HYPERLEDGER
IROHA
Hyperledger Iroha in a nutshell:
DLT platform & project history

Agenda

- Overview of Iroha and comparison with other platforms
- Iroha architecture and security
- YAC and its properties
- Roadmap
- Conclusions and future steps

Organizational structure

HYPERLEDGER

Checks if the project is conformant to the standards

Development

SORAMITSU ソラミツ

Individuals and companies

Community is here!

Contributors

SORAMITSU ソラミツ

and other companies…

Individuals with defined roles and structure

Maintainers

Support development and ensure quality

What is Hyperledger Iroha?

Our mission:

Empowering people by providing them with decentralized technological solutions

Our vision:

Create simple & straightforward blockchain platform for enterprise, and blockchain enthusiasts

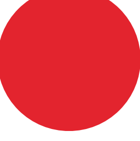
What problems Iroha is trying to solve?

- It tries to lower a high complexity of DLT software for platform integrators, developers of blockchain client application and users.
- Bring back trust and scalability in private networks with a complete byzantine fault-tolerance.
- There is a lack of C++ powered DLT frameworks — so we are one of the pioneers.

Where it all started?

3’149 C++ loc

40’262 loc



Sep 26 2016

Japanese companies propose Iroha to HL,

as «inspired by Fabric C++ project with Sumeragi consensus»

https://docs.google.com/document/d/1sN-6mv-m85NlbI3ZjwFkDT0izTcxbUaZN9LjLEe045Y/edit

Where it all started?

51’341 C++ loc (+3’826)
100’506 loc

Polkadot project

61’260 C++ loc (+9’541)
79’306 loc
Project is active for 2 years already

thon

Aug 2 2018
Iroha v1.0 beta-4 is out with:
Faster throughput, pluggable SQL storage, tx status streaming, fuzzing, transaction batches

Now (Feb 12)
BFT Ordering Service
Windows support

How Iroha is different?

High
STRENGTH OF VISION
Low

Ethereum
Waves
IOTA
Corda (R3)
Multichain
NEM
IBM Blockchain
Hyperledger Fabric
Hyperledger Sawtooth
CONTENDERS
LEADERS
Hyperledger Iroha
Credits
Quorum
Uledger
Near
BigchainDB
Hydrachain
Hyperledger Burrow
Elements
Yggdrash
Stellar
Masterchain
Openchain
Symbiont Assembly
ChainCore / Sequence
Domus Tower Blockchain
LAGGARDS
CHALLENGERS

Low
MARKET AWARENESS
High

How Iroha is different?

Factor per platform	Hyperledger Fabric (and IBM blockchain)	Hyperledger Iroha	Hyperledger Sawtooth
Regional awareness	China! and the rest of the world	Asia, especially Japan	USA
Differentiators	Extendable deployment architecture, «channels»	Universal peer role, SQL state, linearly scalable consensus	Transaction processors, pluggable components
Is this a blockchain?	Yes (although it stores invalid transactions)	Yes	Yes
API	gRPC & REST	gRPC	gRPC
Business logic layer	Smart contracts in Go, Java & Solidity	Commands and queries	Transaction families and processors
Contributing companies	IBM	Soramitsu	Intel
Byzantine fault-tolerance	—	+	+?
Has been already released and used in production	+	+- (used by some projects)	+?

How Iroha is different?

Factor per platform	Corda	Hyperledger Iroha	Ethereum
Regional awareness	UK, India, USA	Asia, especially Japan	The world
Differentiators	Scalability	Universal peer role, SQL state, linearly scalable consensus	Turing-complete smart contacts, same codebase for public and private
Is this a blockchain?	No	Yes	Yes
API	JSON-RPC?	gRPC	JSON-RPC
Business logic layer	Transactions processors in Kotlin?	Commands and queries	Solidity smart contracts
Contributing companies	R3	Soramitsu	Ethereum foundation
Byzantine fault-tolerance	—	+	—
Has been already released and used in production	+	+- (used by some projects)	+?

How Iroha is different?

Factor per platform	IOTA	Hyperledger Iroha	Waves
Regional awareness	Switzerland, Germany…	Asia, especially Japan	Russia
Differentiators	IOT-focused solution	Universal peer role, SQL state, linearly scalable consensus	Built-in decentralized exchange
Is this a blockchain?	No	Yes	Yes
API	?	gRPC	JSON-RPC
Business logic layer	?	Commands and queries	Non-turing-complete SC
Contributing companies	IOTA foundation	Soramitsu	Waves
Byzantine fault-tolerance	—	+	—
Has been already released and used in production	?	+- (used by some projects)	+

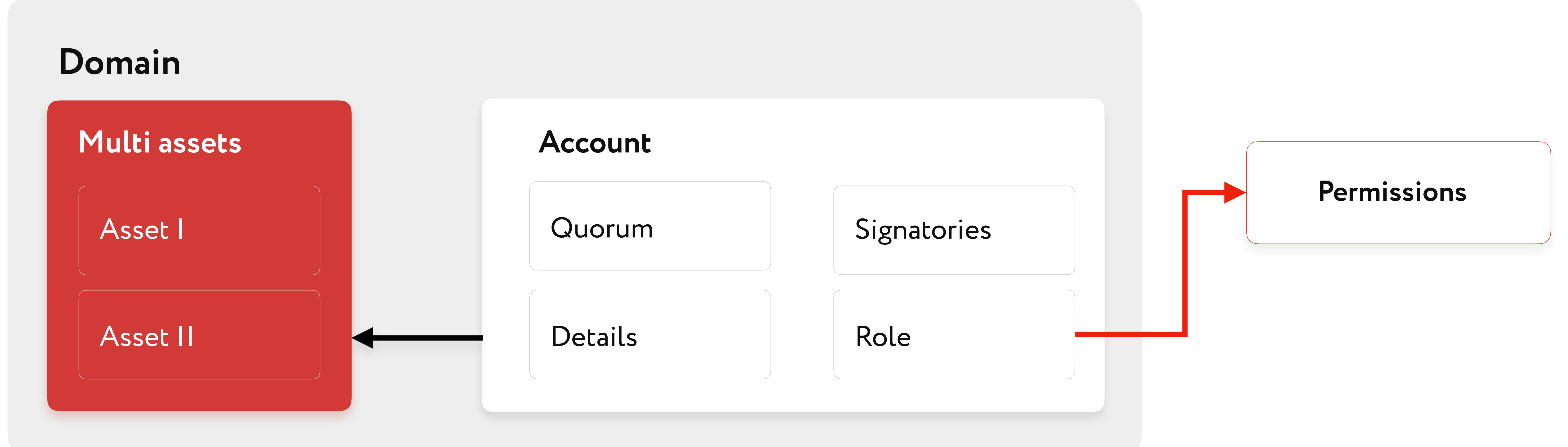
Features of Iroha

- Command-driven architecture
  - Asset management
  - Identity management
- Support of linux, macOS, Windows environment
- Byzantine fault-tolerant ordering service and consensus
- Role-based access control
- Client libraries, including example apps for iOS, JS (Vue.JS), Android (Java 8)
- Universal peer role and easy scripted deployment with Docker and Ansible
- Multi-signature transactions

Command-driven architecture

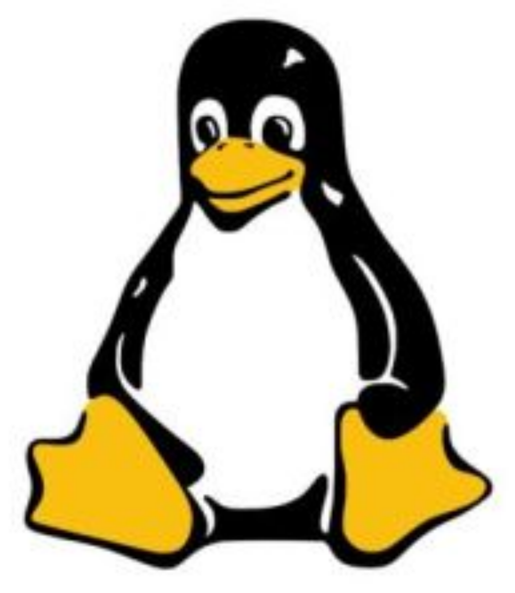
- Any atomic state-changing action is a «command», which is a piece of transaction
- A «query» is a request for a part of state: GetAssetInfo, GetAccountDetails, etc.

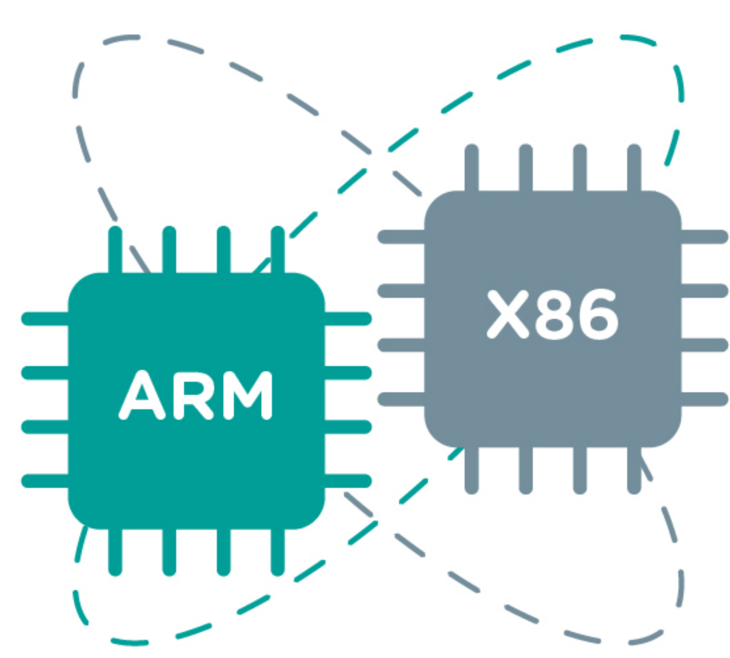
Domains	Assets	Account	Permissions
CreateDomain	CreateAsset	CreateAccount	CreateRole
	AddAssetQuantity	AddSignatory	AppendRole
	SubtractAssetQuantity	RemoveSignatory	DetachRole
	TransferAsset	SetAccountQuorum	GrantPermission
		SetAccountDetail	RevokePermission



