DFINITY

The Internet Computer

Yvonne Anne Pignolet (yvonneanne@dfinity.org)
Thomas Locher (thomas.locher@dfinity.org)

DFINITY Foundation 🇨🇭

- emerged from early Ethereum community in 2015
- Swiss not-for-profit foundation, not a corporation
- world’s largest cryptography and blockchain RnD team
- 270+ team members globally

1600+
research papers

100 000+
academic citations

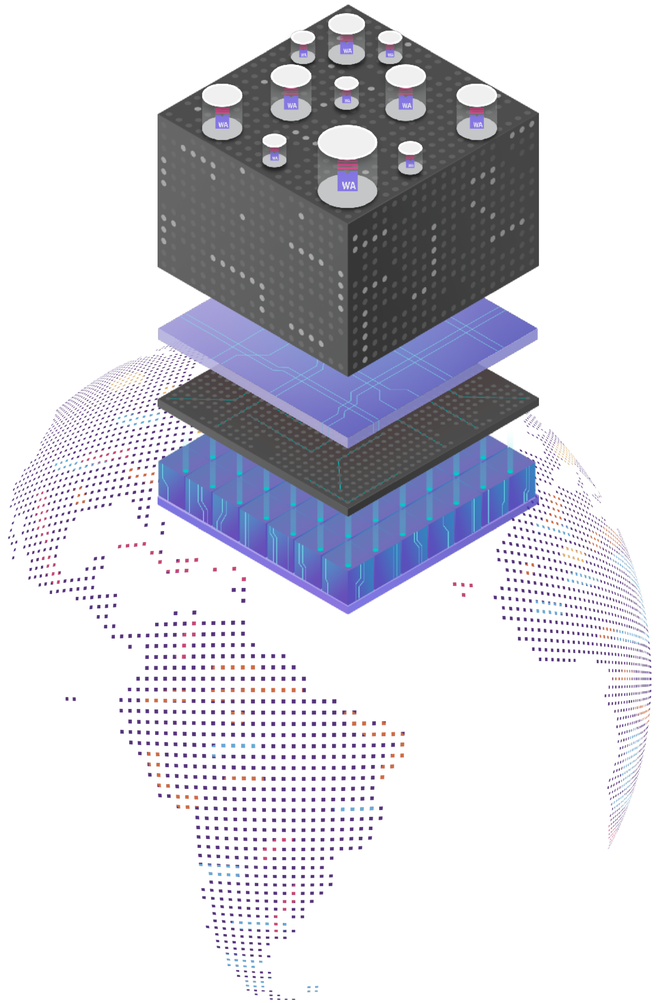
250+
technical patents

About us

- PhD in Distributed Computing 2009
- 14 years in industrial research labs (IBM Research, ABB Corporate research)
- @DFINITY since January 2019 / 2021

Outline

1. What is the Internet Computer?
2. Networking Layer
3. Consensus Layer
4. Message Routing Layer
5. The Internet Computer Today



1. What is the Internet Computer Protocol?

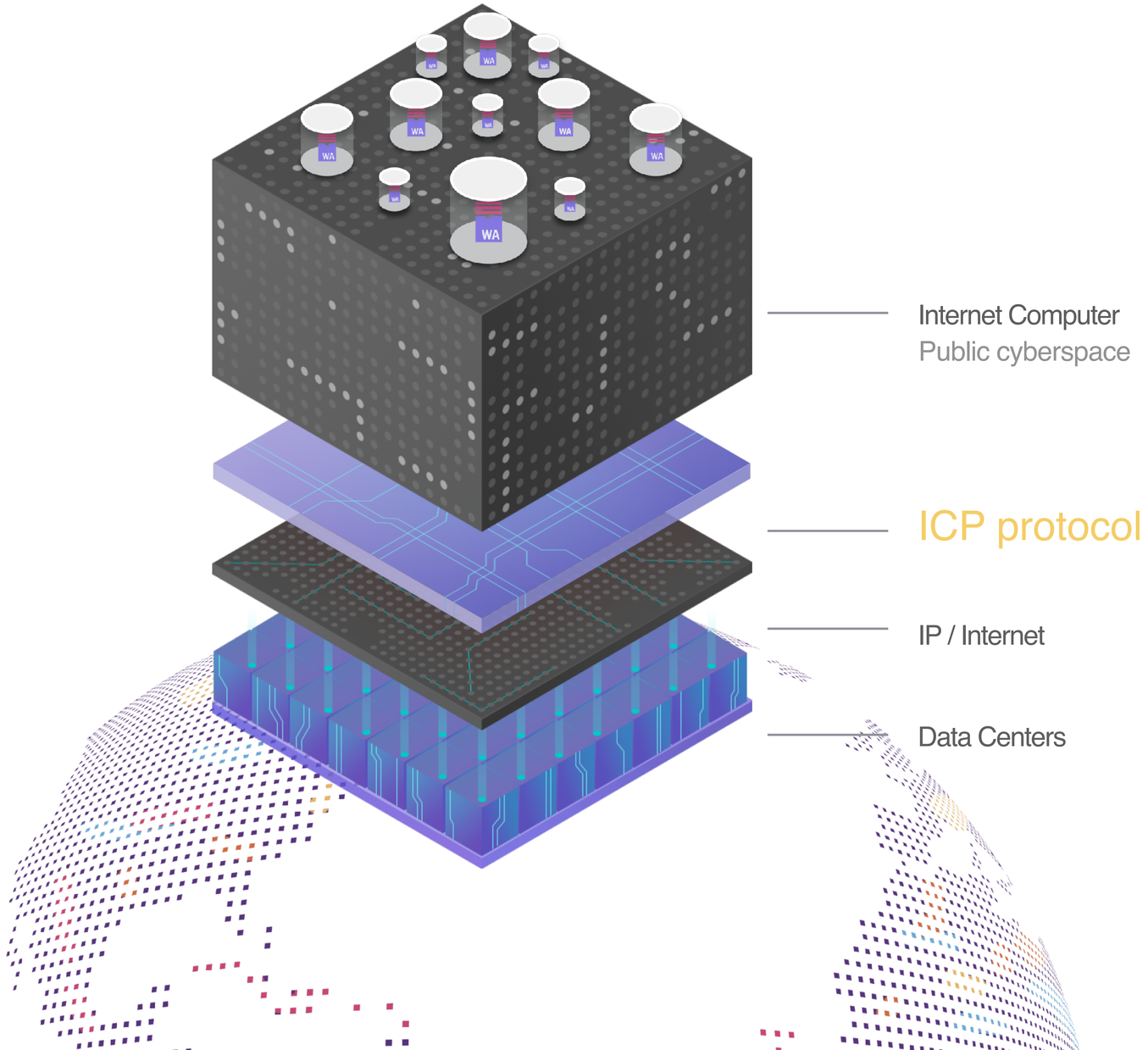
Protocol to run any computation, using blockchain technology for decentralisation and security

