A DISTRIBUTED OPERATING SYSTEM FOR PERMISSIONED BLOCKCHAINS

HYPERLEDGER FABRIC

BYZANTINE GENERALS' PROBLEM

- An agreement problem
- A group of generals wants to attack a city
- They must to know when to attack or when to retreat
- They cannot see or hear each other
- What do they need? –> A PLAN
- We should build a network (nodes, communication channels, protocols)

BYZANTINE GENERALS' PROBLEM

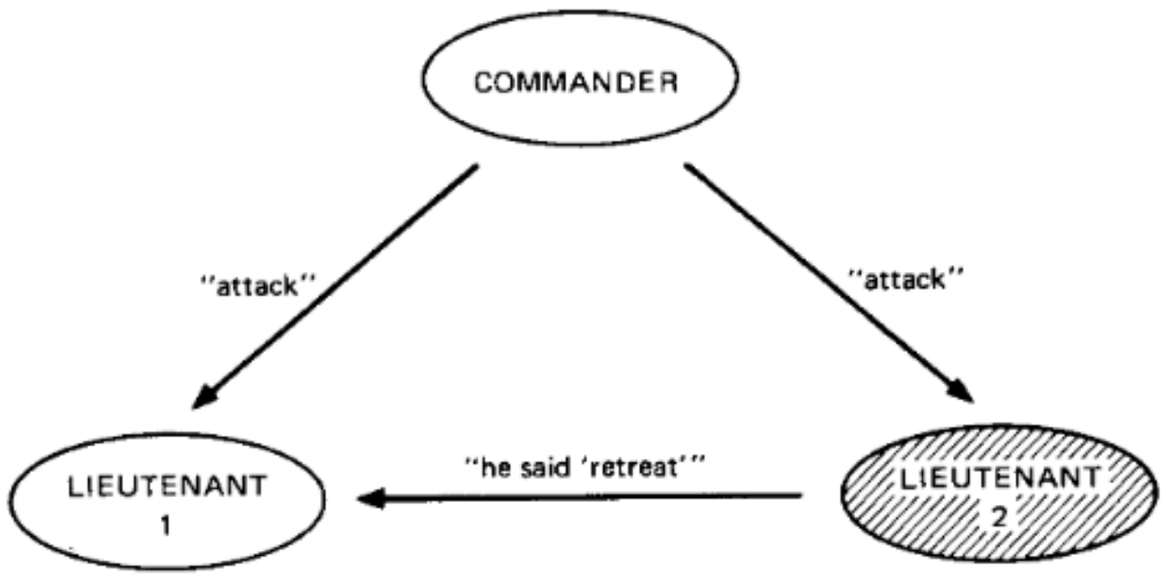


Fig. 1. Lieutenant 2 a traitor.