Multi-platform support

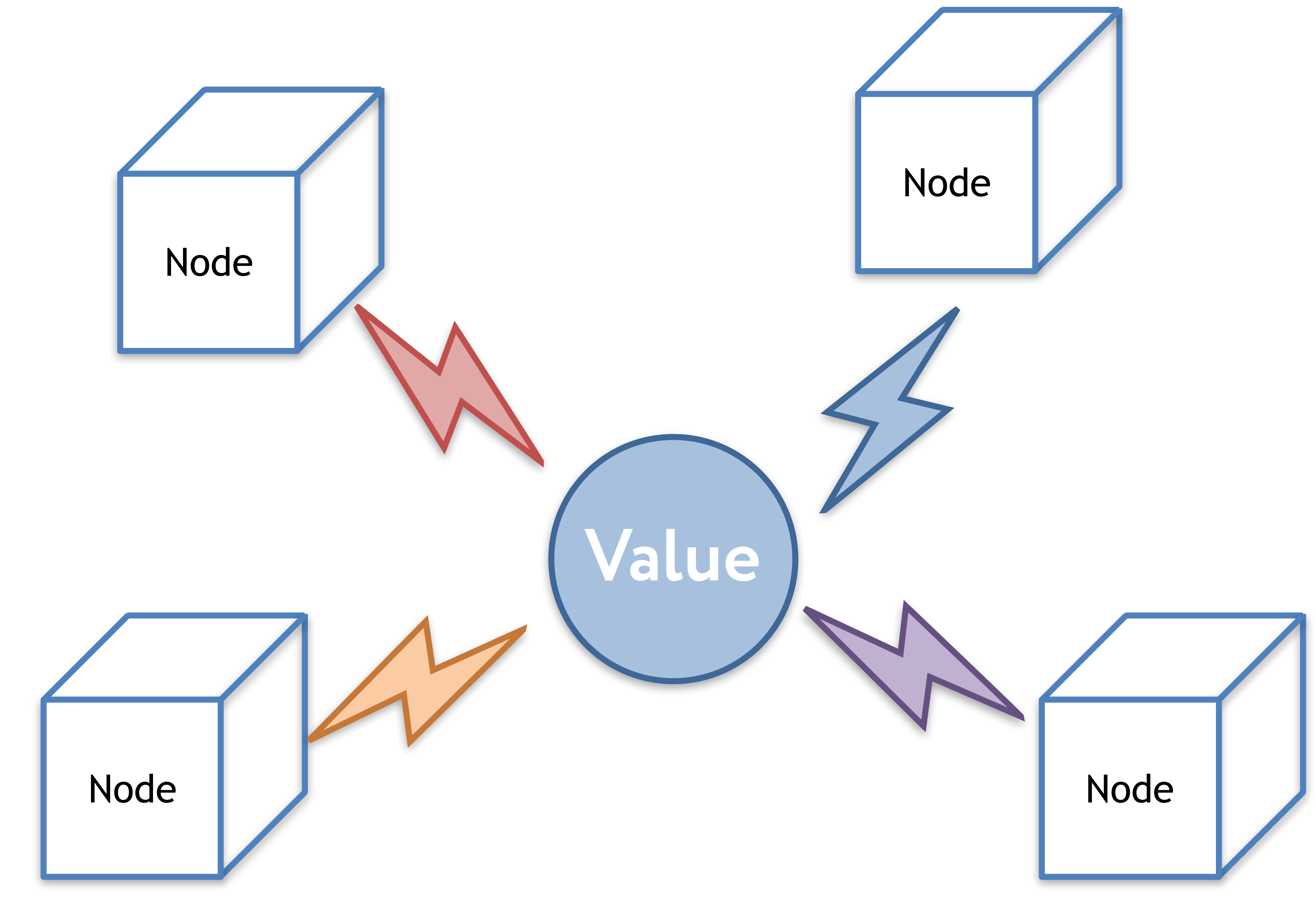
- Iroha supports linux, Windows, macOS software environment, with hardware layer including x86 and ARM-powered systems for IoT and chain supply use-cases.





Byzantine fault-tolerant consensus

- Iroha has novel, fast, and highly secure consensus algorithm, called Yet Another Consensus, which protects Iroha networks from failures or adversary participants.



Role-based access control

- Iroha accounts have rights with respect to their multiple roles
- Genesis block is an initial place where they are defined

Domains

Account

admin
alice
bob

Role

admin
user
money_creator

Permissions

can_append_role
can_detach_role
can_create_role
can_add_asset_qty
can_add_peer
can_add_signatory
can_create_account
can_create_asset
can_create_domain
can_remove_signatory
can_set_quorum
can_transfer
can_receive
can_subtract_asset_qty

can_read_assets
can_get_roles
can_get_my_account
can_get_all_accounts
can_get_my_signatories
can_get_all_signatories
can_get_my_acc_ast
can_get_my_acc_detail
can_get_all_acc_ast
can_get_my_acc_txs
can_get_all_acc_txs
can_get_my_acc_ast_txs
can_get_all_acc_ast_txs
can_grant_add_signatory

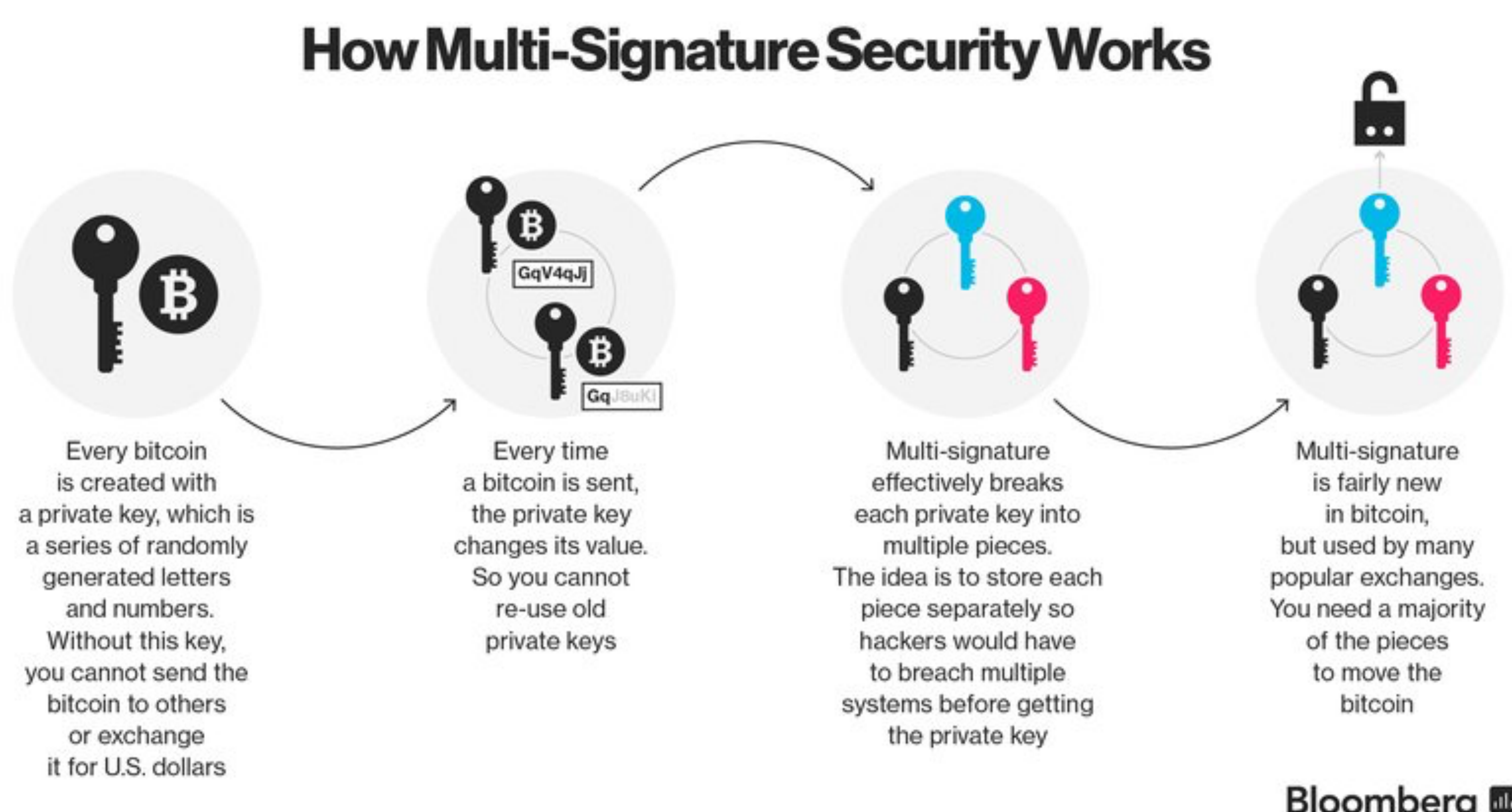
Client libraries

- https://github.com/hyperledger/iroha-java
- https://github.com/hyperledger/iroha-python
- https://github.com/hyperledger/iroha-ios
- https://github.com/hyperledger/iroha-javascript

We also have youtube videos explaining how to use the libraries with a lot of details here:

- https://www.youtube.com/watch?v=CsMIpZXWTLo iOS
- https://www.youtube.com/watch?v=PN7WoLReDs4 Python SDK
- https://www.youtube.com/watch?v=_HZB58jqi9c Java 8

Multisignature transactions



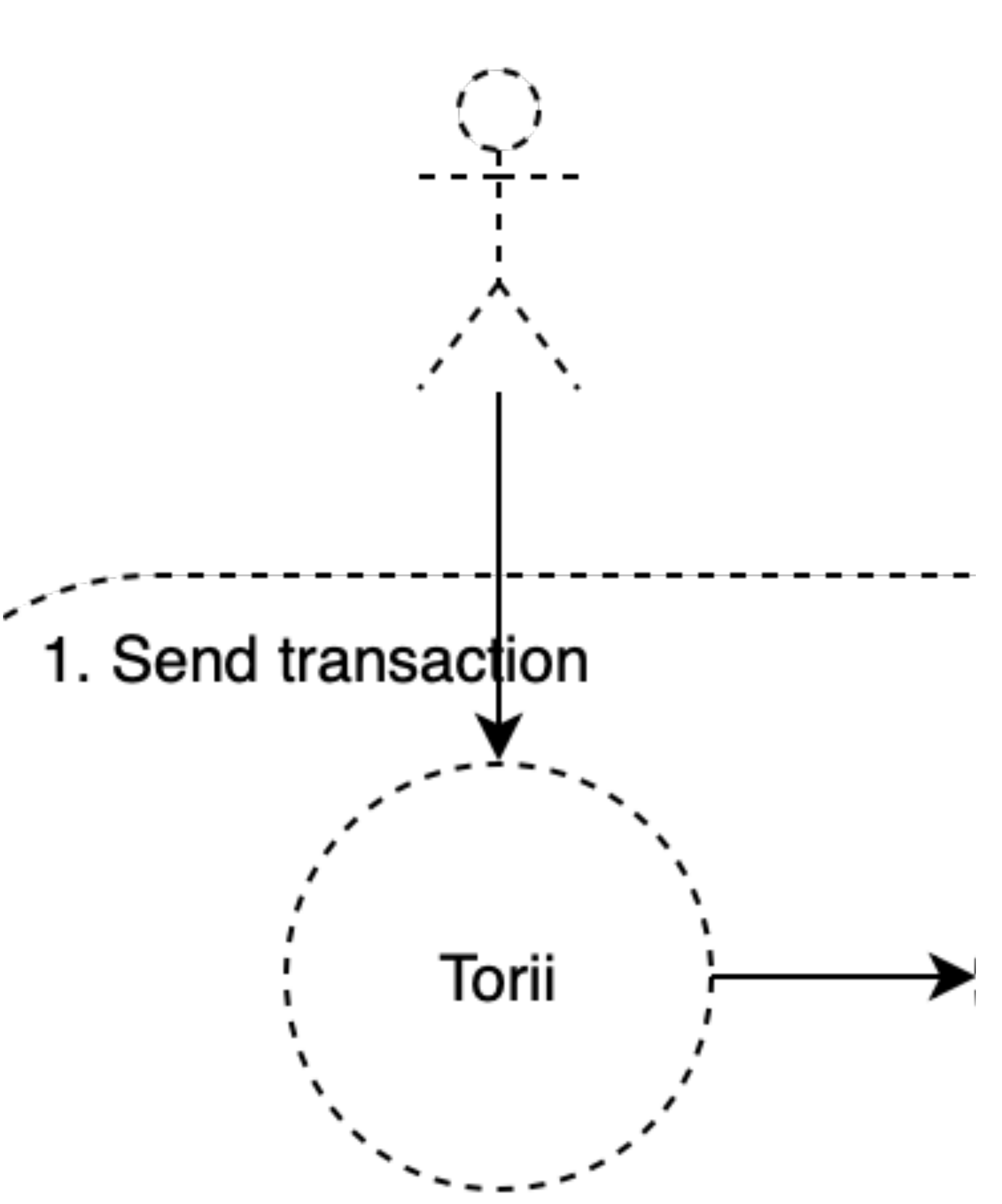
1'. Exchange shared state (unsigned transactions)
4. Send proposal
1. Send transaction
2. Send statelessly valid transaction
3. Send statelessly valid transaction
Torii
MST
Ordering Gate
4. Received proposal
Ordering Service
4. Received proposal
Ordering Gate
MST
Torii
5. Relay proposal
World State View (PostgreSQL DB)
Simulator
Block Creator
Block Creator
7. Send verified proposal
Simulator
6. Validate proposal
World State View (PostgreSQL DB)
8. Send block
11. Update WSV
10'' Append / Send block
Synchronizer
Blockstore
Block Consensus (YAC)
9. Vote, commit message exchange
Block Consensus (YAC)
10. Append block
Blockstore
Synchronizer
10' Request synchronization
10'' Exchange blocks