1.1 Internet Computer Protocol (ICP)

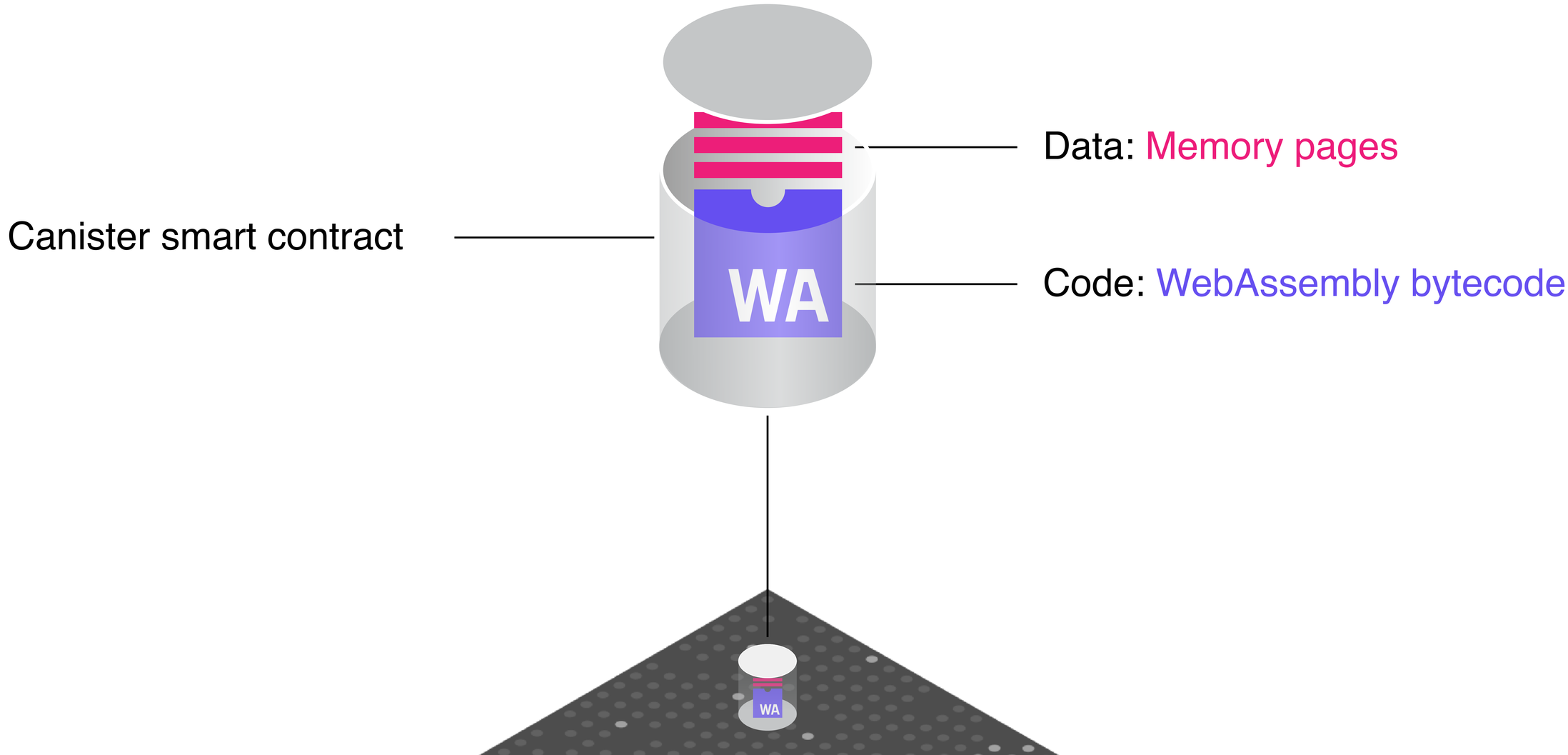
Coordination of nodes in independent datacenters, jointly performing any computation for anyone