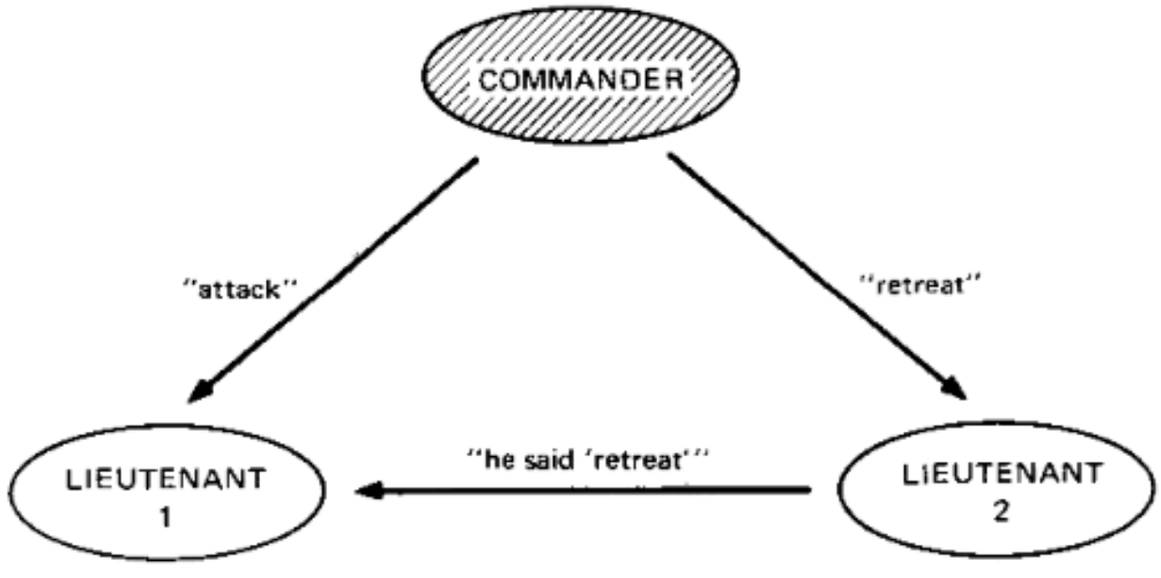


Fig. 2. The commander a traitor.

... take me to your commander

BYZANTINE GENERALS' PROBLEM IN PRACTICE

- Aircraft systems (Boeing 777 and 787 flight control systems)
- Spacecrafts (SpaceX Dragon flight system)
- Bitcoin

BLOCKCHAINS

- Simply, a blockchain is a huge ledger for recording transactions
- It is maintained within a distributed network of mutually untrusting peers (the generals)
- Every peer maintains a copy of the ledger
- They validate and order the transactions through a consensus protocol
- Blockchains have emerged with Bitcoin

BLOCKCHAINS

- In a public (*permissionless*) blockchain anyone can participate without a specific identity
- *Permissioned* blockchains run a blockchain (of course) among a set of *known*, *identified* participants. This is a way to secure the interactions among a group of entities that have a common goal but which do not fully trust each other
- Blockchains may execute arbitrary, programmable transaction logic in the form of smart contracts (as exemplified by Ethereum)

SMART CONTRACTS

- A smart contract functions as a trusted distributed application and gains its security from the blockchain and the underlying consensus among the peers

- Many existing smart contracts blockchains follow the blueprint of State-Machine Replication and implement so-called active replication: first, the transactions are ordered and propagated to all peers and second, each peer executes the transactions sequentially

SMART CONTRACTS

- Prior permissioned blockchains suffer from many limitations:
  - Smart contract must be written in a *fixed, non-standard, or domain-specific language*
  - The sequential execution of transaction by all peers *limits performance*
  - Transaction must be *deterministic*, which can be difficult to be ensured programmatically
  - Every smart contract runs on *all peers*, which is at odds with *confidentiality*

OK, LET'S SEE FABRIC

- Fabric introduces a new blockchain architecture aiming at *flexibility, scalability and confidentiality*
- Fabric support the execution of distributed applications written in *standard programming languages*
- Fabric is the first *distributed operating system* for permissioned blockchains

FABRIC

- The architecture of Fabric follows the *execute-order-validate* paradigm

FABRIC

- This design departs radically from the *order-execute* paradigm
- It combines the two approaches to replication, *passive* and *active*:
  - Every transaction is executed (endorsed) only by a subset of peers, which allows for parallel execution (passive)
  - The transaction's effects on the ledger state are only written after reaching consensus of a total order among them (active)

FABRIC

- This *hybrid replication design*, which mixes passive and active replication in the Byzantine model, and the *execute-order-validate* paradigm, represent the most innovation in Fabric architecture

ORDER-EXECUTE ARCHITECTURE

- All previous blockchain systems follows *order-execute architecture*
- Let's take Ethereum for example:

    1. every peer assembles a block containing valid transactions
    2. the peer tries to solve the puzzle
    3. if the peer is *lucky* and solves the puzzle it disseminates the block to the network
    4. every peer receiving the block validates the solution *and* all transaction in the block

- Simply said, every peer repeats the execution of the lucky peer from its first step
- If this is not enough, all transactions must be executed sequentially

LIMITATIONS OF ORDER-EXECUTE ARCHITECTURE

- Sequentially execution
  - It limits the effective throughput that can be achieved
  - Since the throughput is inversely proportional to the execution latency, this may become a performance bottleneck
  - A *Denial-of-Service* attack could simply introduce smart contracts that take very long time to execute

