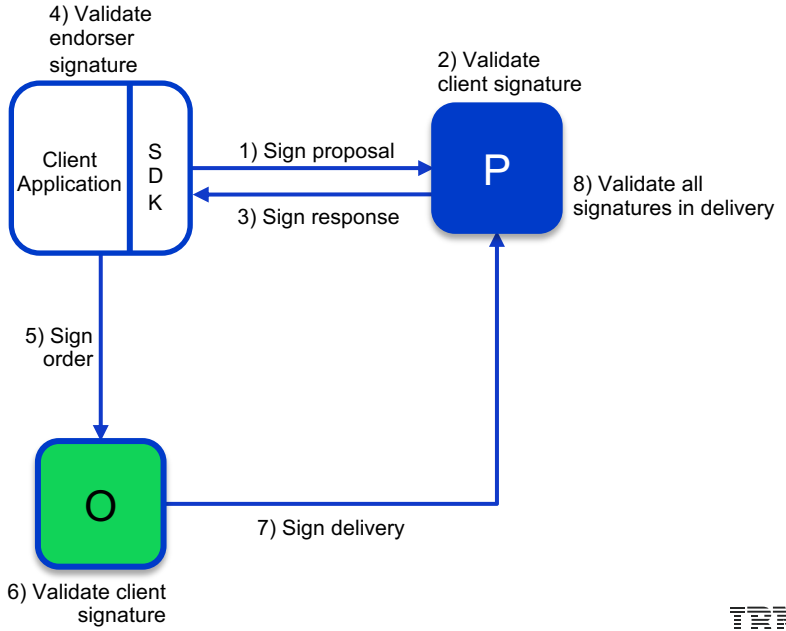
Registration and Enrollment

- Admin registers new user with Enroll ID
- User enrolls and receives credentials
- Additional offline registration and enrollment options available

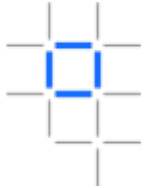
Transaction Signing

All transactions within a Hyperledger Fabric network are signed by permissioned actors, and those signatures validated

- Actors sign transactions with their enrolment private key
  - Stored in their local MSP
- Components validate transactions and certificates
  - Root CA certificates and CRLs stored in local MSP
  - Root CA certificates and CRLs stored in Org MSP in channel

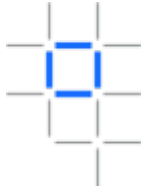


Summary and Next Steps



- Apply shared ledgers and smart contracts to your Business Network
- Think about your participants, assets and business processes
- Spend time thinking about realistic business use cases
- Get some hands-on experience with the technology
- Start with a First Project
- IBM can help with your journey

Further Hyperledger Fabric Information



- Project Home: https://www.hyperledger.org/projects/fabric
- GitHub Repo: https://github.com/hyperledger/fabric
- Latest Docs: https://hyperledger-fabric.readthedocs.io/en/latest/
- Community Chat: https://chat.hyperledger.org/channel/fabric
- Project Wiki: https://wiki.hyperledger.org/projects/fabric
- Design Docs: https://wiki.hyperledger.org/community/fabric-design-docs