Torii (⛩)

gRPC server, that accepts incoming messages from clients:

- Transactions
- Transaction batches
- «Half-baked transactions» (with less signatures than required)
- Queries

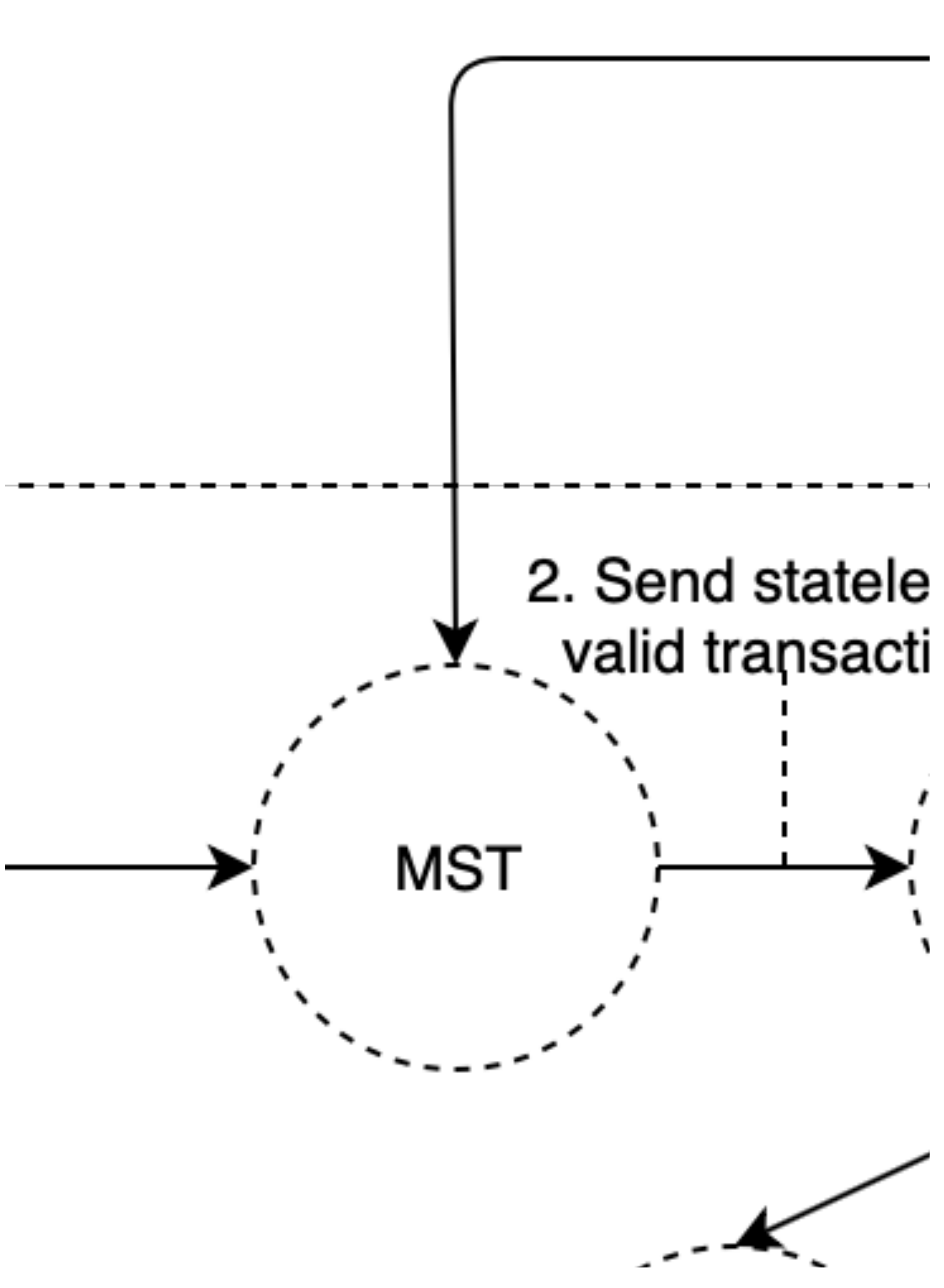
Responsibility: stateless validation of an incoming message



MST (actually MstProcessor)

gRPC server/client, that sends/accepts incoming messages from other peers.

Its' responsibility is to share the state of «half-baked» signatures across peers via Gossip protocol — so that any Iroha client can send a transaction with a partial «quorum» until the «quorum» is met; to any peer in the network.



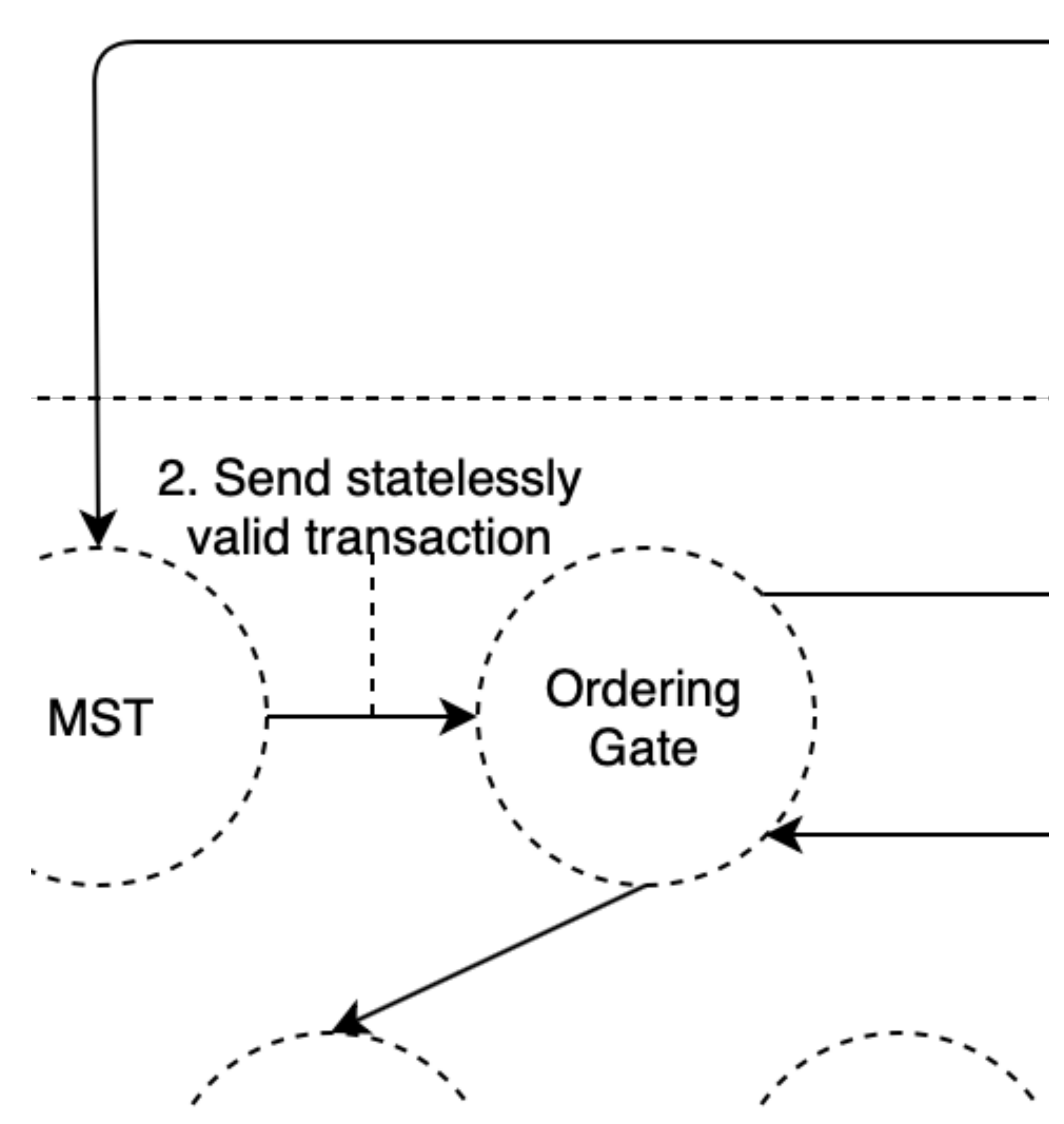
OrderingGate

gRPC server/client, that sends/accepts incoming messages from other peers.

It has to relay transactions to an Ordering Service, which collects transactions to a batch, called «proposal».

A proposal of transactions is a candidate for the next block in blockchain.

Ordering gate requests proposal from an Ordering Service, based on consensus round number.