LIMITATIONS OF ORDER-EXECUTE ARCHITECTURE

- Non-deterministic code
  - This is usually addressed by programming blockchains in domain-specific languages (e.g. Ethereum Solidity) that are expressive enough for their applications but limited for deterministic execution
- Only one non-deterministic contract created with malicious intent is enough to bring the entire blockchain to a halt

LIMITATIONS OF ORDER-EXECUTE ARCHITECTURE

- Confidentiality of execution
  - Many permissioned systems run all smart contract on all peers
  - However, many intended use cases for permissioned blockchains require confidentiality

ARCHITECTURE

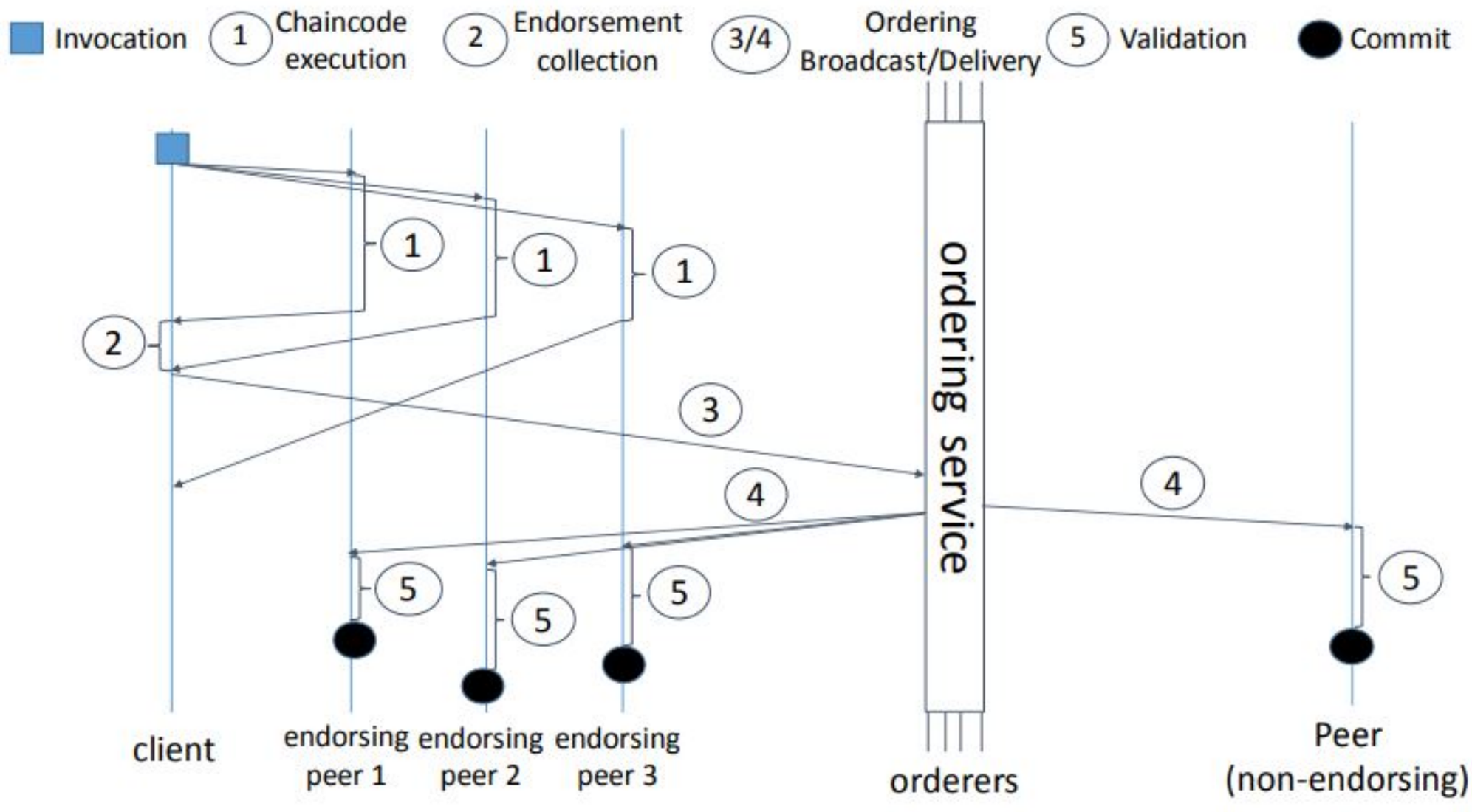
- **Overview**
- Execution Phase
- Ordering Phase
- Validation Phase
- Trust and Fault Model

OVERVIEW

- A distributed application for Fabric consists of two parts:

1. A smart contract, called chaincode, which is program code that implements the application logic and runs during the execution phase.
2. An endorsement policy that is evaluated in the validation phase and is used for validating the transaction. This cannot be modified by untrusted application developers.