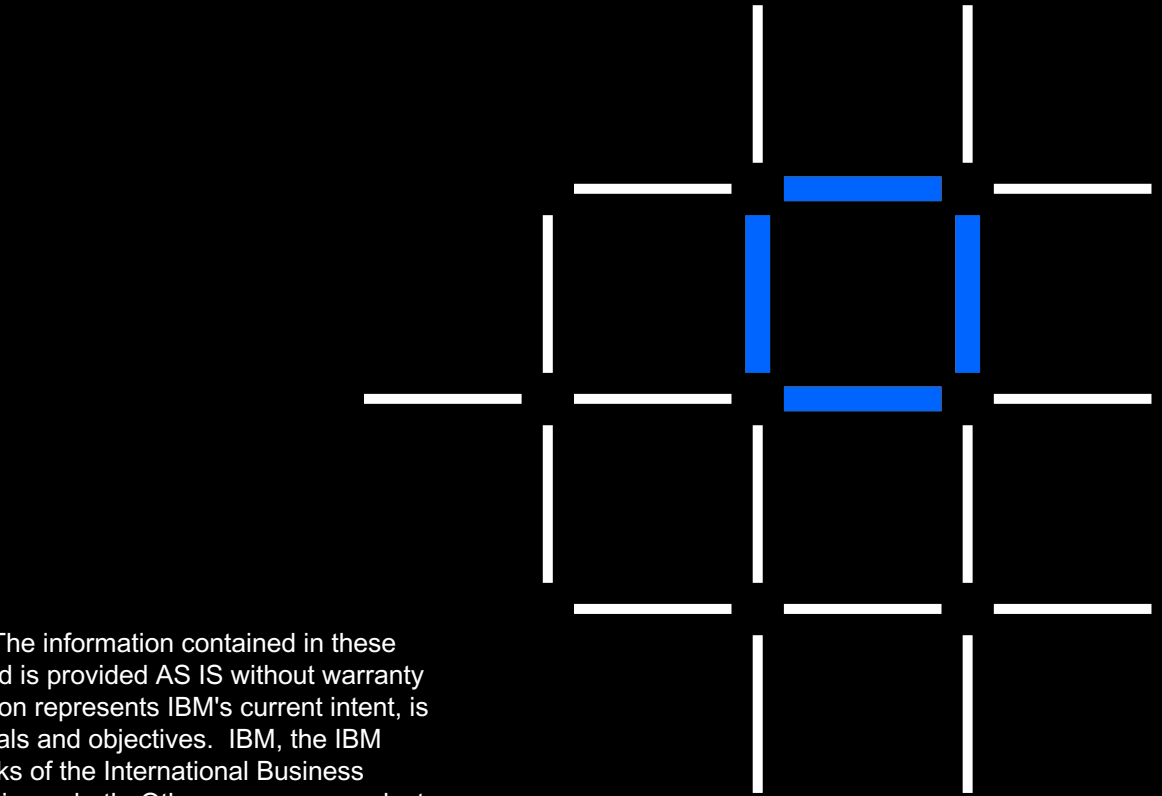
Thank you

IBM **Blockchain**

www.ibm.com/blockchain

developer.ibm.com/blockchain

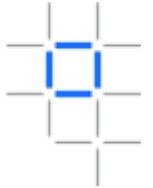
www.hyperledger.org



IBM

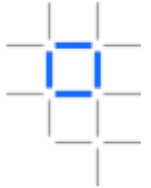
Appendix A: **Hyperledger Fabric Commands**