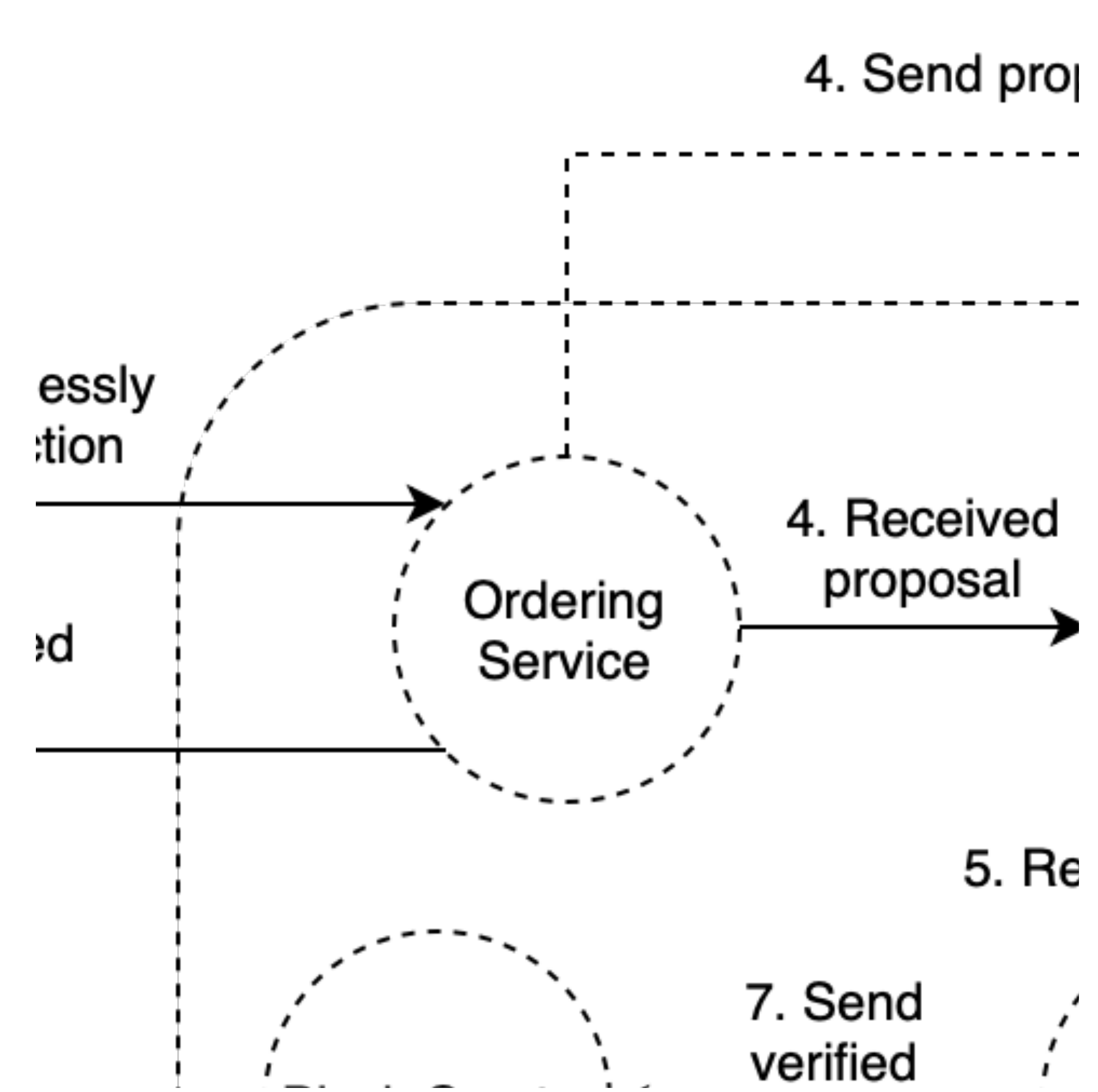


OrderingService

gRPC server/client, that sends/accepts incoming messages from other peers.

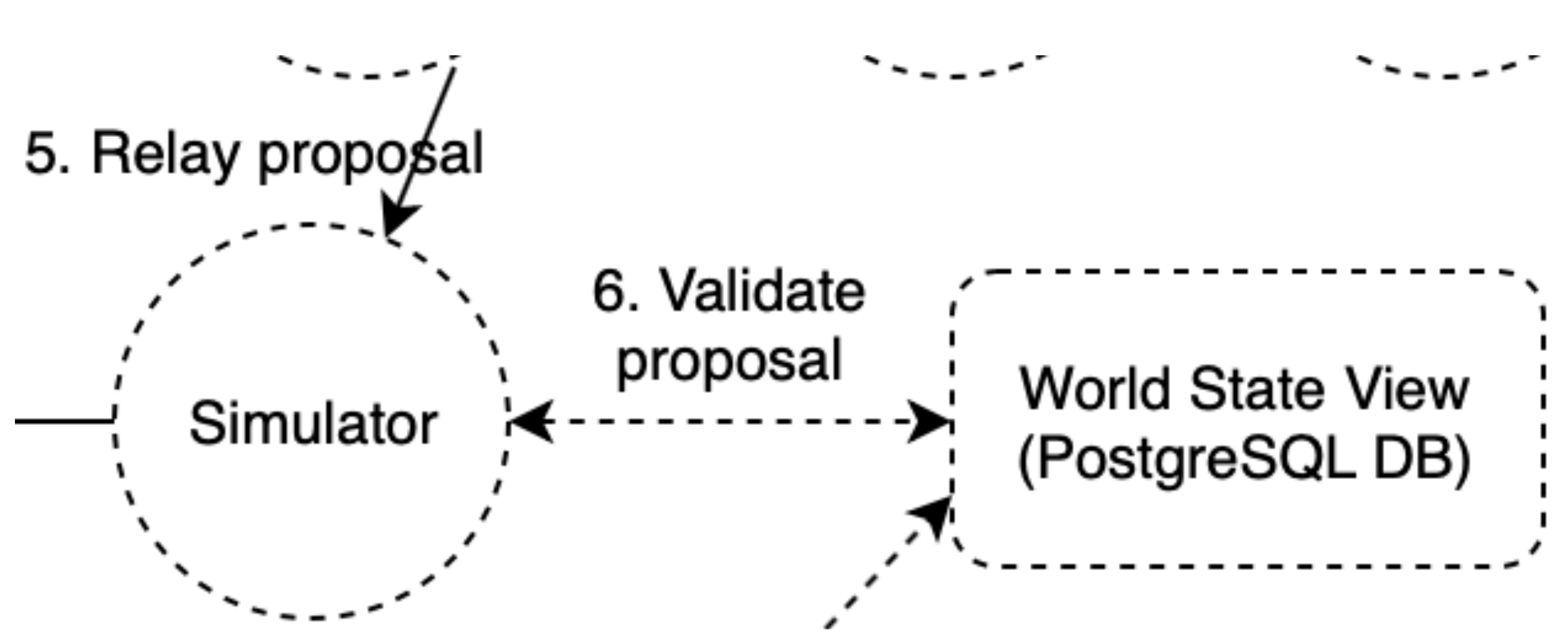
Proposal emitter and transaction accumulator.

Performs preliminary validation of proposals (clears out all rejected transactions from proposals, etc.)



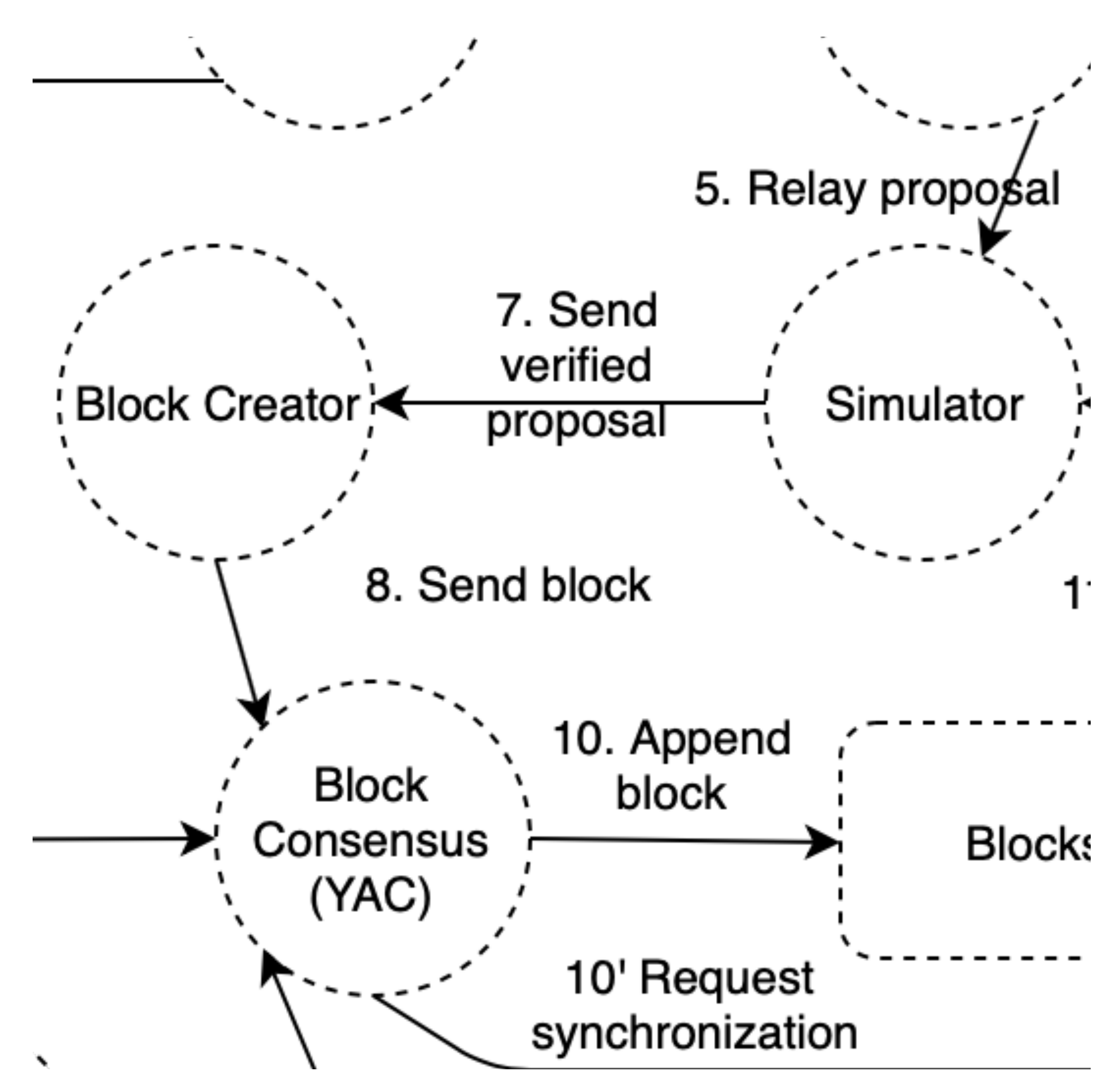
Simulator

Has an SQL database dependency. It uses «WorldStateView», or current representation of ER model based on blockchain for the purpose of **stateful validation** (reject invalid transactions where the rights aren't sufficient for actions: e.g. an account might not have enough assets for a transfer)