FABRIC HIGH LEVEL TRANSACTION FLOW



ARCHITECTURE

- Overview
- **Execution Phase**
- Ordering Phase
- Validation Phase
- Trust and Fault Model

EXECUTION PHASE

- Clients send transaction proposal to endorsers (peers) based on endorsement policy

- Endorsers simulate the proposal by executing the operation on the local blockchain without synchronization

- As a result of the simulation, each endorser produces a value writeset with the state updates and a readset

- Endorser create cryptographic message and send back to client as response

- The client collects endorsements until they satisfy the endorsement policy of the chaincode and then creates a transaction that is sent to ordering service

ARCHITECTURE

- Overview
- Execution Phase
- **Ordering Phase**
- Validation Phase
- Trust and Fault Model

ORDERING PHASE

- The ordering phase establish a total order on all submitted transactions per channel (blockchain)

- Batch transactions in blocks and outputs a hash-chained sequence of blocks containing transactions

- Ordering service ensures that these blocks are ordered and rules are satisfied on the blockchain

ARCHITECTURE

- Overview
- Execution Phase
- Ordering Phase
- **Validation Phase**
- Trust and Fault Model

VALIDATION PHASE

- Blocks are delivered to peers by ordering service or gossip then the the validation begins by following these steps:
- The endorsement policy evaluation occurs in parallel for all transactions within the block.
- A read-write conflict check is done for all transactions in the block sequentially.
- A ledger update is done in which the block is appended and stored in local ledger

ARCHITECTURE

- Overview
- Execution Phase
- Ordering Phase
- Validation Phase
- **Trust and Fault Model**

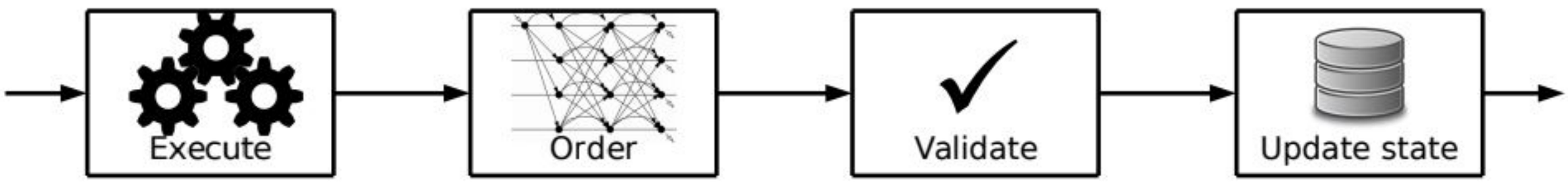
TRUST AND FAULT MODEL

- Fabric can accommodate flexible trust and fault assumptions. In general, any client is considered potentially malicious

- The ordering service considers all peers (and clients) as potentially Byzantine

- A distributed application can define its own trust assumptions through the endorsment policy