Fabric Commands



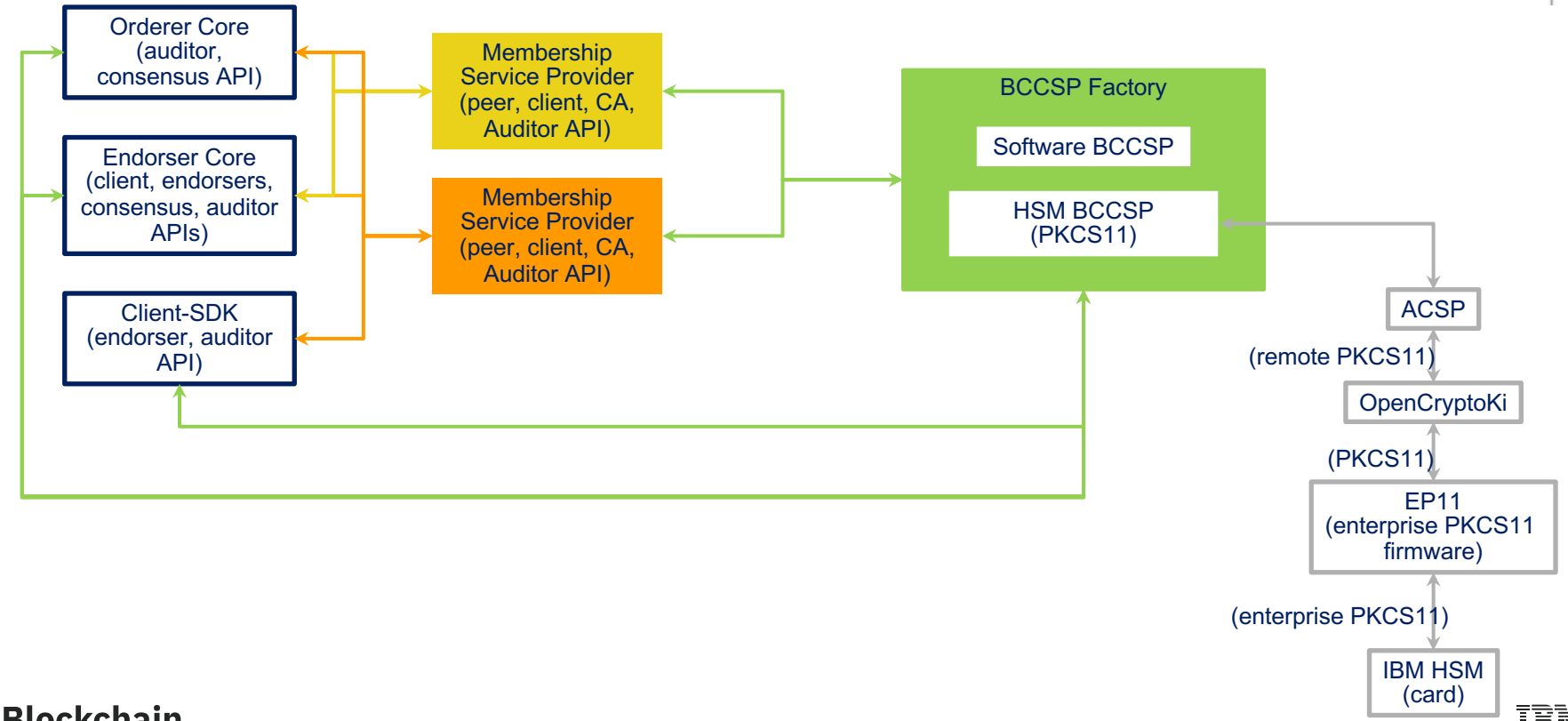
- Fabric has the following commands:
  - **peer …** (For operating and configuring a peer)
    - **peer chaincode …** (Manages chaincode on the peer)
    - **peer channel …** (Manages channels on the peer)
    - **peer node …** (Manages the peer)
    - **peer version** (Returns the peer version)
  - **cryptogen …** (Utility for generating crypto material)
  - **configtxgen …** (Creates configuration data such as the genesis block)
  - **configtxlator …**(Utility for generating channel configurations)
  - **fabric-ca-client …** (Manages identities)
  - **fabric-ca-server …** (Manages the Fabric-CA server)

Configuration Detail

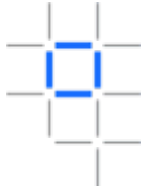


Path	MSP ID	Attributes	Attributes	
config -> channel_group -> groups -> application -> groups	Org1MSP	mod_policy		Admins
		policies -> Admins	mod_policy	Admins
			policy -> value -> identities	Org1MSP, Admin
			Policy -> value -> rule	n_out_of
		Policies -> Readers	Mod_policy	Admins
			Policy -> value -> identities	Org1MSP
			Policy -> value -> rule	N_out_of
		Policies -> Writers	Mod_policy	Admins
			Policy -> value -> identities	Org1MSP
			Policy -> value -> rule	N_out_of

MSP and BCCSP (Modularity and Decentralisation)



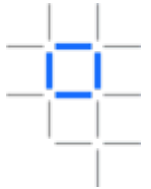
Blockchain Crypto Service Provider (BCCSP)



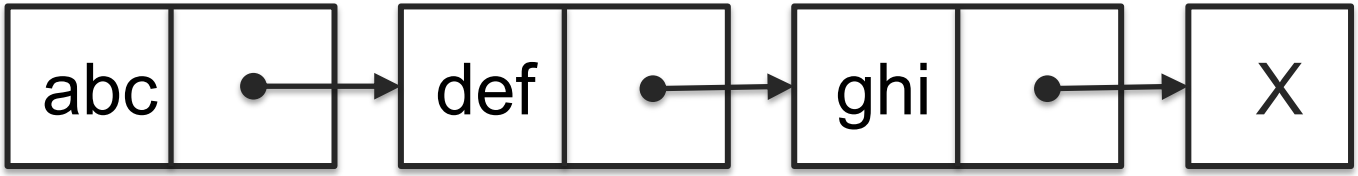
- Pluggable implementation of cryptographic standards and algorithms.
- Pluggability
  - alternate implementations of crypto interface can be used within the Hyperledger Fabric code, without modifying the core
- Support for Multiple CSPs
  - Easy addition of more types of CSPs, e.g., of different HSM types
  - Enable the use of different CSP on different system components transparently
- International Standards Support
  - E.g., via a new/separate CSP
  - Interoperability among standards is not necessarily guaranteed