ICP creates the
Internet Computer blockchains

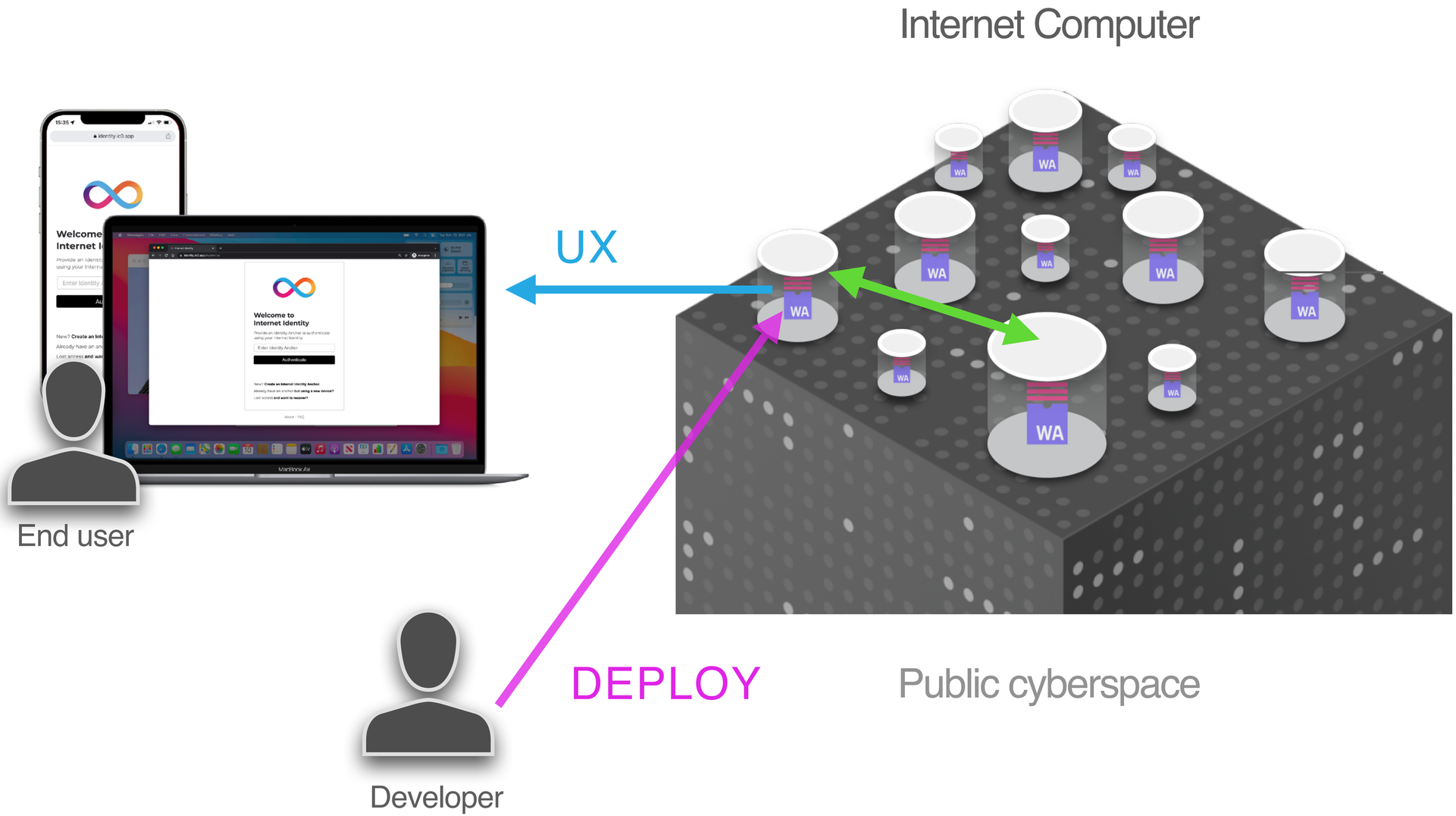
Guarantees safety and liveness of smart contract execution despite Byzantine participants



1.2 Canister Smart Contracts: Combination of Data and Code



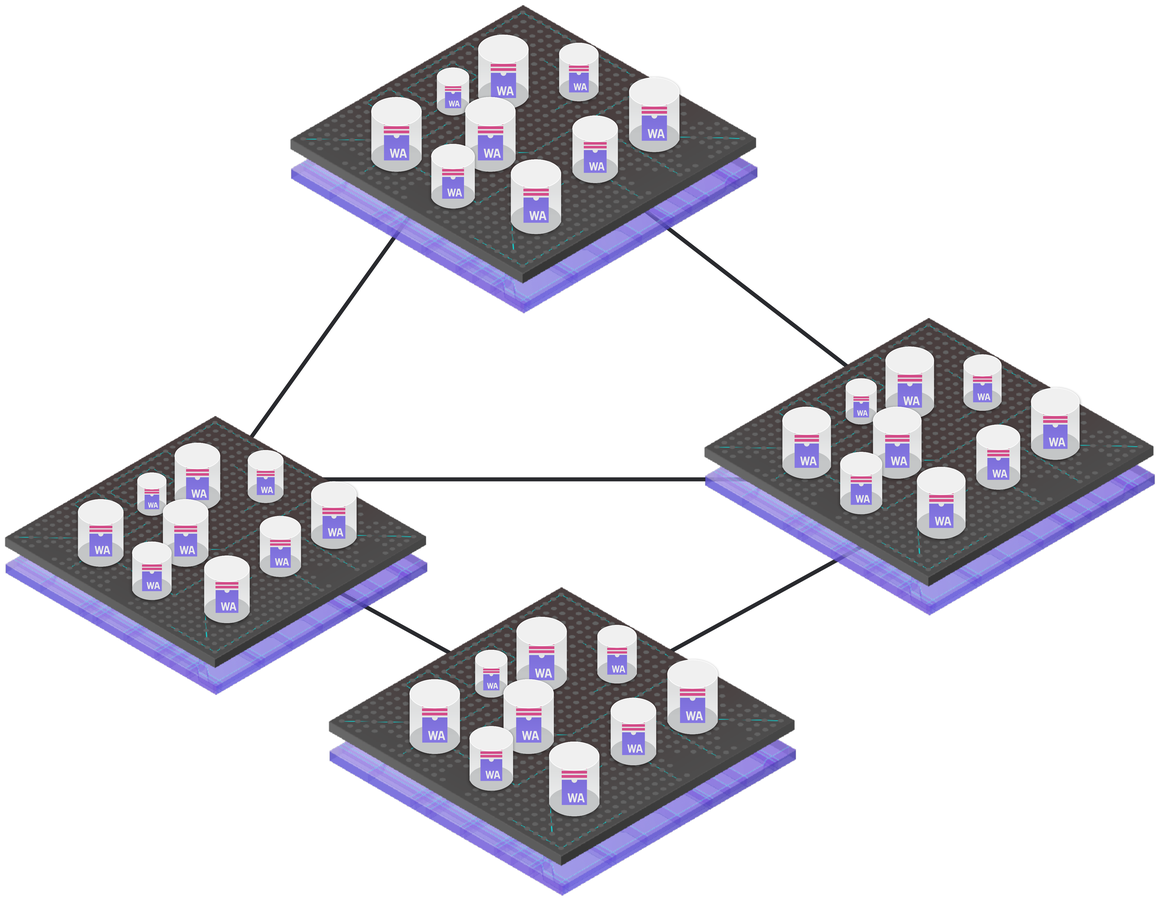
1.3 Developers and users interact directly with Canisters on the IC