EXECUTE-ORDER-VALIDATE ARCHITECTURE OF FABRIC



- Simulate trans. and endorse
- Create rw-set
- Collect endorse-ments

- Order rw-sets
- Atomic broadcast (consensus)
- Stateless ordering service

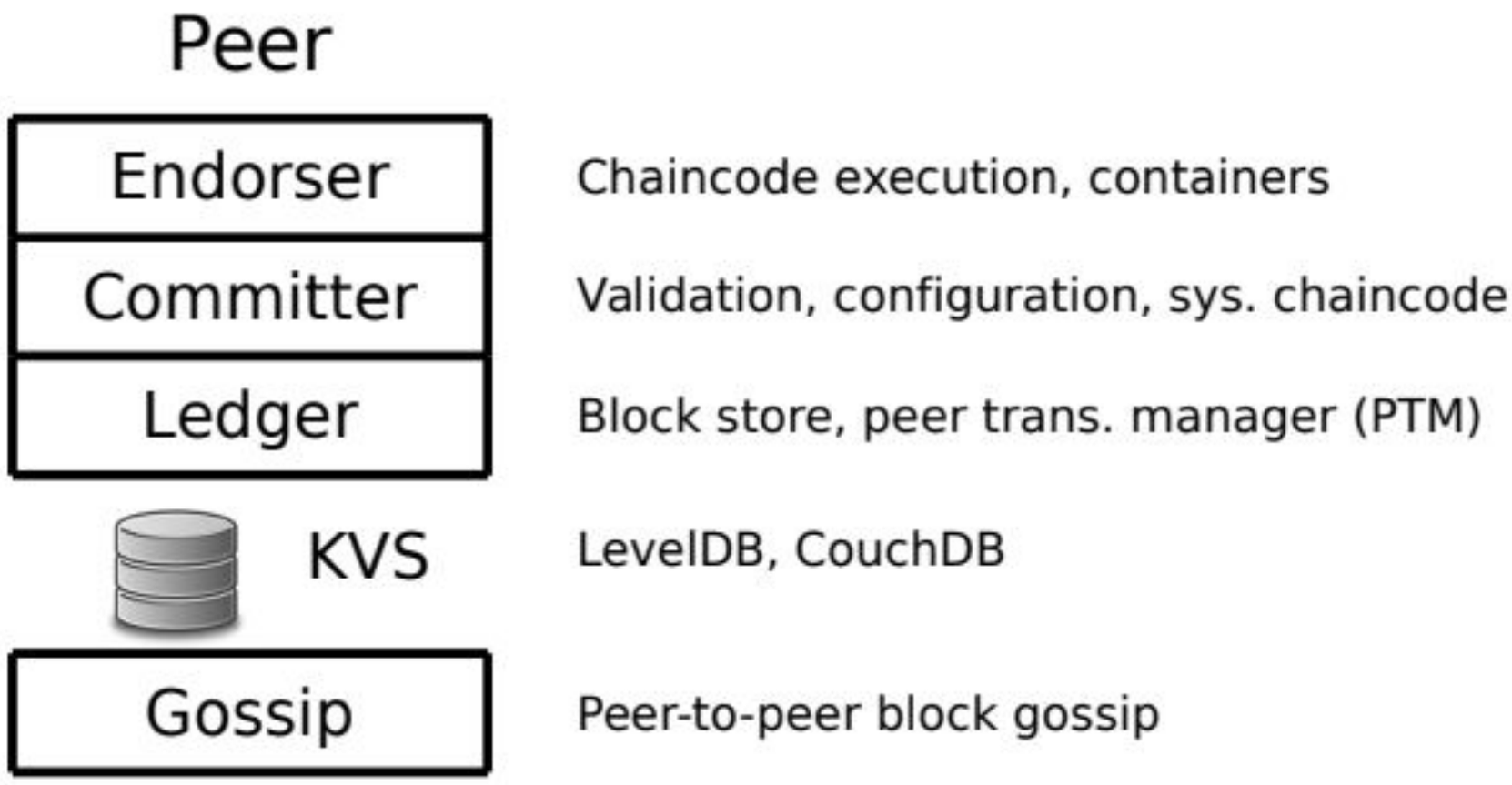
- Validate endorse-ments & rw-sets
- Eliminate invalid and conflicting trans.

- Persist state on all peers

COMPONENTS

- **Membership Service**
- Ordering Service
- Peer Gossip
- Ledger
- Chaincode Execution
- Configuration and System Chaincodes

COMPONENTS OF FABRIC PEER



MEMBERSHIP SERVICE

- Maintains the identities of all the nodes in the system(clients, peers,and OSN)

- Responsible for issuing node credentials that are used for authentication and authorization

- The membership service comprises a component at each node, where it may authenticate transactions, verify the integrity of transactions, sign and validate endorsements, and authenticate other blockchain operations

COMPONENTS

- Membership Service
- **Ordering Service**
- Peer Gossip
- Ledger
- Chaincode Execution
- Configuration and System Chaincodes