Appendix B:
Blockchain Data Structures Overview

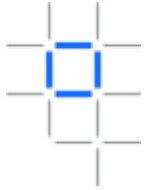
The Linked List



- Linear collection of data elements
- Each element is linked to the next
- Concept dates from 1955



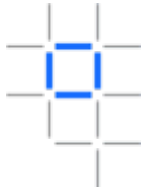
One-Way Hash Functions



- Any function that can be applied to a set of data that is guaranteed to produce the same output for the same input
- One-way means that you can't derive the input from the output
- Often, outputs are *unlikely* to repeat for different inputs
- Forms the basis of much cryptography

h(abc) = 7859
h(def) = 8693
h'(7859) = ?
h(abc) = 7859

The Hash Chain



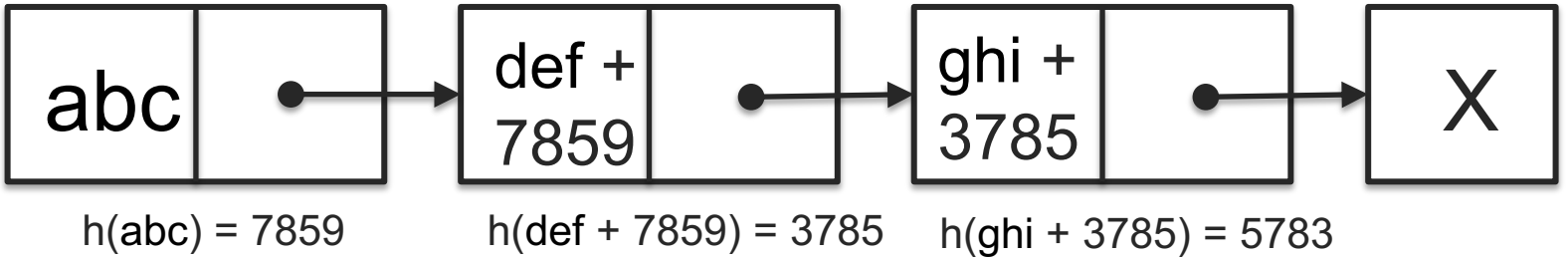
Hash chain: A successive application of a hash function

$$h(h(h(abc))) = 1859$$

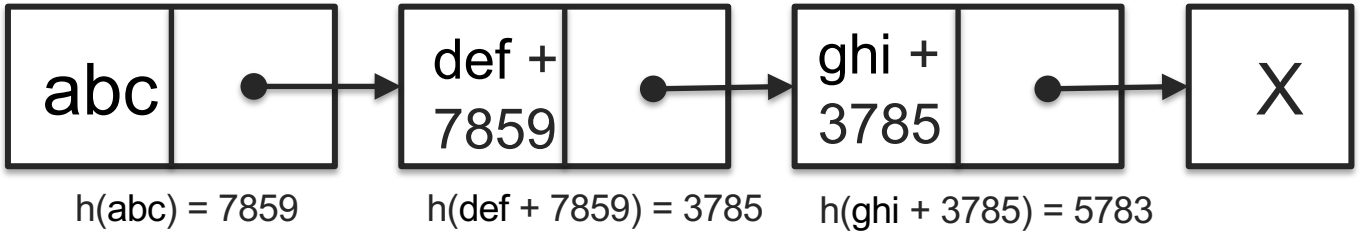
Can combine new data with each successive hash:

$$h(ghi+h(def+h(abc))) = 5783$$

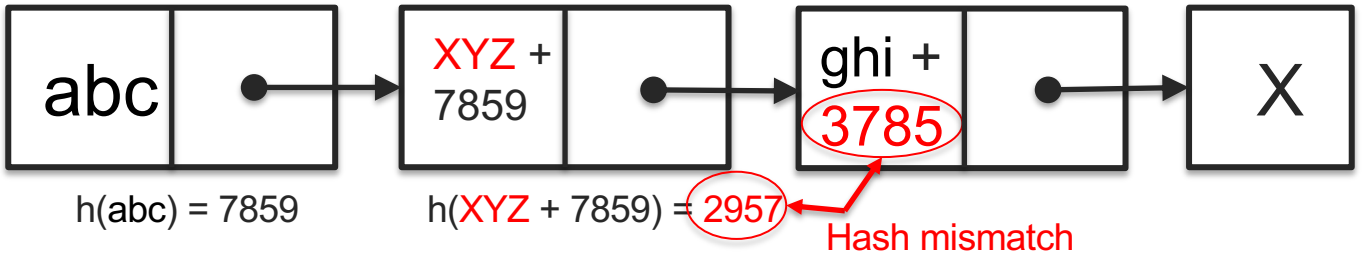
Using this concept you can produce a tamper resistant linked list:



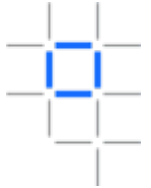
How is This Tamper Resistant?



- Any modification to a data element means that the hashes will not match up
  - You would need to recreate the downstream chain



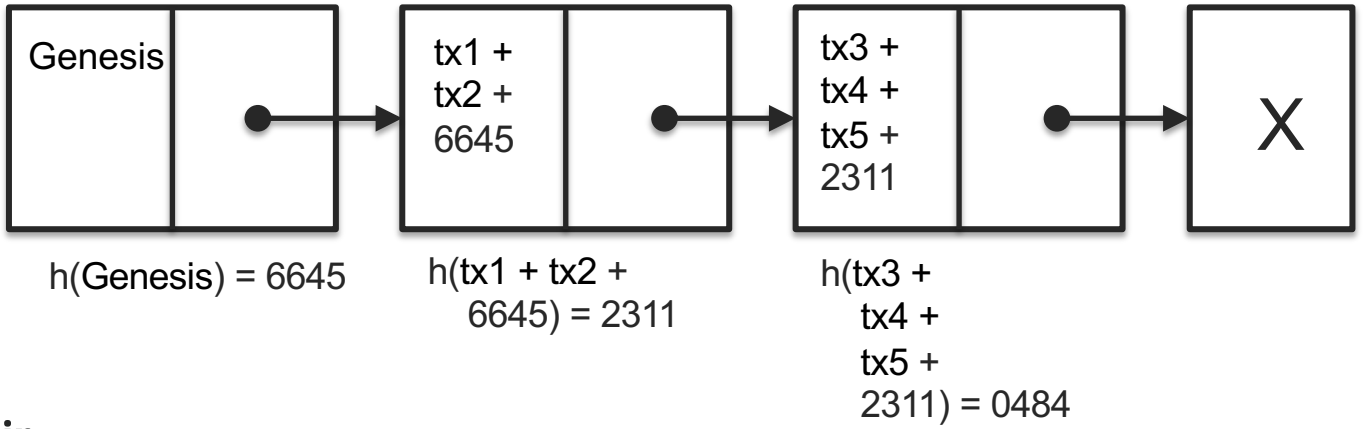
Applied to Blockchain



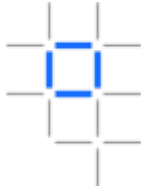
- A blockchain is a hash chain (*with optimizations* that we'll cover shortly)

- Each element (block) in the linked list is a set of zero or more transactions
  - Transactions are an implementation-dependent data object

- First block known as a genesis block
  - May contain some identifying string or other configuration metadata