1.4 Scalability: Nodes and Subnets

Nodes are partitioned into **subnets**

Canister smart contracts are assigned to different subnets



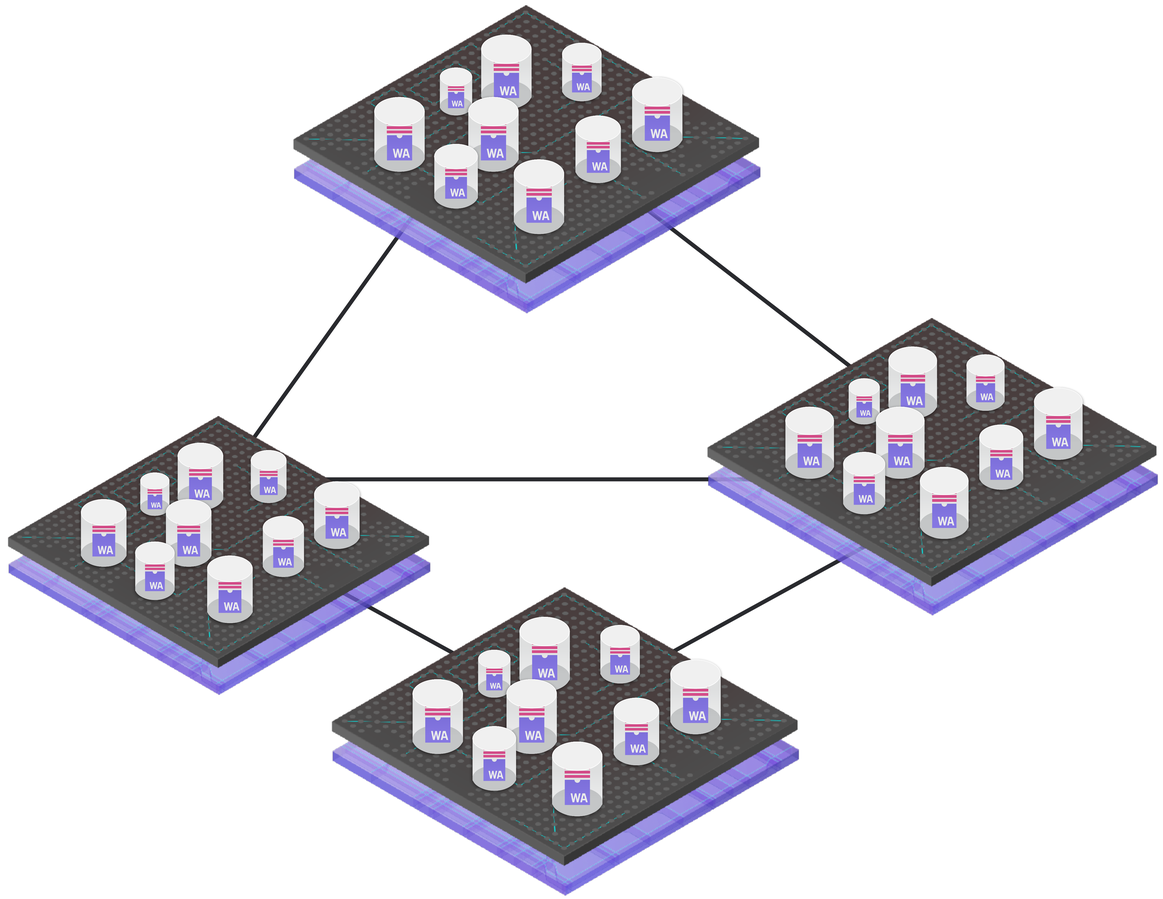
1.5 Each Subnet runs State Replication

State:
- canisters and their queues

Inputs:
- new canisters to be installed,
- messages from users and other canisters

