BLOCKCHAIN: SMART CONTRACTS LECTURE 11 – **ADVANCED TOPICS**

FLORIN CRACIUN

IMPORTANT

Some of the following slides are the property of

Dr. Emanuel Onica & Dr. Andrei Arusoaie

Faculty of Computer Science,

Alexandru Ioan Cuza University of Iași

and are used with their consent.

CONTENTS

ETHEREUM NAME SERVICE (ENS)

- ENS = think on a DNS for Ethereum
- Example:
  - Ethereum foundation donation address: 0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359
  - Same address in an ENS enabled wallet: ethereum.eth
- ENS launched as public registrar on May 4, 2017
- Three EIPs defining the operation:
  - EIP-137 – the basic functionality
  - EIP-162 – an auction system for „domains” in the .eth root
  - EIP-181 – reverse resolution of addresses

ETHEREUM NAME SERVICE (ENS)

ERC 137 – base registry layer

- **Contract with simple functionality (<50 LoC)**
- **Registry keeps domain names in the form of *nodes* (numerical bytes32 values)**
- **Domain names = dot separated labels**
- **Conversion from domain names to *nodes* done via a recursive algorithm – *namehash***
  - **Example for *catchyname.eth*:**

```
node = '\0' * 32
node = sha3(node + sha3('eth'))
node = sha3(node + sha3('catchyname'))
```

ETHEREUM NAME SERVICE (ENS)

- ***Registry***:
  - keeps mapping from a registered name to a *resolver*
  - permits a name owner to set the *resolver* for the name
  - permits a name owner to create subdomains for a name

- ***Resolvers:***
  - represented by addresses associated to nodes
  - perform resource lookups for a name – can provide an associated contract, content, IP addresses
  - their base specification is defined in EIP 137, but can be extended to provide also other types of records

- ***Registrars:***
  - the owners of name nodes
  - responsible for for allocating domain names and updating the ENS registry (e.g., changing resolvers)
  - can be represented by contracts or EOAs

ETHEREUM NAME SERVICE (ENS)

Resolving a name of a contract – two step process

Step I: getting the associated resolver address

- ***function resolver(bytes32 node) constant returns (address);***
- **method provided in registry contract**
- **should receive as parameter the namehash of the name**

Step II: getting the address associated to the name

- ***function addr(bytes32 node) constant returns (address);***
- **method that should be supported by the resolver implementation**
- **should receive as parameter the namehash of the name**
- **resolver must return 0 if no address corresponds to the name**
- **resolver must emit an event if the address is changed**

ETHEREUM NAME SERVICE (ENS)

Other methods, besides *resolver*, in the registry contract

Callable by anybody:

- ***function owner(bytes32 node) constant returns (address);***

(returns owner/registrar of a node)

- ***function ttl(bytes32 node) constant returns (uint64);***

(returns a caching time duration for a node mapping)

Callable only by the node owner:

- ***function setOwner(bytes32 node, address owner);***

(transfers ownership of a node)

- ***function setSubnodeOwner(bytes32 node, bytes32 label, address owner);***

(creates a new subdomain and sets its owner)

- ***function setResolver(bytes32 node, address resolver);***

(sets the resolver of a node)

- ***function setTTL(bytes32 node, uint64 ttl);***

(sets the TTL for a node)

ETHEREUM NAME SERVICE (ENS)

- Resolvers can implement multiple record types (most generic – returning a contract address via *addr* method)

- Must implement the function:

function supportsInterface(bytes4 interfaceID) constant returns (bool)

- *interfaceID* is computed as XOR of hashes of functions provided by the resolver implementation (or the hash if only one function is provided)

- Should return true if the the resolver implements the „interface" defined by the XOR-ed summary

ETHEREUM NAME SERVICE (ENS)

How to get ownership on a domain?

- Root node is controled by 7 signatures, 4 being required for any change
- Currently, the only usable top level domain: .eth
- .eth subdomains are distributed to new owners via a complex Vickrey-type auction system

  Generic Vickrey auctions (ENS system has some changes):

  - All bids are sealed and revealed at the end
  - Highest bidder wins
  - Pays the amount of second-highest bid

ETHEREUM NAME SERVICE (ENS)

Ethereum ENS bidding changes:

- Bidders must lock up a value at least equal with their bid as payment guarantee
- To hide the bids value and the name they bid on, a two-step commit-reveal mechanism is used
- In commit phase a typical hash(name|value|salt) is submitted by each bidder
- There's no central authority for the reveal phase to simultaneously „open" the bids
- Bidders must execute the reveal phase themselves – if they don't, they lose their locked funds (Why?)