ORDERING SERVICE

- The ordering service manages multiple channels(stateless)
- Provides following services:
  - Atomic broadcast for establishing order on transactions, implementing the broadcast and deliver calls
  - Reconfiguration of a channel, when its members modify the channel by broadcasting a configuration update transaction
  - Optionally, access control, in those configurations where the ordering service acts as a trusted entity
- The atomic broadcast is done by Kafka and the OSN are basically proxies between Kafka and peers

COMPONENTS

- Membership Service
- Ordering Service
- **Peer Gossip**
- Ledger
- Chaincode Execution
- Configuration and System Chaincodes

PEER GOSSIP

- The throughput of the ordering service is capped by the network capacity of its nodes
- Maintains an up-to-date membership view of the online peers in the system
- Used to optimally utilize the bandwidth
- Resilient to leader failures

COMPONENTS

- Membership Service
- Ordering Service
- Peer Gossip
- **Ledger**
- Chaincode Execution
- Configuration and System Chaincodes

LEDGER

- Maintains the ledger and the state on persistent storage and enables simulation, validation, and ledger-update phases
- Has 2 components:
  - A ledger block store
  - A peer transaction manager
- The ledger block store persists transaction blocks and is implemented as a set of append-only files
- The peer transaction manager (PTM) maintains the latest state in a versioned key-value store

COMPONENTS

- Membership Service
- Ordering Service
- Peer Gossip
- Ledger
- **Chaincode Execution**
- Configuration and System Chaincodes

COMPONENTS

- Membership Service
- Ordering Service
- Peer Gossip
- Ledger
- Chaincode Execution
- **Configuration and System Chaincodes**

EVALUATION

- Fabric is a complex distributed system
- Its performance depends on many parameters
    - The choice of distributed application
    - Transaction size
    - The ordering service
    - Consensus implementation
    - Topology of nodes in the network
    - Number of nodes
    - The hardware on which nodes runs

FABRIC COIN

- In the absence of a standard benchmark for blockchains, it is used a simple authority-minted cryptocurrency that uses the data model of Bitcoin — Fabcoin

FABCOIN

- The data model introduced by Bitcoin has become known as "unspent transaction output" or UTXO
- UTXO represents each step in the evolution of data object as a separate atomic state on the ledger
- Such a state is created by a transaction and destroyed (or "consumed") by another unique transaction occurring later
- Every given transaction destroys a number of input states and creates one or more output states

FABCOIN

- A "coin" in Bitcoin is initially created by a coinbase transaction that rewards the "miner" of the block
- This appears of the ledger as a coin state designating the miner as the owner
- Any coin can be spent in the sense that the coin is assigned to a new owner by a transaction that atomically destroys the current coin state
- Every state may be seen as a KVS entry with logical version 0 after creation
- When it is destroyed again, it receives version 1
- There should not be any concurrent updates to such entries

FABCOIN IMPLEMENTATION

- Each state in Fabcoin is a tuple of the form (key, val) = (txid.j, (amount, owner, label)), denoting the coin state created as the j-th output of a transaction with identifier txid
- Labels are strings used to identify a given type of a coin ('USD', 'EUR', 'FBC')
- The Fabcoin implementation consists of three parts
  - A client wallet
  - The Fabcoin chaincode
  - A custom VSCC (validation system chaincode) for Fabcoin implementing its endorsement policy

CLIENT WALLET

- By default, each Fabric client maintains a Fabcoin wallet that locally stores a set of cryptographic keys allowing the client to spend coins
- The client wallet signs, with the private keys that correspond to the input coin states
- The client wallet includes the Fabcoin request into a transaction and sends this to a peer of its choice

FABCOIN CHAINCODE

- A peer runs the chaincode of Fabcoin which simulates the transaction and creates readsets and writesets.
- In the case of a transaction, for every input coin state the chaincode first performs *GetState(in)*
- This includes *in* in the readset along with its current version
- Then the chaincode executes *DelState(in)* for every input state *in,* which also adds *in* to the writeset and effectively marks the coin state as "spent"