Outputs:
- responses to users and other canisters

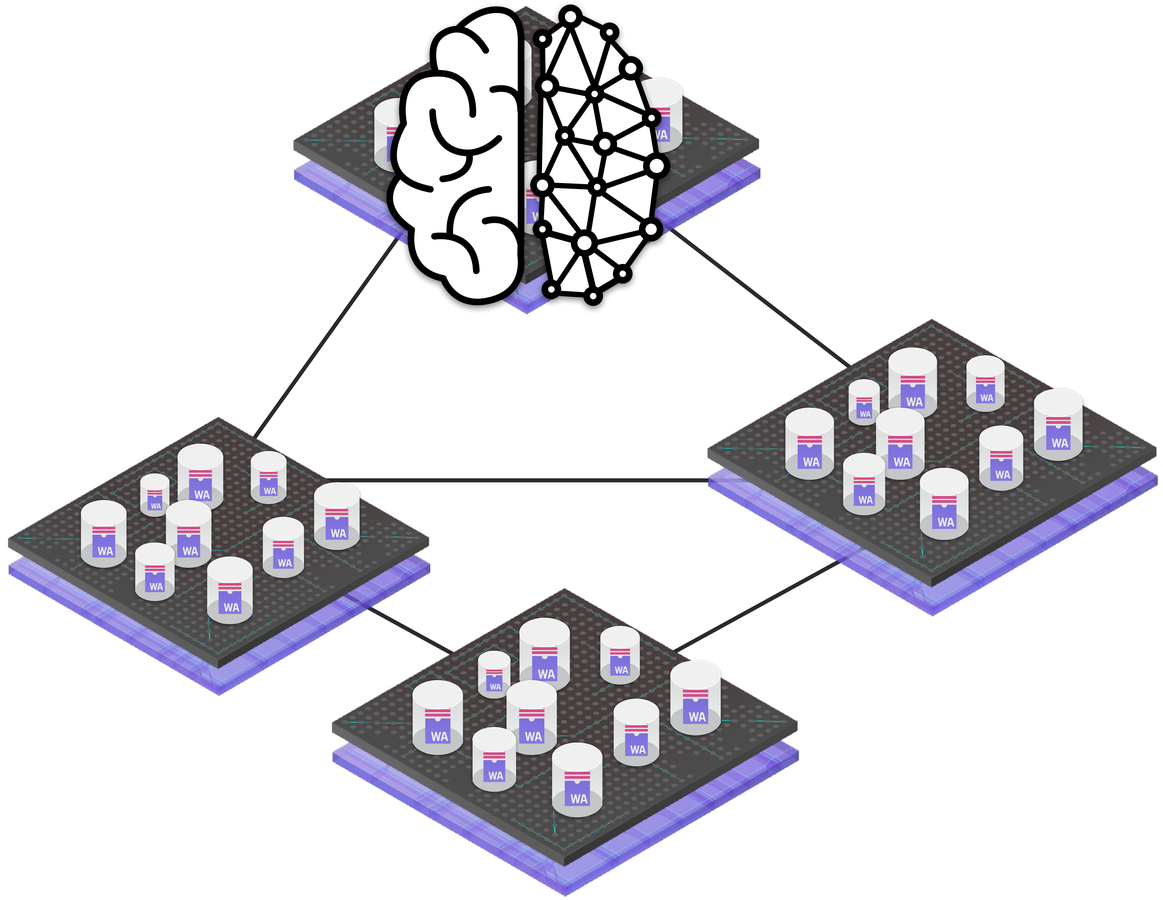
Transition function:
- message routing and scheduling
- canister code



1.6 Network Nervous System

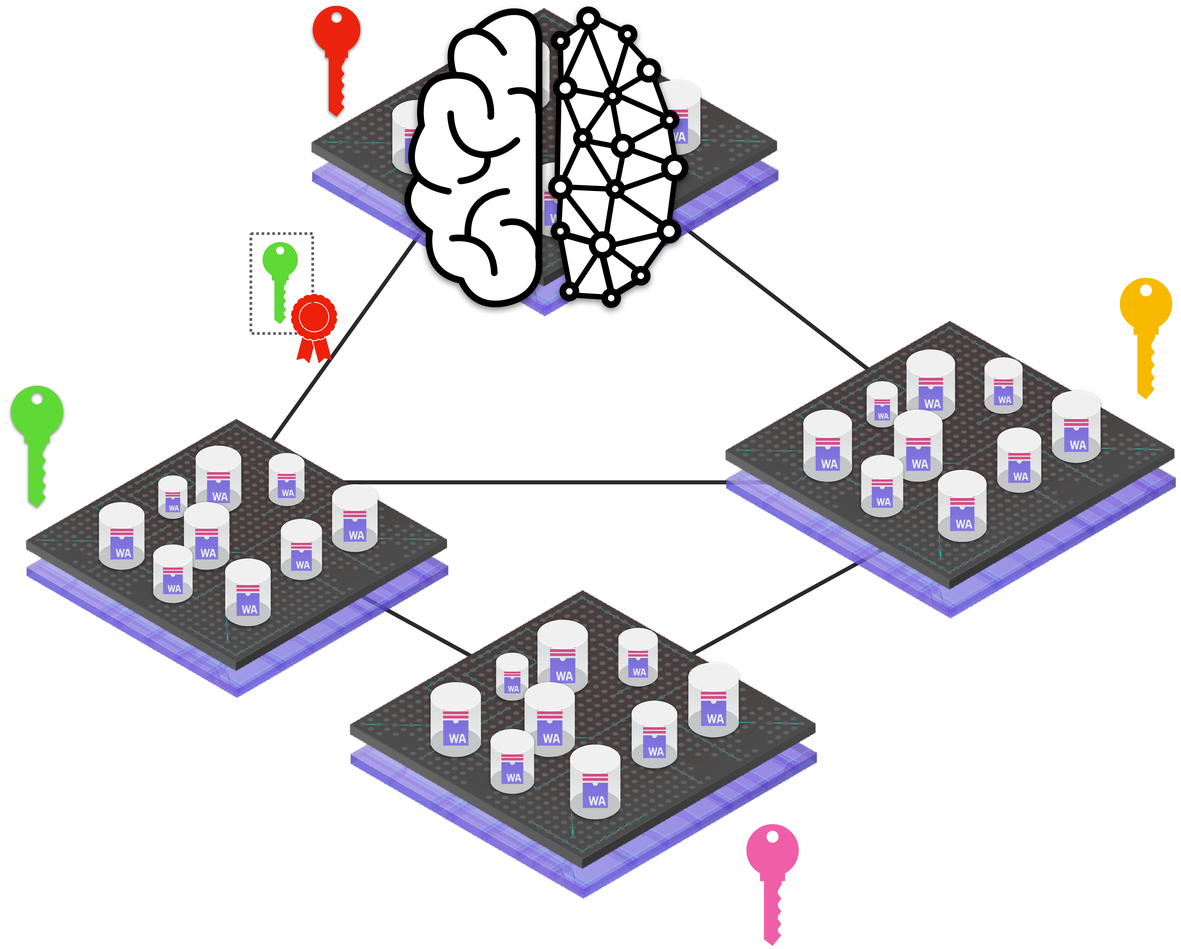
One subnet is special: it host the **Network Nervous System (NNS)** canisters which govern the IC

ICP token holders vote on
- Creation of new subnets
- Upgrades to new protocol version
- Replacement of nodes
- …