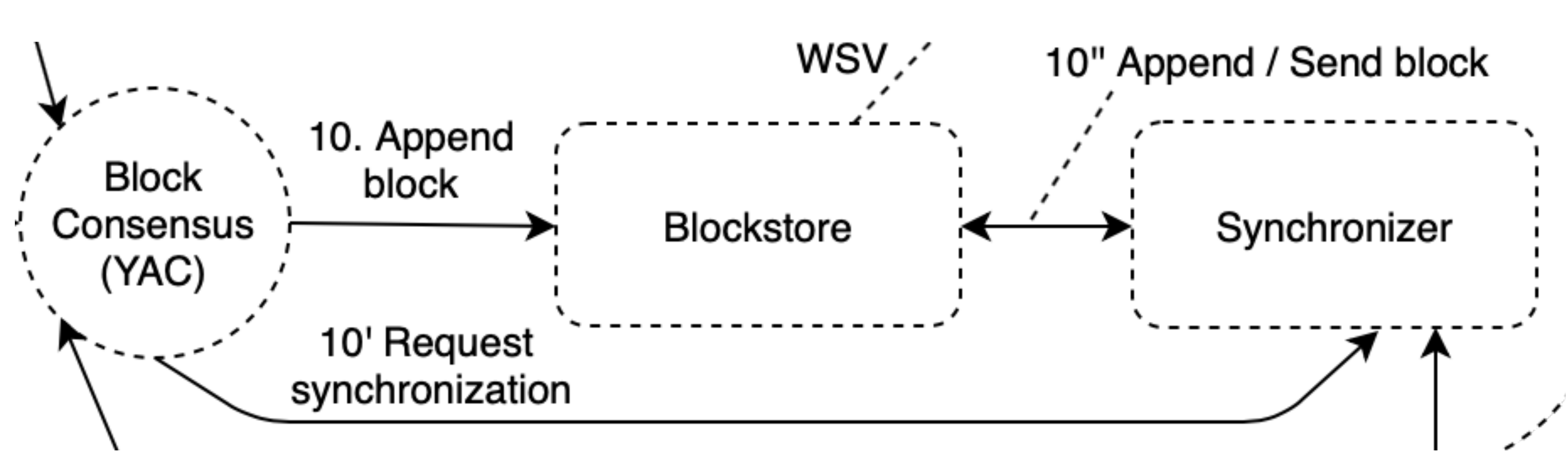
BlockCreator and ConsensusGate

BlockCreator adds a block meta to a verified proposal and then sends it to ConsensusGate, which executes logic for consensus in a distributed network. The consensus algorithm is called YetAnotherConsensus, it is byzantine fault-tolerant.



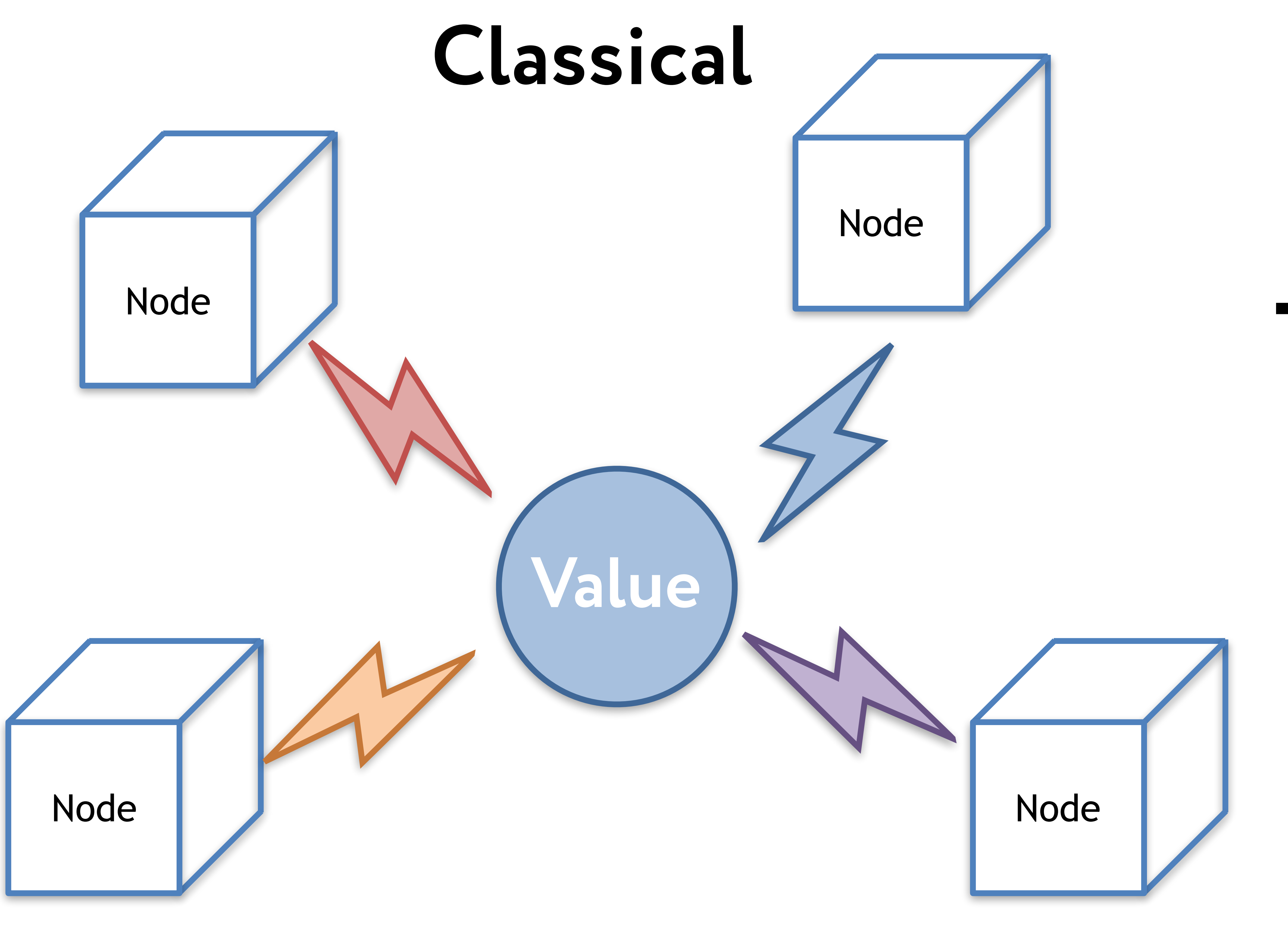
Synchronizer

After commit (affirmative decision for block candidate), the peer should put this block into a blockstore. If there's a gap (e.g. peer had only 4th block, but the network agreed on 11th block) the synchronizer downloads missing blocks based on peer signatures of commit for the block.



1'. Exchange shared state (unsigned transactions)
4. Send proposal
1. Send transaction
2. Send statelessly valid transaction
3. Send statelessly valid transaction
Torii
MST
Ordering Gate
4. Received proposal
Ordering Service
4. Received proposal
Ordering Gate
MST
Torii
5. Relay proposal
World State View (PostgreSQL DB)
Simulator
Block Creator
Block Creator
7. Send verified proposal
Simulator
6. Validate proposal
World State View (PostgreSQL DB)
8. Send block
11. Update WSV
10'' Append / Send block
Synchronizer
Blockstore
Block Consensus (YAC)
9. Vote, commit message exchange
Block Consensus (YAC)
10. Append block
Blockstore
Synchronizer
10' Request synchronization
10'' Exchange blocks

Consensus definition



+

Blockchain

- Agreement on **a list of values**
- **Trust** is important

Note: consensus algorithms solve **only write** problem, but not reads by clients of a service.

Blockchain platforms and their consensuses

Untrusted — Trust — Trusted

Public ← | → Private

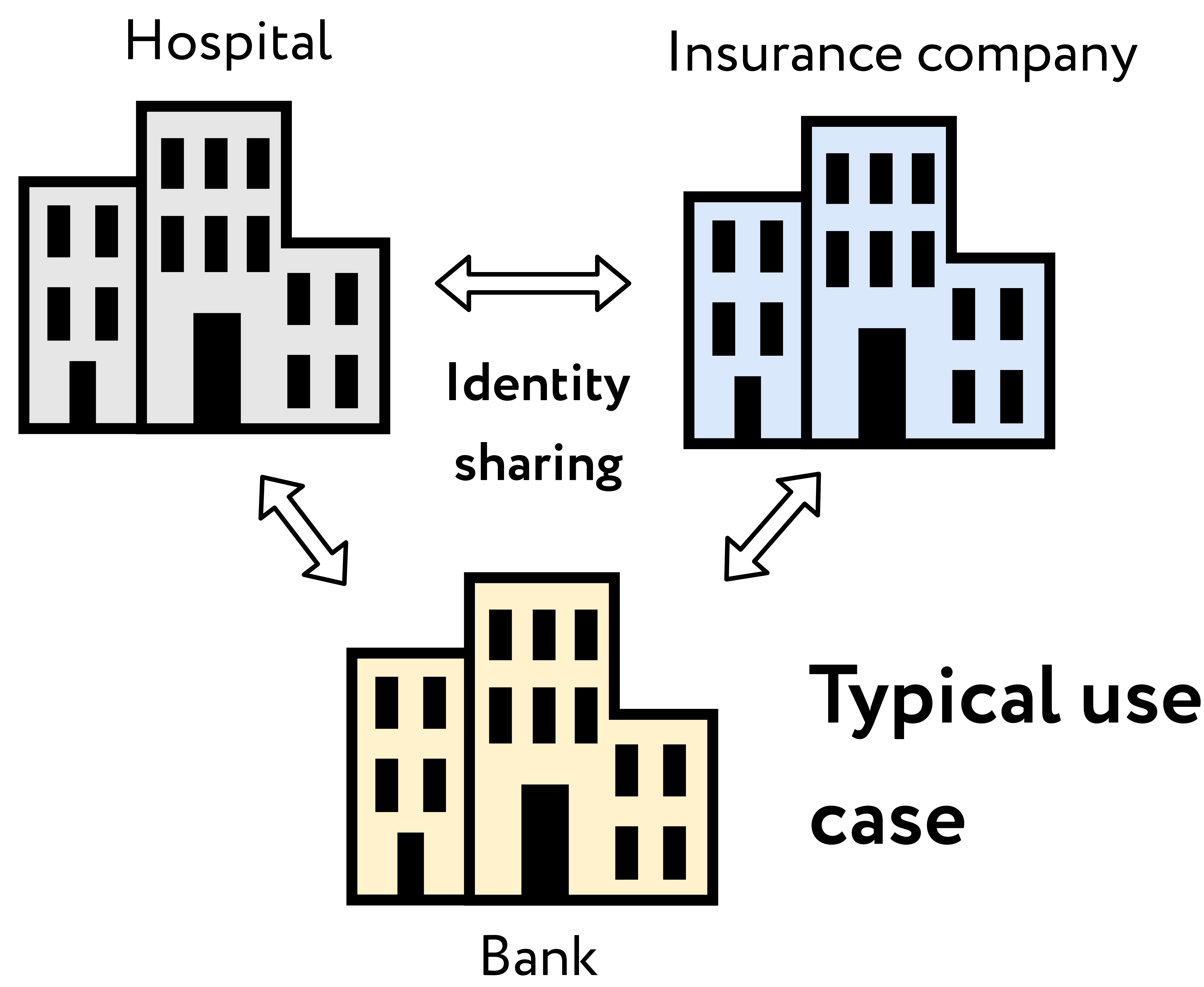
PoW
- Bitcoin
- Ethereum
- Monero

PoS
- EOS
- LISK

BFT
- Stellar
- Tendermint
- **Hyperledger Iroha**

CFT
- Paxos
- Quorum

Enterprise needs



Benefits of BFT

- supports untrusted consortium of companies
- high throughput or low latency of messages(transactions)
- doesn't spend lots of resources
- has the property of **finality**