CUSTOM VSCC

- Every peer validates Fabcoin transaction using a custom VSCC (validation system chaincode)
- This verifies the cryptographic signature(s) in sigs under the respective public key(s) and performs semantic validation
- Note that the Fabcoin VSCC does not check transactions for double spending, as this occurs through Fabric's standard validation that runs after the custom VSCC
- In particular, if two transactions attempt to assign the same unspent coin state to a new owner, both would pass the VSCC logic but would be caught subsequently in the read-write conflict checked performed by the PTM

EXPERIMENT 1

- Choosing the blocking size
- A critical Fabric configuration parameter that impacts both throughput and latency is block size
- We can observe that throughput does not significantly improve beyond a block size of 2MB, but latency gets worse (as expected)

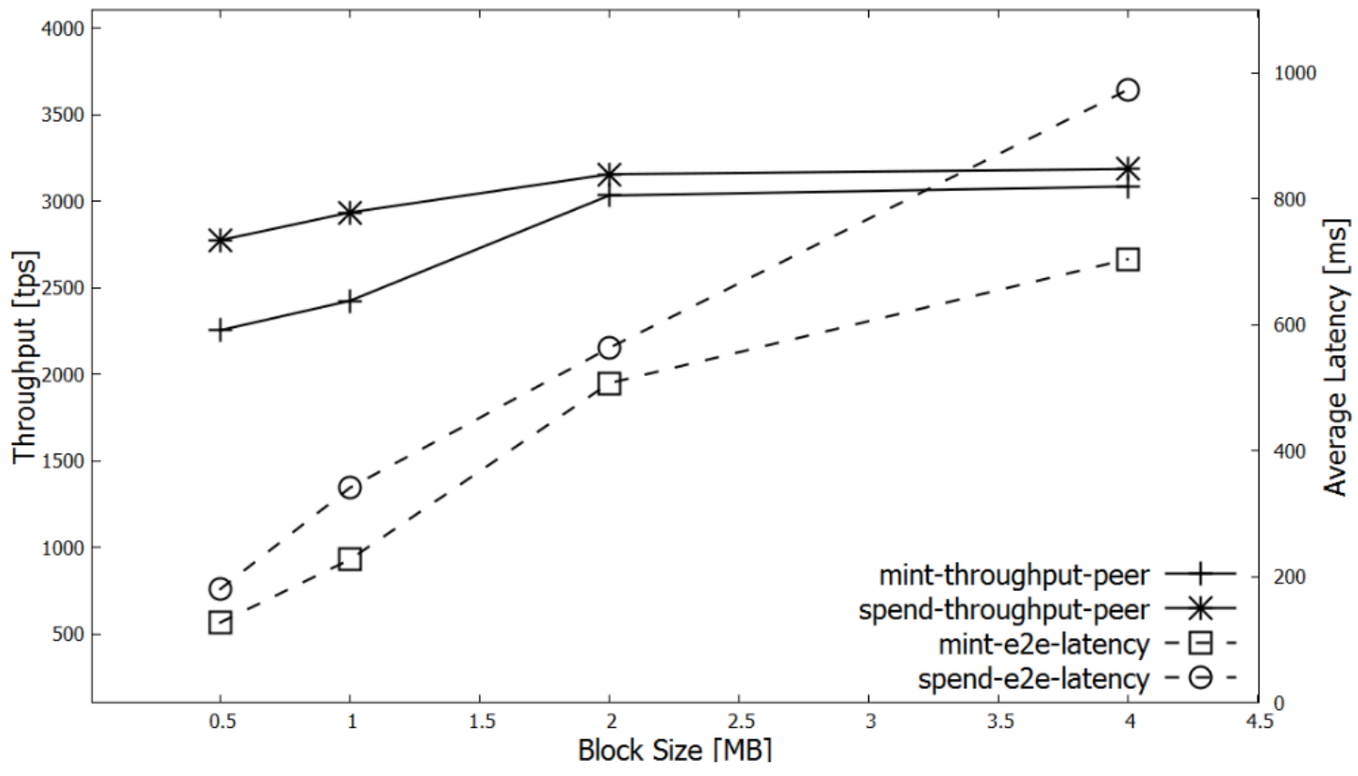


Figure 6: Impact of block size on throughput and latency.