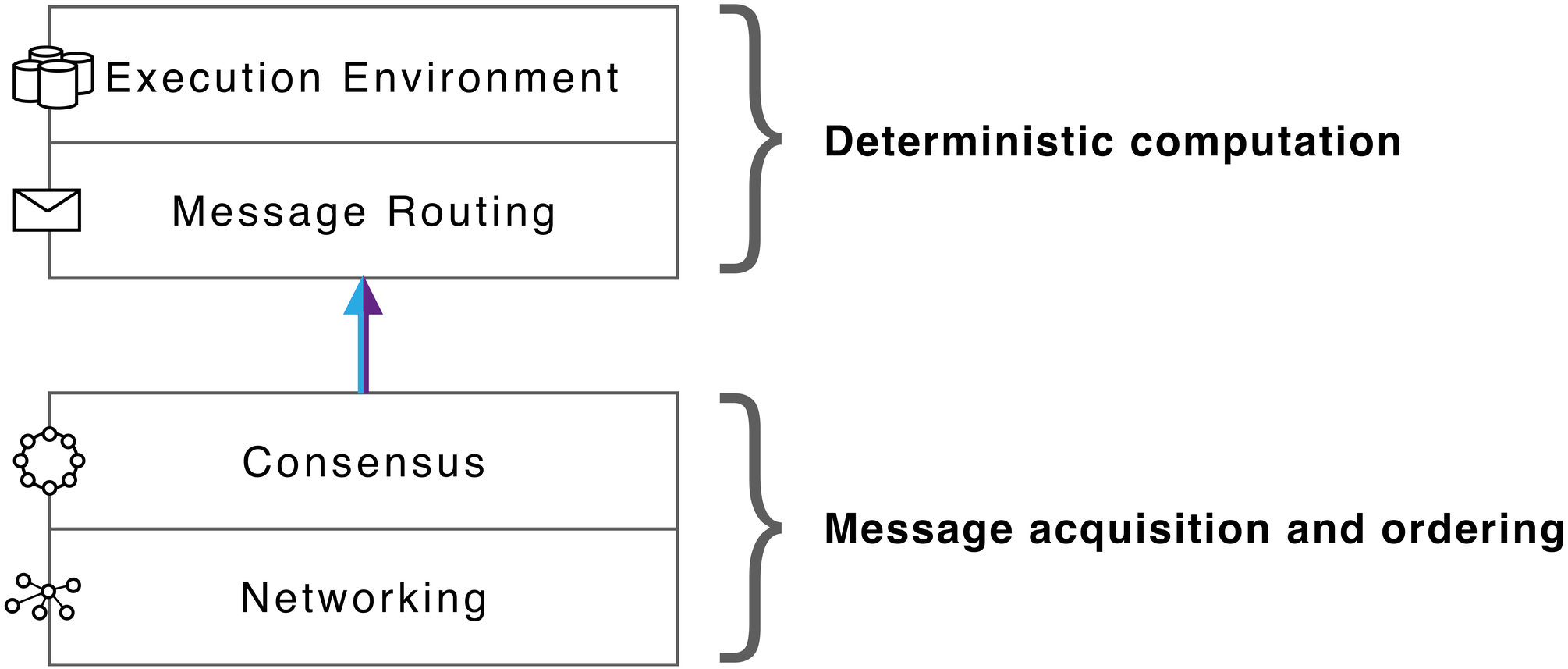


1.7 Chain Key Technology

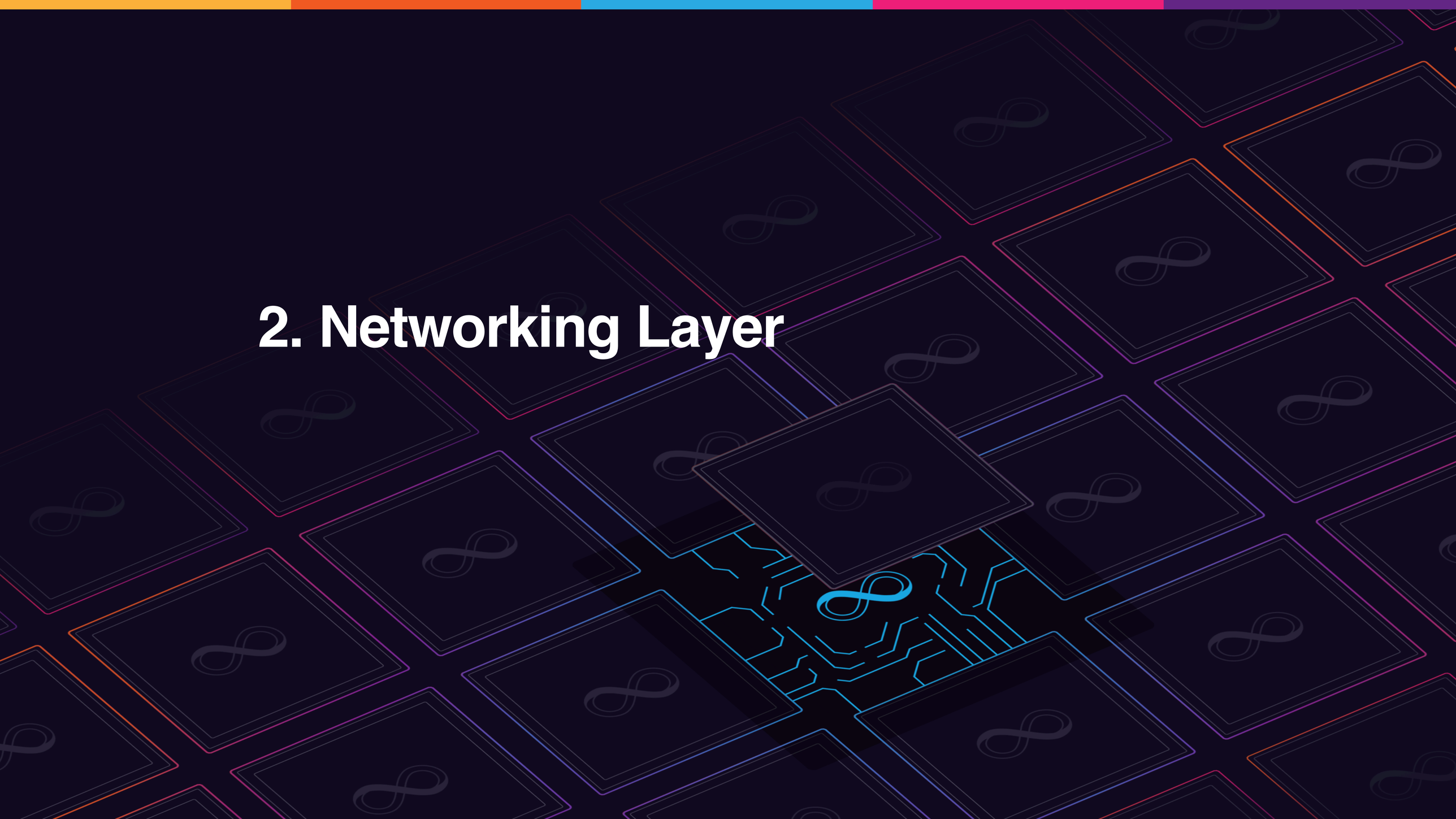
- Public key of NNS never changes, nodes in NNS share private key
- NNS generates key of subnets and certifies them
- Node in subnets use these keys to secure communication