Existing BFT consensuses

Tendermint

- \+ Uses existing and well-known 3 phase schema described in PBFT
- – O(N^2) scaling, N - number of peers
- – Stake-weighted voting

BChain

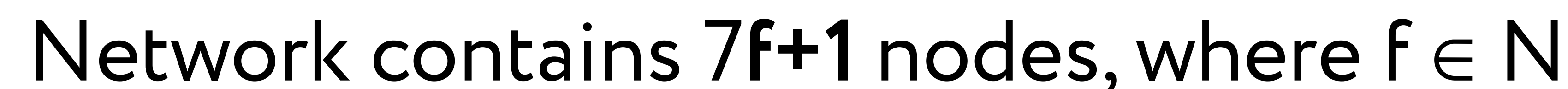
- \+ High throughput
- \+ O(N) scaling
- – High latency
- – Slow in malicious case

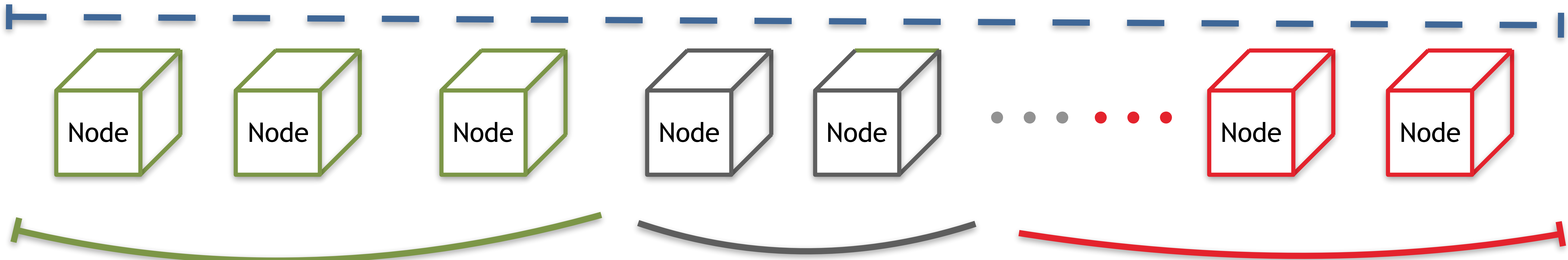
Hash graph

- \+ O(log(N)) scaling
- – The algorithm is licensed and can’t be reused as is
- – There is no open-source reference implementation

Should we make yet another consensus?

Byzantine fault tolerance description





5f+1 nodes follow the algorithm

f nodes might be inactive

f nodes are **malicious**. They can do everything:

- shut down
- lie
- collaborate with other faulty nodes

Assumptions

- Every node knows others network members
- Each node has a key pair for creating digital signatures
- The network tries to make an agreement on a new block

Initial statements

- **O(N)** scaling or less
- Low transaction latency
- Emphasis on validation step
- Input size does not affect agreement time
- Asynchronous environment*

*Consensus has **asynchronous** environment. There is no reliance on time in the network.

Yet Another Consensus architecture

Peer

OS

proposal

Peer

Validation

block

Agreement

agreed block

Peer

OS

Peer

OS

Peer

OS

Ordering Service(OS) is a component which is responsible for ordering messages.

Each peer has own OS. A common OS is chosen for each round.

Peer requests **proposal** from OS. Proposal contains list of messages for the validation.

Each peers creates own block from valid messages and votes for it in the network.

Peer switches to the next round and fetches new proposal.

Part 1. YAC core. Hash agreement

What is an input?

What about the output?

Vote is

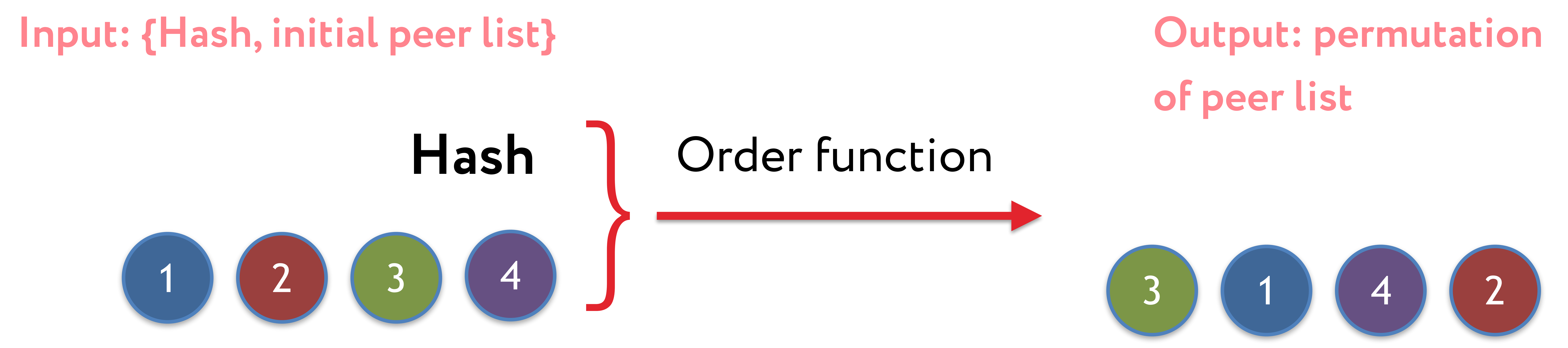
- Hash: String
- Round: <ConsensusRound: Int,
  RejectRound: Int>
- <Hash, Round> || Signature

Agreement is

- Votes: Vote[]

Votes indicate **commit** or **reject** of agreement on a hash in the round.

How does YAC core work? Order function



Order function constraints

- Pure function: same input generates same output
- Uniform distribution of hash leads to uniform distribution of elements in resulting list

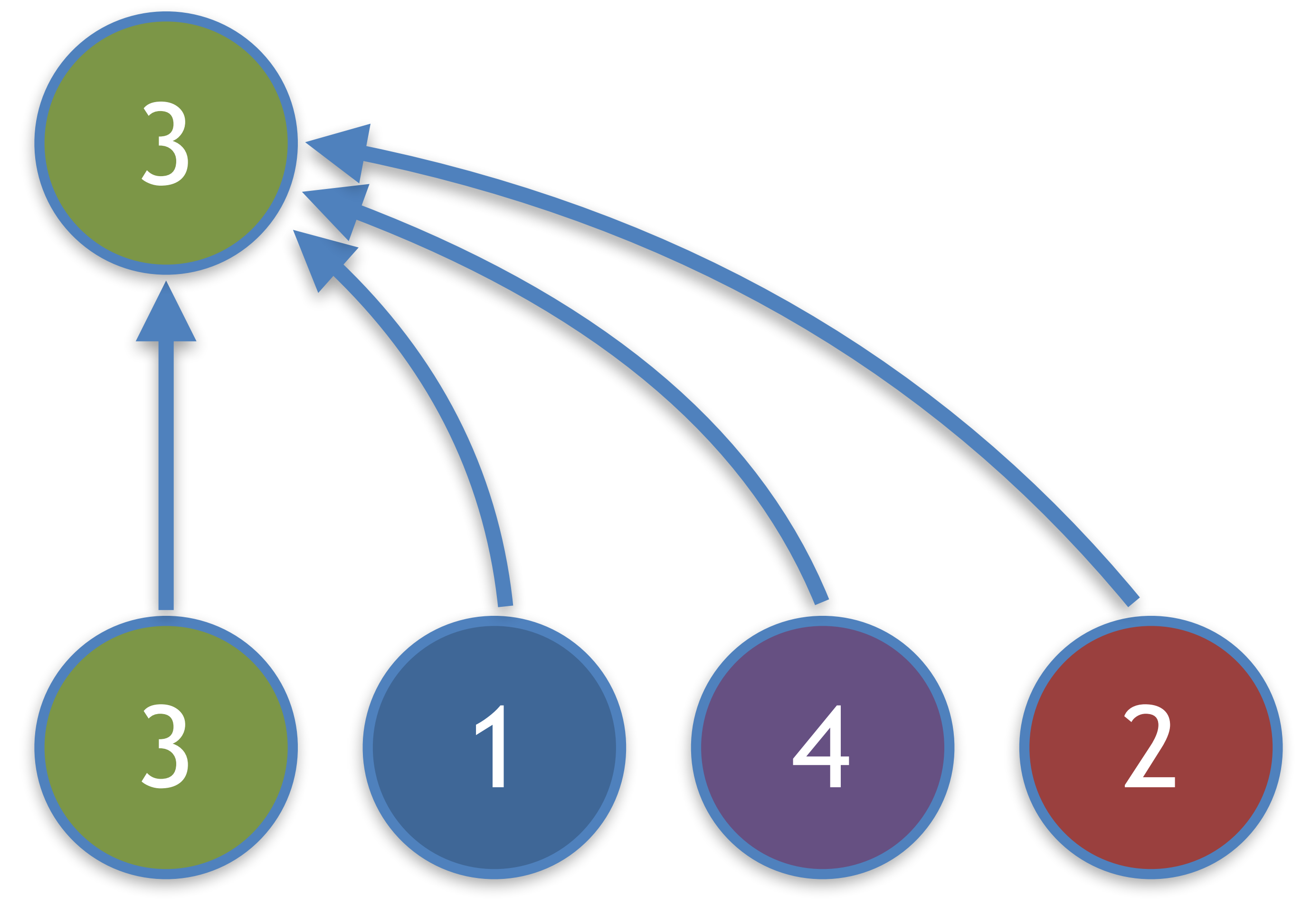
How does YAC core work? Agreement process

Propagation

- Peer 4 shares its state according to permuted peer list. It waits for some time until the next propagation.