ETHEREUM NAME SERVICE (ENS)

Ethereum ENS bidding details:

- If no penalty is applied on locked funds for unrevealed bids, and funds are simply returned at the end of the auction then somebody could abuse/cheat the system by registering multiple bids and revealing only a convenient one.
- An auction is started with first announcing the intention of registering a name – by using its hash; this triggers the start of an auction *bid time*
- Privacy of the wanted name can be, therefore, theoretically maintained being hidden by the hash
- Many auctions may have only one bidder – who intended registering the name and first started the auction
- Cases of multiple bidders:
  - The name proposer makes public that an auction is ongoing for the name
  - Some users just happen to want the same name at the same period of time

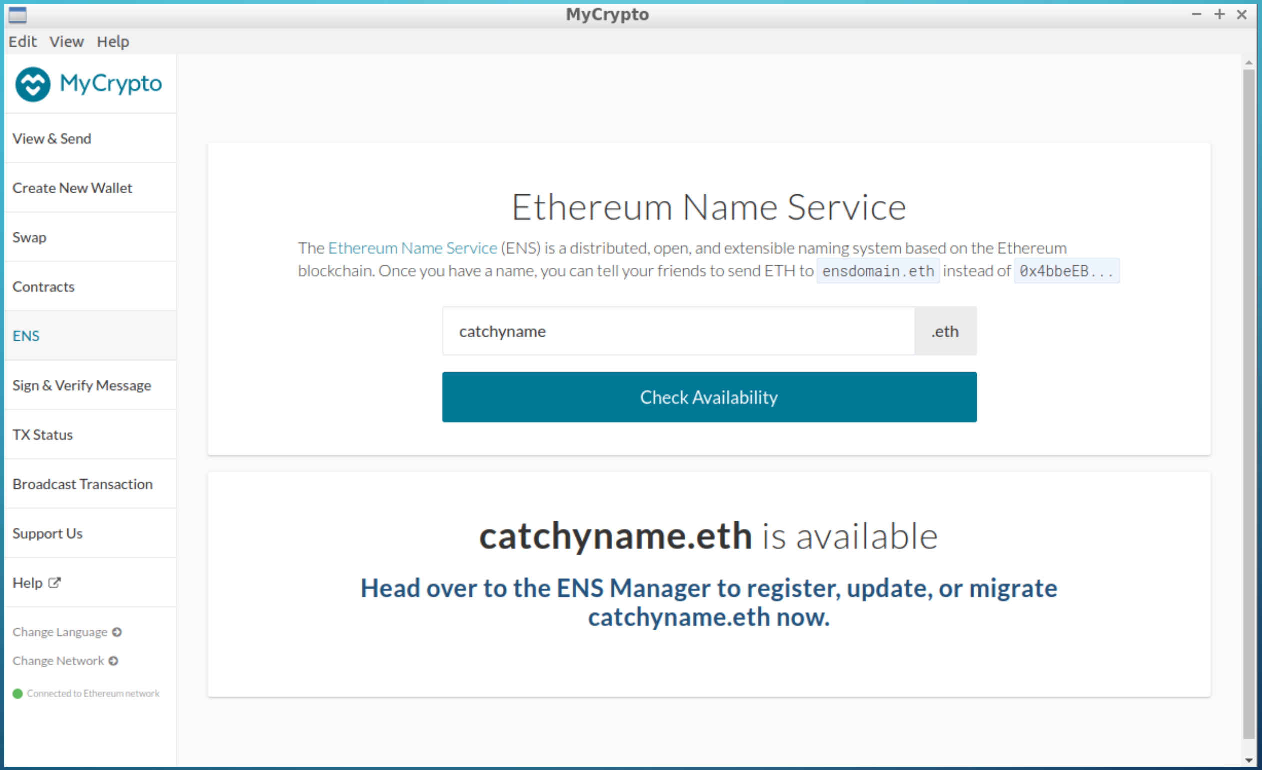
ETHEREUM NAME SERVICE (ENS)

Ethereum ENS bidding details:

- After the *bid time* expires a *reveal time* starts for revealing the bids
- Each independent bid reveal triggers a re-computation of the potential winner
- Winner is set to the highest revealed bid before expiration of *reveal time*
- Winner can recover the difference between his bid and 2nd highest revealed bid
- Winner payment for the registration is locked in a deed contract for the name for the desired period of holding the name (minimum one year)
- Winner can afterwards release the name back to the system and recover the locked funds