1.8 The Layers of the Internet Computer Protocol



2. Networking Layer

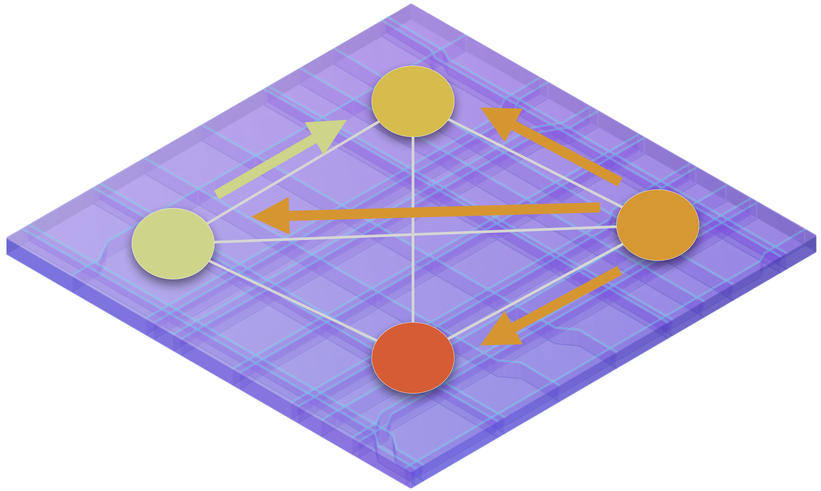


2.1 Communication Patterns

Messages must be sent from nodes to other nodes within the same subnet

Three communication patterns:

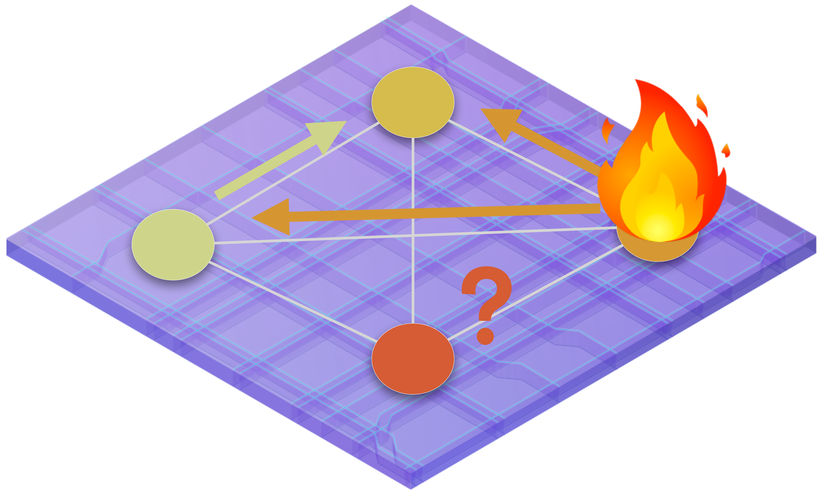
1) A node sends a message to another node
2) A node broadcasts a message to all other nodes
3) A node **reliably** broadcasts a message to all other nodes



2.2 Faulty/Malicious Behavior

It is possible that some message must arrive at all (honest) nodes

What if a node is faulty/malicious and fails to send a message to all other nodes?



2.3 Reliable Broadcast

Reliable broadcast for message m:

1. (Validity) If the sender is correct, then every correct node **delivers** m.

2. (Agreement) If two correct nodes **deliver** m and m', then m = m'.

3. (Integrity) Every correct node **delivers** at most one message m, and if it **delivers**, then some node must have invoked reliable broadcast for m.

4. (Totality) If a correct node **delivers** m, then all correct nodes eventually **deliver**.

Note: A variant of reliable broadcast was discussed in Chapter 18 (see Definition 18.10). The crucial difference is the integrity property!

2.4 Bracha’s Reliable Broadcast Algorithm

Sender S: **broadcast** m

upon receiving m from S **and not** sent_echo:
    broadcast (echo, m)
    sent_echo := true

if (received f+1 (ready, m) **or** 2f+1 (echo, m) messages) **and not** sent_ready:
    broadcast (ready, m)
    sent_ready := true

upon receiving 2f+1 (ready, m) messages:
    deliver m

2.5 Bracha’s Reliable Broadcast Algorithm - Analysis

Validity (If the sender is correct, then every correct node **delivers** m):

After S broadcast m, every correct node broadcasts (echo, m).
→ Every correct node eventually receives 2f+1 (echo, m) messages and sends (ready, m).
→ Every correct node eventually receives 2f+1 (ready, m) messages and delivers m.