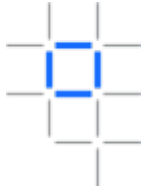


Some Problems with This Approach

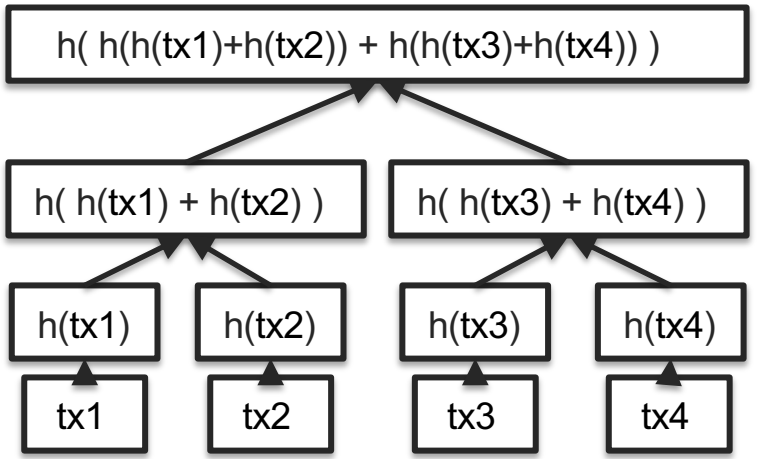


- In the event of tampering, it can be difficult to identify which transaction was modified (particularly when there are many transactions in a block)
    - It is not feasible to have one transaction per block
- It requires all transaction data in order to retain integrity of chain
- Searching transactions is linear (time consuming)

Merkle Tree

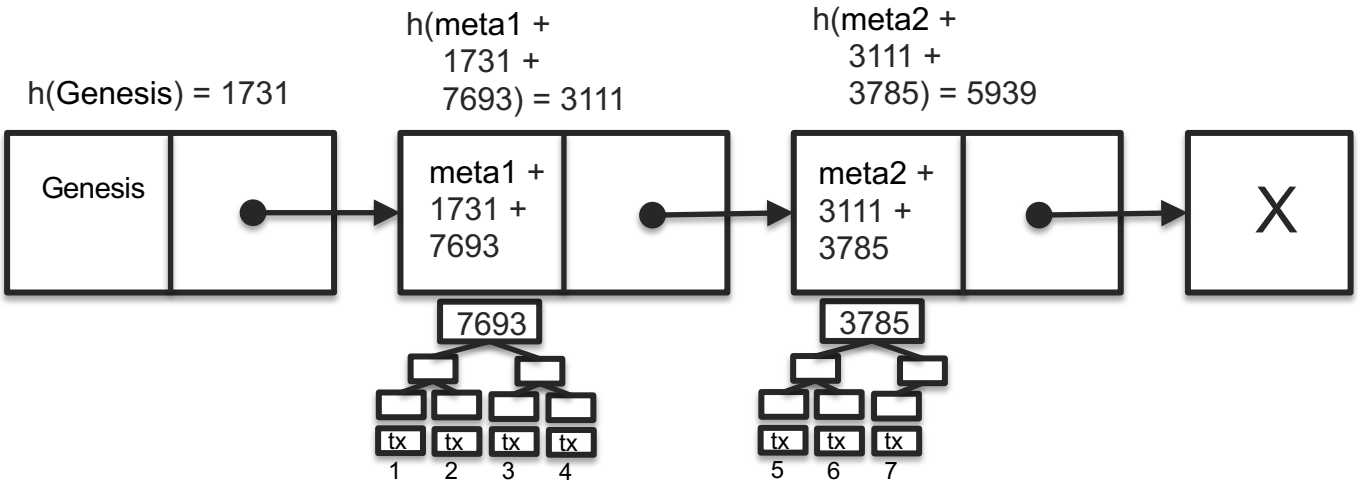


- It is possible to optimise the chain data structure if we arrange it as a tree
  - Makes it easier to identify tampering without sacrificing stability
  - Makes it quicker to traverse
- However, this makes it impossible to add new transactions without re-hashing root nodes

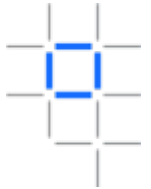


Combining Tree and Chain

- Each element in the chain contains:
  - A pointer (“Merkle root”) to the tree of transactions
  - Other metadata (e.g. timestamp)
  - A hash of the previous block’s data (i.e. Merkle root, metadata and hash)



Benefits of This Approach

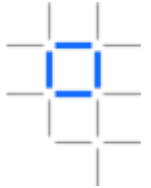


- It allows tampered transactions to be identified easily

- More efficient search within a block
  - O(log N) rather than O(N)