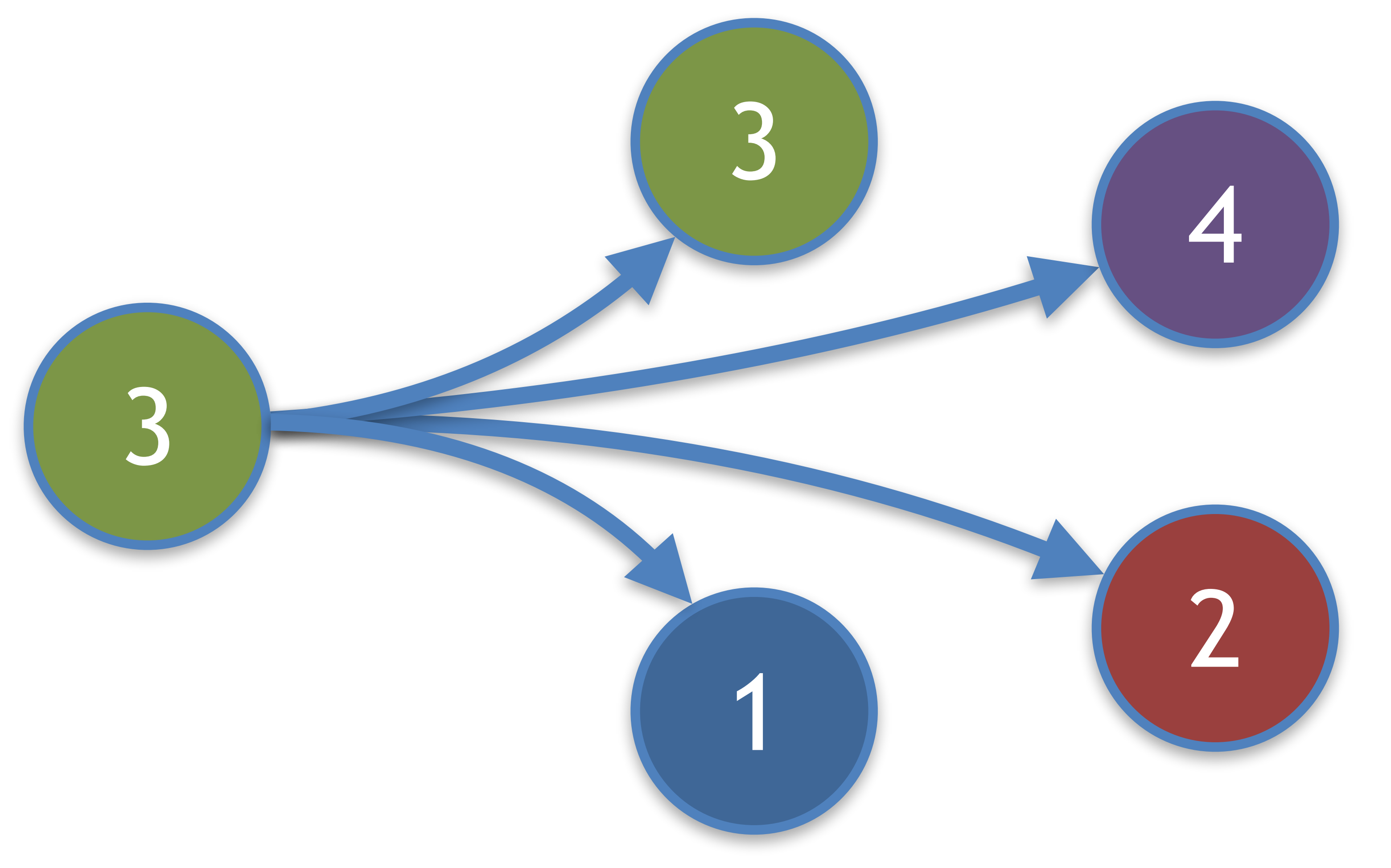
Collecting

Peer 3 collects the votes from the network and waits until the supermajority of votes are received.



Agreement

Peer 3 broadcasts the commit in the network. Everyone verifies the **commit** or **reject** message and applies the hash.



Note: all phases are performed simultaneously. The process is the same for all peers. There is no “leader” peer which shares commit, everyone can do it.

Commit & reject

Commit and reject messages are just a sets of votes(Vote[]). But what is the difference?

Commit contains supermajority (⩾ 5f+1) of votes for one hash.

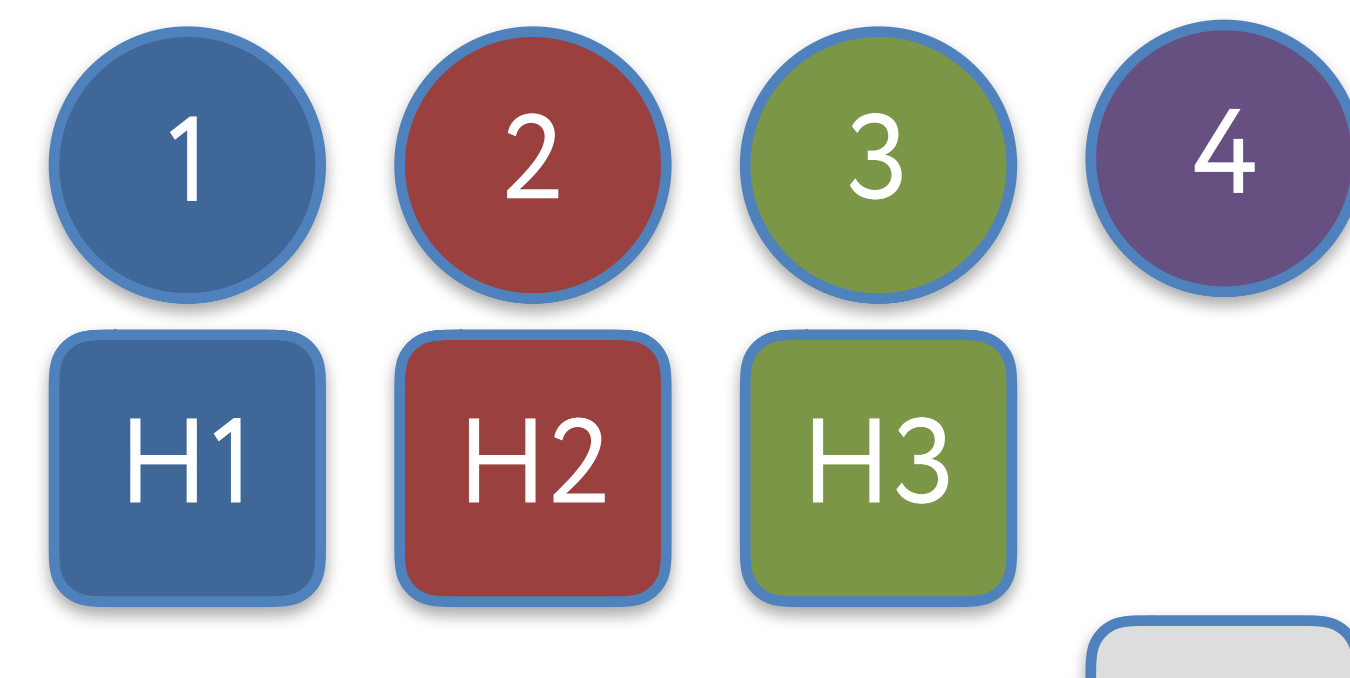
Reject proves that there is no agreement on a particular hash in current round.

Vote recap

Vote is

- Hash: String
- Round: <ConsensusRound: Int, RejectRound: Int>
- <Hash, Round> || Signature

Example:



Peers 1, 2 and 3 have different opinions about new value. Peer 4 doesn't respond something yet.

Nobody cares about the hash of peer 4, because reject message already exists.

Part 2. Round pipeline

Peer

round

OS

proposal?

Validation

block

YAC
core

votes

Outcome
Strategy

agreement
outcome

? - means that
value is optional.

- How can we transform block to vote?
- What we are going to do with votes?

Block to vote transition

Proposal + **Block**

\\ \\

P_hash: String. **B_hash: String**

Concatenation of hashes

Vote

- Hash: String

Same as we already have

Round

- Round: <ConsensusRound: Int, RejectRound: Int>

Sign vote with private key

Peer's key pair

- <Hash, Round> II Signature

Outcome strategy

Another task is to make a decision on **votes**.

Votes

- **Commit**
- **Reject**

Outcome Strategy

- **Commit**
- **Synchronisation**
- **Malicious OS**
- **Faulty OS**
- **BFT violation**

Commit & reject recap

- **Commit** - supermajority of votes for one hash
- **Reject** - proves that supermajority of votes cannot be collected

Outcome strategy (cont.)

Type	Condition	What to do: Storage	What to do: Round transition
➡ **Commit**	Commit with voted hash	add(Block)	R(i, j) ⇒ **R(i+1, 1)**
➡ **Synchronisation**	Commit with greater round	add(□ ← □ ← … ← □)	⇒ **R(i+k, j+l)**
➡ **Malicious OS**	Reject with different proposal hash	—	⇒ **R(i, j+1)**
➡ **Faulty OS**	Commit with {None, None} hash	—	⇒ **R(i, j+1)**
➡ **BFT violation**	Reject with same proposal but different block hash	¯_(ツ)_/¯	¯_(ツ)_/¯

Part 3. Ordering Service rotation

The last question is how to pick the Ordering service?

Block → **Hash** → **Order**

i^{th}

"a1fe5c"

3 1 4 2

Get R(i, **0**)

Get R(i, **1**)

Get R(i, **3**)

Get R(i, **4**)

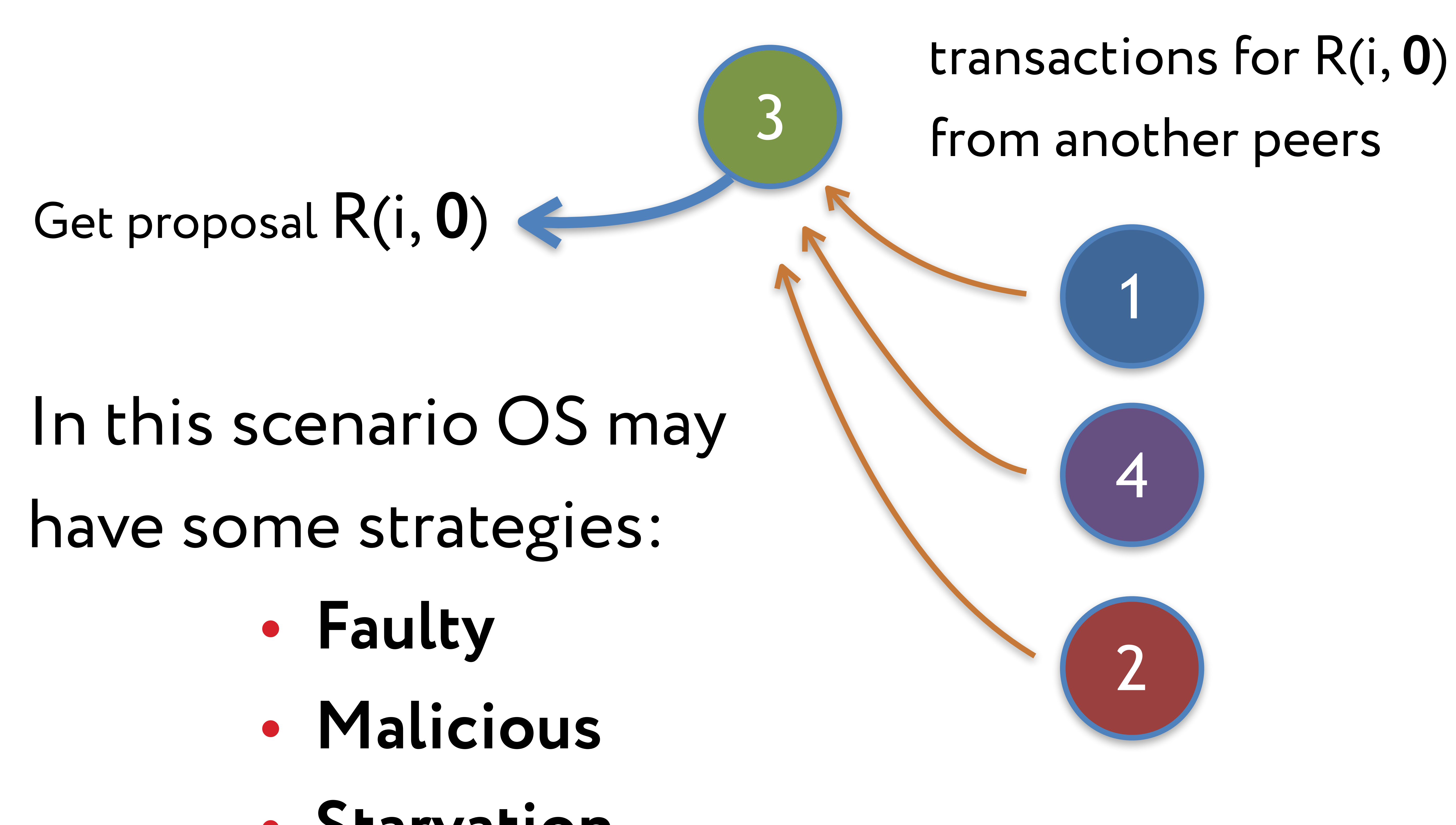
Let's apply the same approach as with the hash agreement!