ETHEREUM NAME SERVICE (ENS)

Various interfaces for checking the ENS availability:

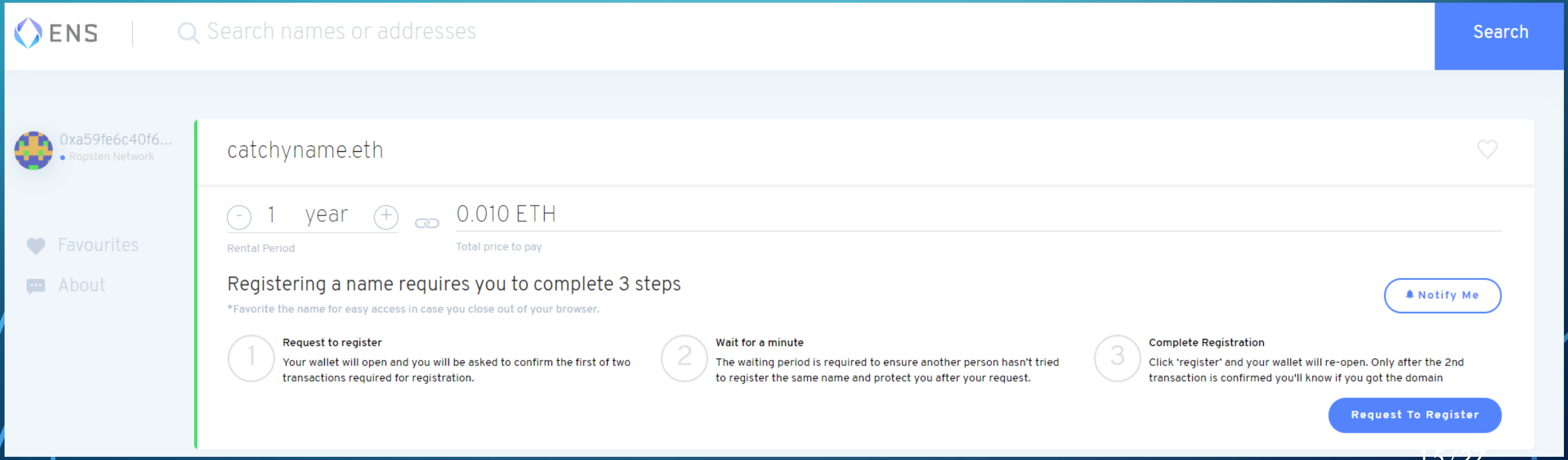


Some wallets also permit starting auctions.

ETHEREUM NAME SERVICE (ENS)

ENS Manager (https://app.ens.domains/)