- Allows transaction detail to be stubbed
  - Bitcoin has a Simplified Payment Verifier (SPV) concept: a type of user that doesn't have the entire tree available, just the Merkel roots
  - Note it is also possible to checkpoint and archive old blocks, creating a new Genesis block mid-way through the chain

Common Transactions

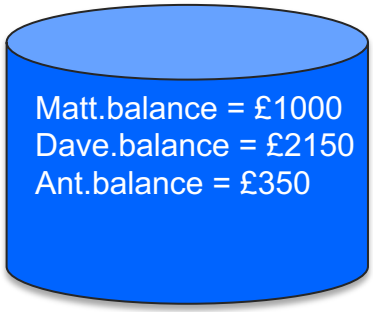


- What's Dave's balance?
- Does Matt have funds to clear a £1000 transaction? (Assuming no overdraft)

#	Transaction	Initiator	Receiver	Amount
1	Create a/c	Cash	Matt	£1000
2	Create a/c	Cash	Dave	£2000
3	Transfer	Matt	Dave	£100
4	Create a/c	Cash	Ant	£500
5	Transfer	Ant	Matt	£50
6	Transfer	Ant	Dave	£200
7	Transfer	Dave	Matt	£100
8	Transfer	Dave	Ant	£50
9	Transfer	Matt	Ant	£50

World State

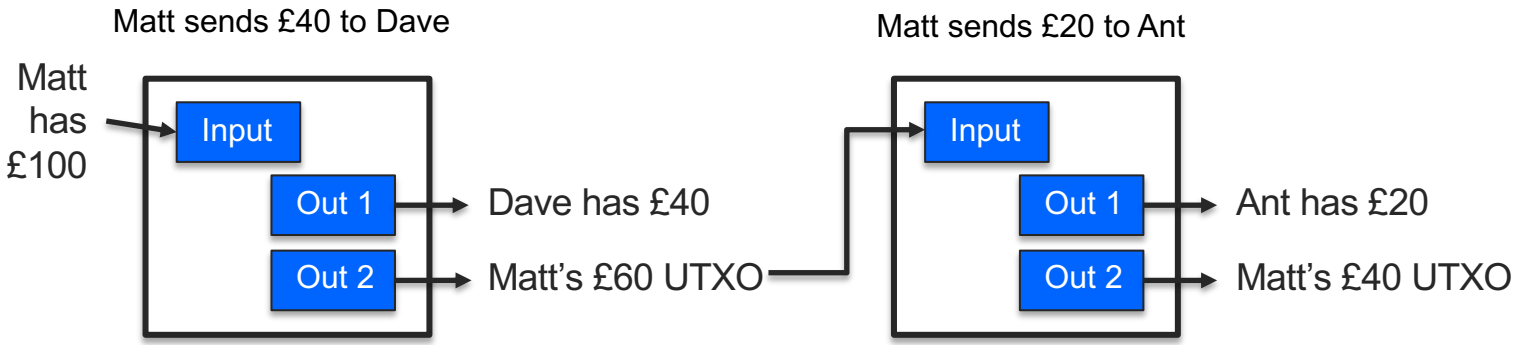
- It is clearly not feasible to reparse the entire transaction log to complete a new transaction
- Blockchains often include an associated database (world state) – e.g. Hyperledger Fabric
- Transactions become a set of creates, reads, updates and deletes of records in this data store



#	Transaction	Initiator	Receiver	Amount
1	Create a/c	Cash	Matt	£1000
2	Create a/c	Cash	Dave	£2000
3	Transfer	Matt	Dave	£100
4	Create a/c	Cash	Ant	£500
5	Transfer	Ant	Matt	£50
6	Transfer	Ant	Dave	£200
7	Transfer	Dave	Matt	£100
8	Transfer	Dave	Ant	£50
9	Transfer	Matt	Ant	£50

Unspent Transaction Outputs

- Some blockchains (e.g. Bitcoin) don't maintain balances
  - Transactions are linked to earlier transactions using an ID (TXID)
  - Outputs always equal inputs
  - Unspent funds are marked as an "Unspent Transaction Output" (UTXO)
  - Only UTXOs can be used as inputs (to prevent double spending)
  - Your "balance" is the aggregation of all of your UTXOs
- In Bitcoin, if your application doesn't specify the UTXO output then the miner gets the excess!




IBM