For one **block round** we may request a proposal from the corresponding OS.

But what about collecting the transactions?

Collecting the transactions. Naive approach

Naive approach implies that we can send our transactions to the **current** ordering service.



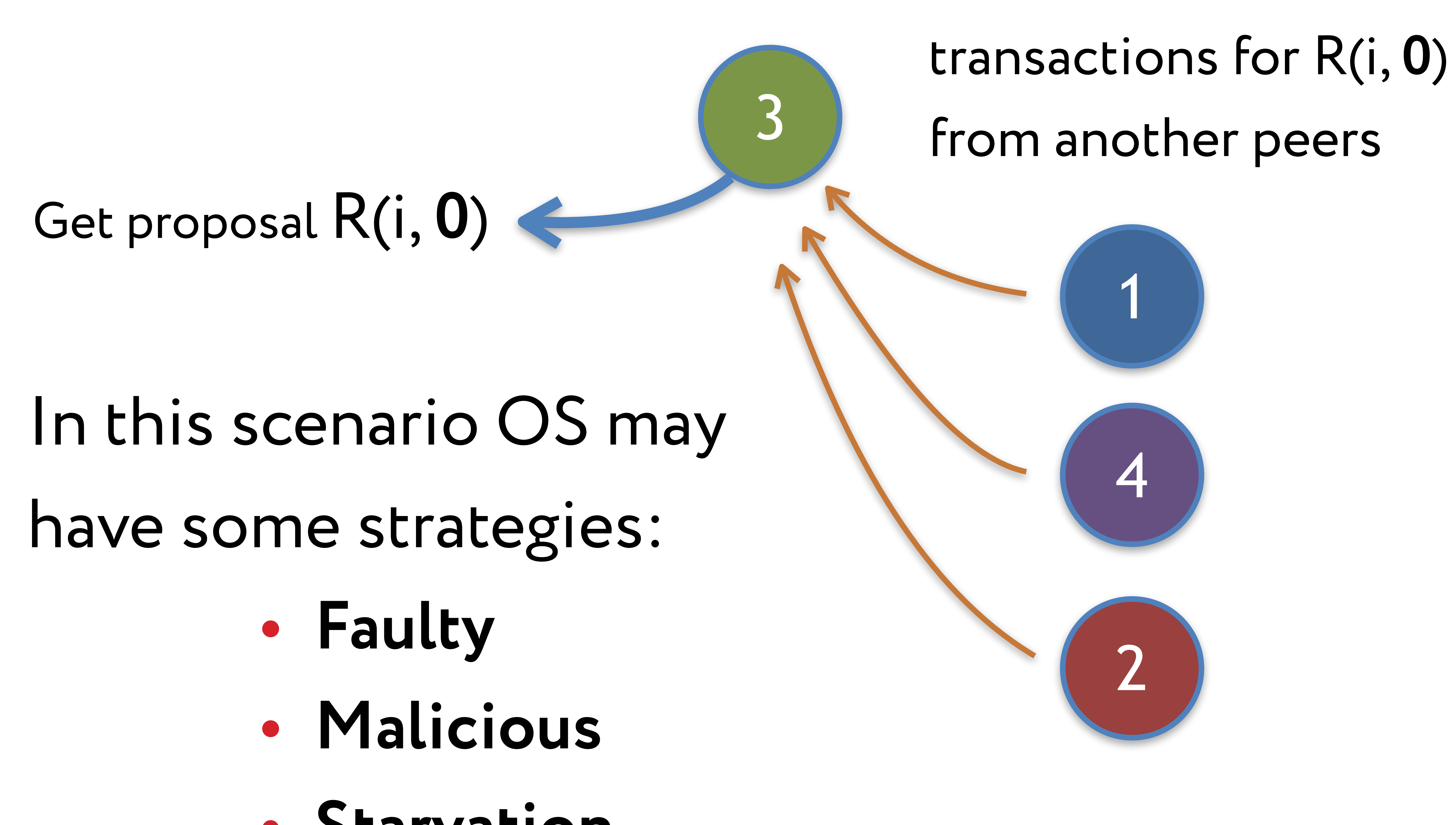
In this scenario OS may have some strategies:

- **Faulty**
- **Malicious**
- **Starvation**

How can we define the OS which collects enough transactions?

Collecting the transactions. Naive approach

Naive approach implies that we can send our transactions to the **current** ordering service.



In this scenario OS may have some strategies:

- **Faulty**
- **Malicious**
- **Starvation**

How can we define the OS which collects enough transactions?

Rounds and blocks

Block	Round	0	1	2	3
top - 2	i	proposal round	round for closing proposal	round for sending transactions	
top - 1	i+1	round for closing proposal	round for sending transactions		
top	i+2	round for sending transactions			

- proposal round
- round for closing proposal
- round for sending transactions

How do rounds and blocks correlate?

We have to know the permutation for next 3 rounds. And we want to update the order on new round.

The solution is to compute round with the block before the previous top block and send transactions to three ordering services in 3 different rounds.

Overview

OS

proposal?

Validation

block

YAC core

Agreement pipeline

OS

request proposal

votes

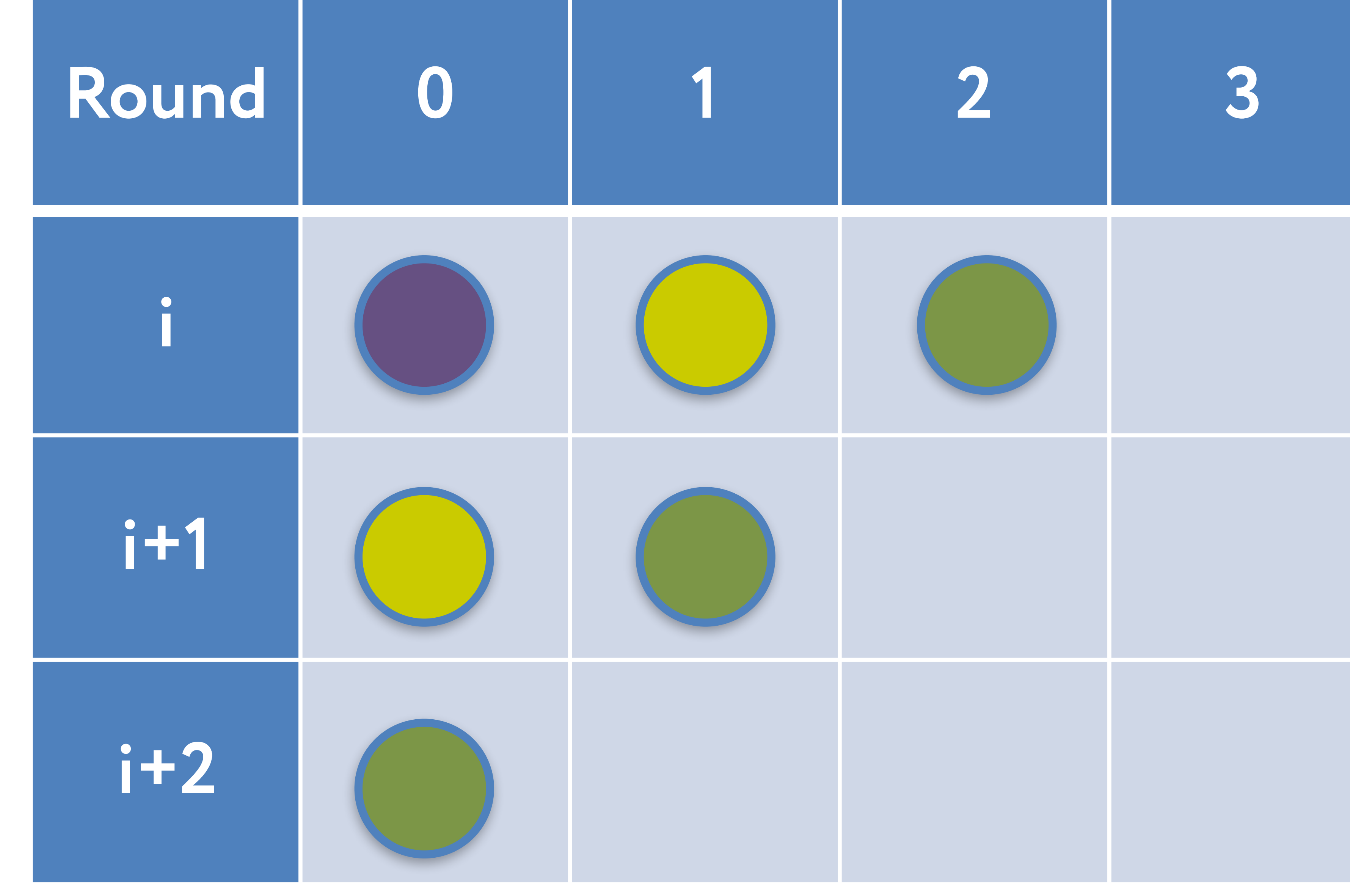
agreement outcome

Make new round

Storage committer

Outcome Strategy

Proposal generation



- proposal round
- round for closing proposal
- round for sending transactions

Future work

Current issues:

- Transaction latency is not minimal

✨ **Contribute to Iroha** ✨

Next research:

- BFT schema with theoretically minimal latency
- Ordering services collaboration
- Log-scaling of votes/commits propagation

Additional topics

- Parallelization of validation for BFT consensuses
- Consensus for public Iroha network

Roadmap

Quarter	Q1 2019	Q2 2019	Q3 2019	Q4 2019
Features	Updated consensus algorithm (from 3f+1 to 7f+1); Improved stability and performance; Final release of Iroha	Prototype of custom commands and queries	Polkadot project integration; smart contracts on WebAssembly VM	Parallelized validation for BFT consensus

- https://github.com/hyperledger/iroha/

✨ Contribute to Iroha ✨