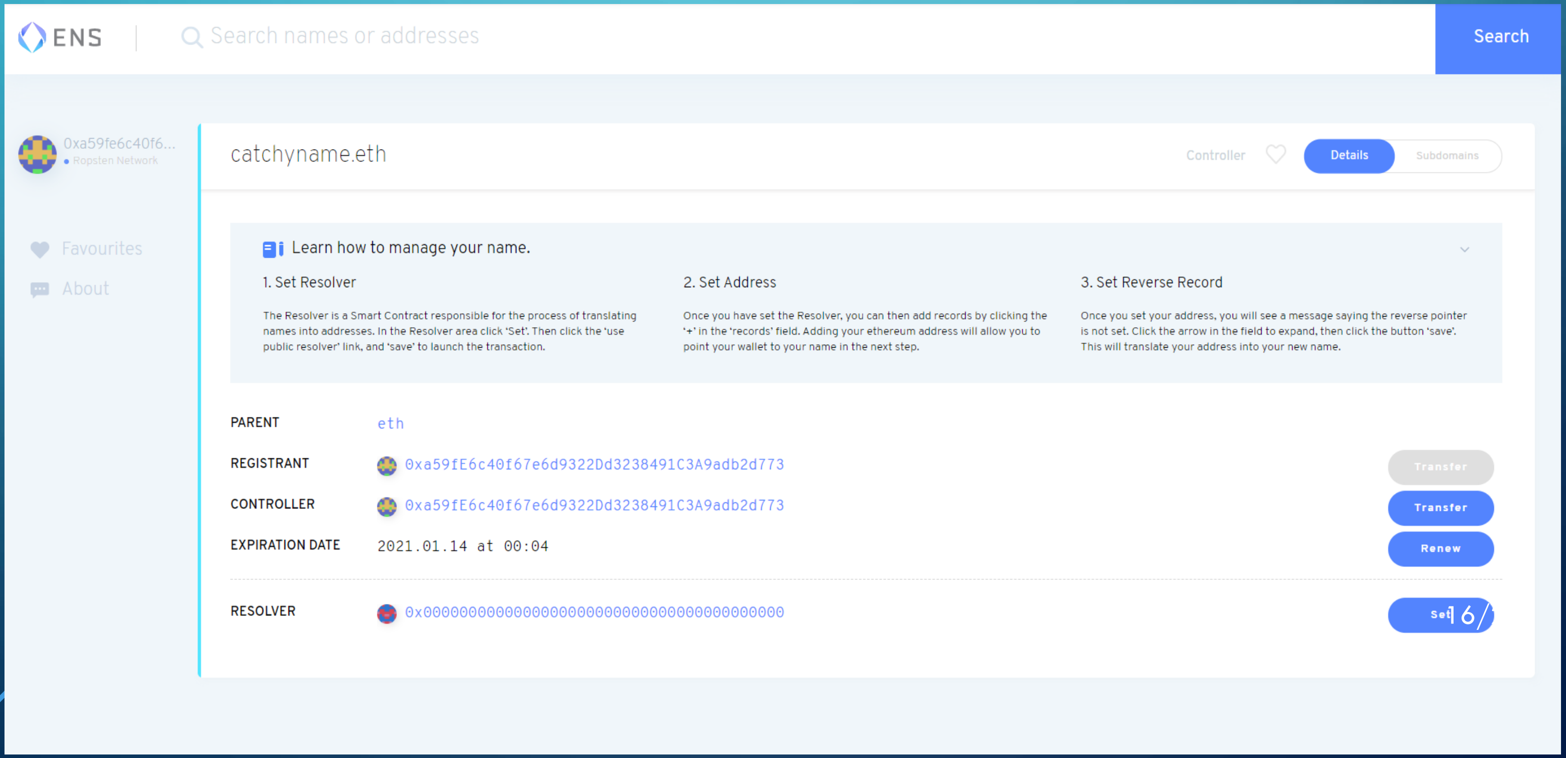
- Most used interface for domain management
- Permits registering, adding resolvers, transferring domain, adding subdomains, etc.



ETHEREUM NAME SERVICE (ENS)

ENS Manager (https://app.ens.domains/)

- Most used interface for domain management
- Permits registering, adding resolvers, transferring domain, adding subdomains, etc.



ETHEREUM SWARM

- P2P storage system integrated with Ethereum
- Developed as part of the the Go-Ethereum (geth) tools suite
- Files replicated by Swarm nodes are referred via hashes
- Useful mostly for storing resources used by Dapps
- Code can also be stored on Swarm

ETHEREUM SWARM

- Comand line tool installed with *geth* and connects to the storage system via *geth*
- After syncing a local geth node and starting a local Swarm node, a web interface can be accessed for querying hosted files:

ETHEREUM SWARM

- A file can be uploaded using the command line tool:

  swarm up /home/myfilename.file

- The –*recursive* option can be used to upload an entire DApp
- The –*defaultpath* option indicates the file (e.g., index.html) to load the DApp

- A Swarm URL is generated after upload (e.g., bzz://cd234c387ac99854e43a284446cd00ab8334afe6331b591229bcd121a33d1244 – basically the resulting hash)

- The Swarm URL can be mapped for easier use via a resolver in ENS - i.e. the resolver to associate the above address to myDapp.eth

- http://swarm-gateways.net/bzz:/<file_hash>/ is a public gateway to access Swarm hosted files

- Storage relies on chunk splitting, cross node replication, uses a Kademlia type DHT and foresees implementation of erasure coding for redundancy (more info at: https://swarm-guide.readthedocs.io/en/latest/architecture.html)

ETHEREUM WHISPER

- Communication protocol for Dapps
- Oriented towards small content messaging
- Potential uses:
  - Signaling between DApps for collaborating on some transaction
  - Small chatroom apps
  - Announcements such as new offers provided by a Dapp
- General characteristics:
  - API usable exclusively by DApps (i.e., not other individual users)
  - Lacking any latency guarantees
  - Intended to defeat tracing packets/traffic analysis attempts
  - First version implemented as part of web3.shh API – second version in development
  - Follows a topic based subcription model
  - Basic usage: https://eth.wiki/en/concepts/whisper/overview