EXPERIMENT 2

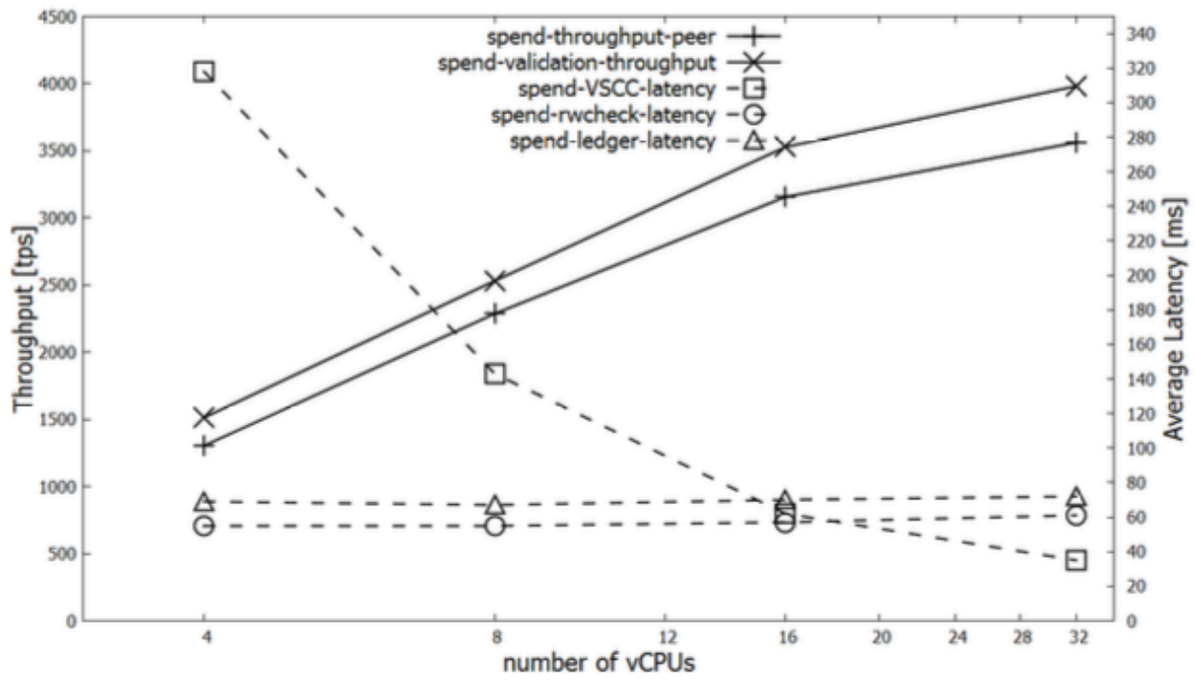
- Impact of peer CPU
- Fabric peers run many CPU – intensive cryptographic operations
- This experiment focused on the validation phase, as ordering with the Kafka ordering service has never been a bottleneck in our cluster experiments
- The validation phase, and the particular the VSCC validation of Fabcoin, is computationally intensive, due to its many digital signature verification
- We calculate the validation throughput at the peer based on measuring validation phase latency locally at the peer

EXPERIMENT 2

- This experiment suggests that future versions of Fabric could profit from pipelining the validation stages
- The *MINT* throughput is, in general, slightly lower than that of *SPEND*, but the difference is within 10%

EXPERIMENT 2

(a) Blocks containing only MINT transactions.



(b) Blocks containing only SPEND transactions.

APPLICATIONS AND USE CASES

- Major cloud operators already offer (or have announced) "blockchain-as-a-service" running Fabric, including Oracle, IBM, and Microsoft
- Examples include a food-safety network, cloud-service block-chain platforms for banking, and a digital global shipping trade solution
- Foreign exchange (FX) netting. A system for bilateral payment netting of foreign exchange runs on Fabric. It uses a Fabric channel for each pair of involved client institutions for privacy

CONCLUSION

- Fabric is a modular and extensible distributed operating system for running permissioned blockchains
- It introduces a novel architecture that separates transaction execution from consensus and enables policy-based endorsement and that is reminiscent of middleware-replicated databases
- Through its modularity, Fabric is well-suited for many further improvements and investigations
- Future work will address
  - Performance by exploring benchmarks and optimizations
  - Scalability to large deployments
  - Privacy and confidentiality for transactions and ledger data through cryptographic techniques

QUESTIONS