HYPERLEDGER FABRIC

- Enterprise-grade DLT platform
- Developed by Linux Foundation
  - Part of the Hyperledger ecosystem
  - Main contributor – IBM

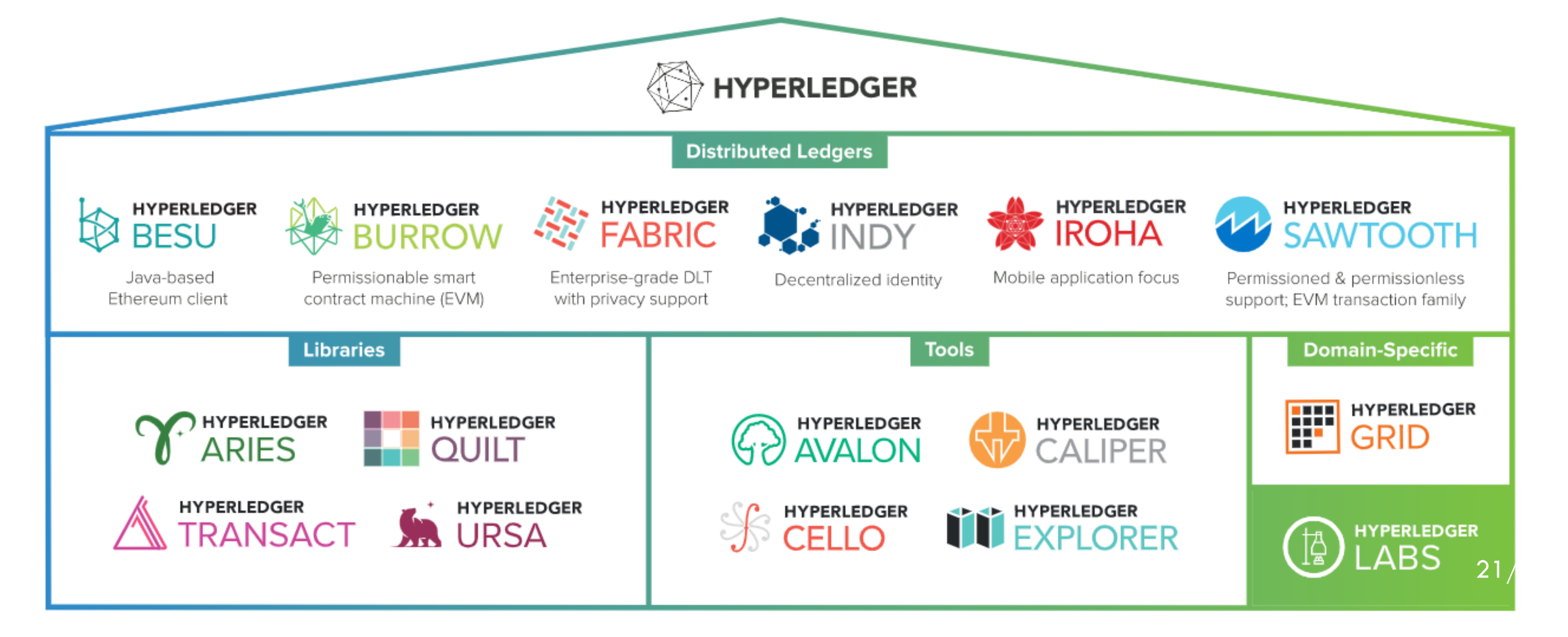


Figure source: https://www.hyperledger.org/

HYPERLEDGER FABRIC

- Permissioned blockchain model:
  - Knowledge of participants
  - Typically smaller numbers of peers (enterprise use oriented)
  - Membership service offering some trust degree

- Modular architecture:
  - Support for pluggable consensus protocols
  - CFT and BFT implementations available
  - Does not require a native cryptocurrency

- Reference paper: *Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains* (E. Androulaki et al. – EuroSys 2018)

HYPERLEDGER FABRIC

- Smart contracts – „chaincodes" – support for general purpose languages (Go, Node.js, Java)

- Typical approach:
  1) Validate & order transactions
  2) Execute transactions

- Fabric approach:
  1) Execute and *endorse* a transaction (check correctness)
  2) Order transactions using a consensus protocol
  3) Validate transactions against application specific endorsement policy

HYPERLEDGER FABRIC

The execution phase:

- **Each chaincode has an endorsement policy associated, i.e., 3 out of 5 *endorser* peers in a defined set should endorse a transaction**
- **Client sends transaction proposal to a set of endorser peers**
- **Endorsers *simulate* the transaction proposal by executing the operations in the associated chaincode:**
  - **done in an isolated Docker container**
  - **modifies chaincode associated state maintained as key-value store**
  - **result:**
    - ***writeset* - modified keys and values**
    - ***readset* - keys required by execution and their version**
  - **result not persistent on ledger state**
  - **result returned to client**
- **Requirement: results must be identical (according to policy specs) to proceed further – conflicts might arise due to some endorser being behind latest chaincode state**

HYPERLEDGER FABRIC

The ordering phase:

- **Client submits a consistent endorsed transaction (the signed obtained result + the endorsers set) to an *ordering service* formed of multiple orderer peers (via a network peer)**

- **The ordering service does not execute or validate transactions**

- **The ordering service adds received transactions to new blocks until:**
  - **A set block size is reached or…**
  - **A set timer expires**

- **Each new formed block is proposed for consensus in the ordering service, and afterwards broadcasted to peers**

HYPERLEDGER FABRIC

The validation phase:

- **New formed blocks reach all peers being transmitted directly from the ordering service or via gossip from other peers**

- **Each peer validates each new received block:**
  - **Checks if each transaction in the block respects the endorsement policy for the chaincode (executed in parallel for the included transactions)**
  - **A read-write conflict is evaluated by checking sequentially the transactions in the block for the readset keys version, by comparing with the state of the ledger – if conflicts appear the transactions are marked as invalid**

- **Finally the peer updates the ledger with the validated block**

HYPERLEDGER FABRIC

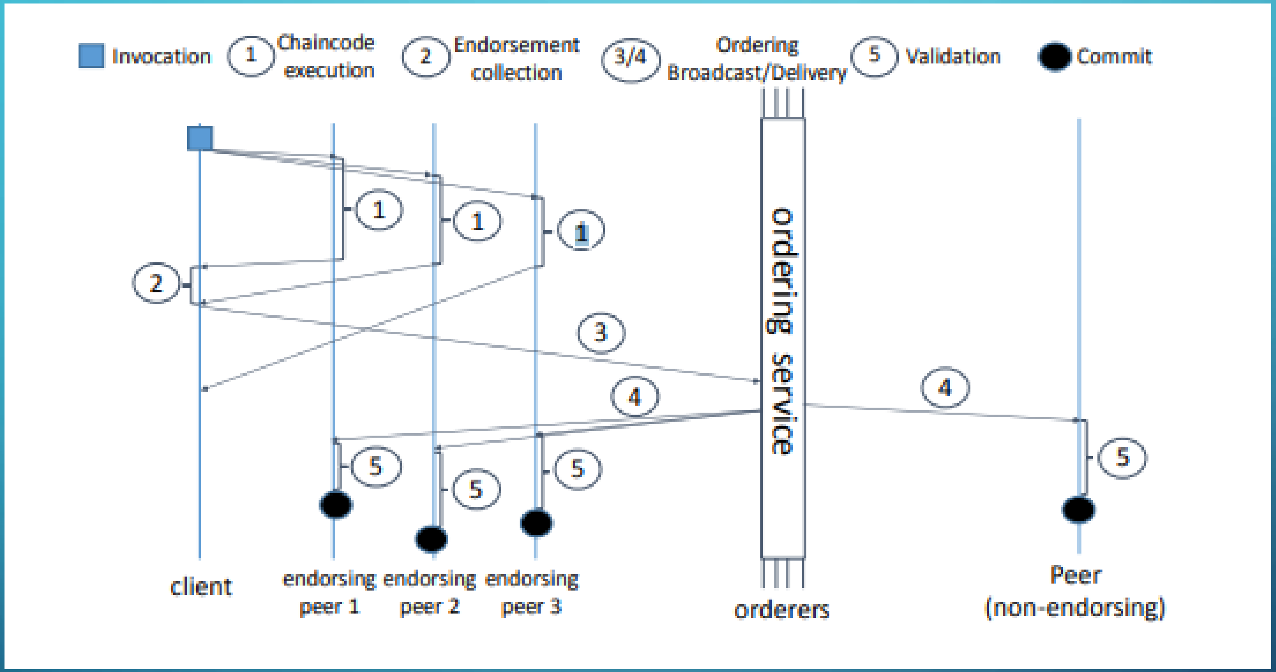


Figure source: *Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains* (E. Androulaki et al. – EuroSys 2018)