Agreement (If two correct nodes **deliver** m and m’, then m = m’):

Assume two correct nodes v and v’ deliver different messages m and m’.
→ v and v’ must have received 2f+1 (ready, m) and (ready, m’) messages, respectively.
Since n=3f+1, there is a correct node that sent both (ready, m) and (ready, m’). This is a contradiction since every correct node only sends a ready message once.

2.6 Bracha’s Reliable Broadcast Algorithm - Analysis 2

Integrity (Every correct replica **delivers** at most one message m, and if it **delivers**, then some node must have invoked reliable broadcast for m):

Since every correct node only sends one ready message, there can only be one message for which 2f+1 ready messages are received.
→ A correct node delivers at most once.

The first correct node to send a ready message must receive 2f+1 echo messages.
→ There must be f+1 correct nodes that send echo messages.
→ Correct node only send echo messages when receiving a broadcast message.
→ A delivered message must have been broadcast.

2.8 Bracha’s Reliable Broadcast Algorithm - Analysis 3

Totality (If a correct node delivers m, then all correct nodes eventually deliver):

A correct node v delivers m
→ v received 2f+1 (ready, m) messages
→ v received $\geq$f+1 (ready, m) messages from correct nodes
→ all correct nodes eventually receive $\geq$f+1 (ready, m) messages
→ all correct nodes eventually broadcast (ready, m) messages
→ all correct nodes eventually receive $\geq$2f+1 (ready, m) messages
→ all correct nodes eventually deliver m.

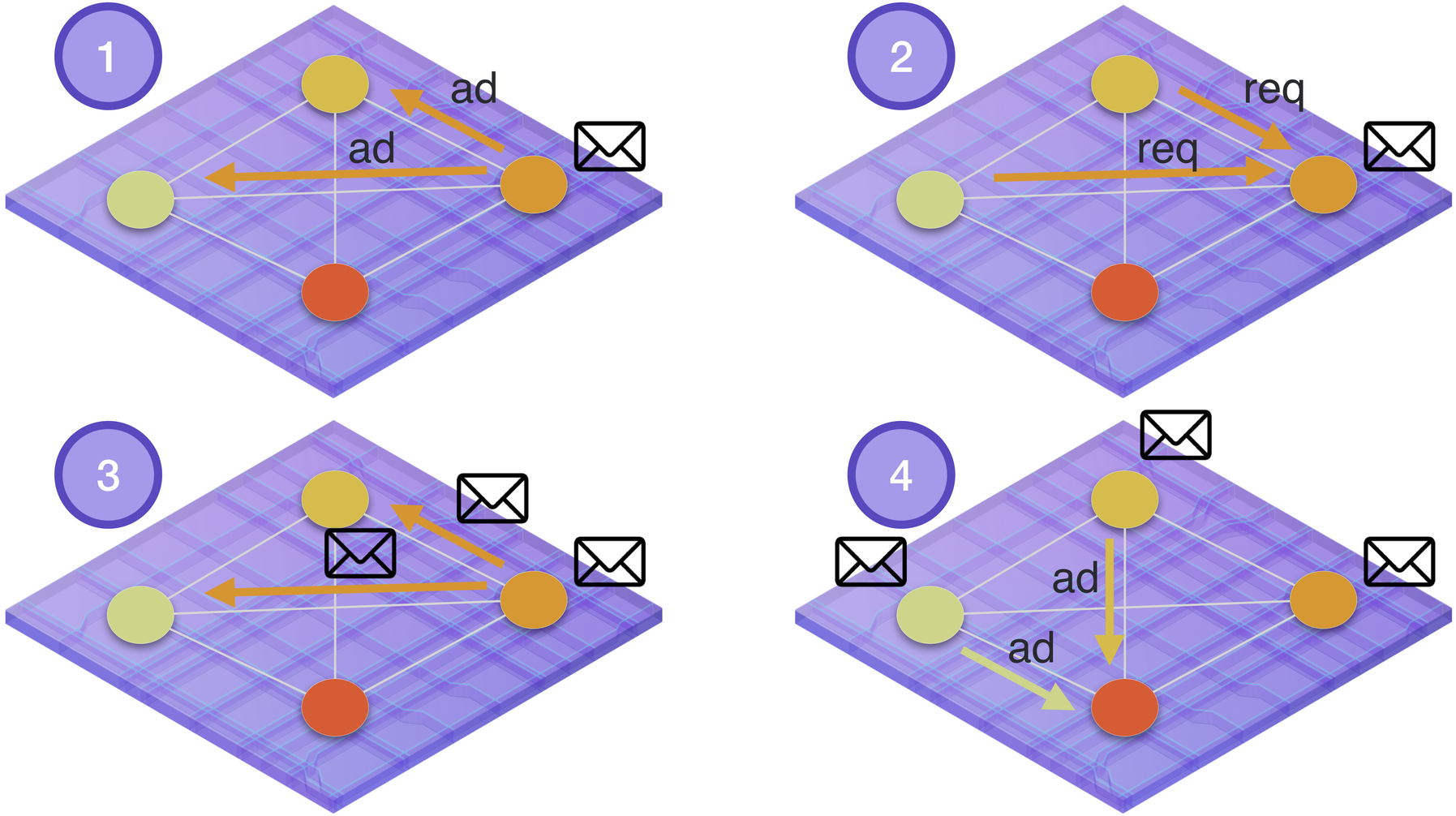
2.9 Bracha's Reliable Broadcast Algorithm - Analysis 4

Theorem: Bracha's algorithm is a reliable broadcast algorithm for $n > 3f$ in the asynchronous communication model.

Theorem: Bracha's algorithm has a communication complexity of $O(n^2L)$ for a message of L bits.
Proof: Every correct node broadcasts at most one (echo, m) and one (ready, m) message of size L each. The sender additionally broadcasts m.
$\rightarrow n \cdot n \cdot 2L + n \cdot L \in O(n^2L)$.

2.10 Advert-based Reliable Broadcast: Idea

If m is available locally, sent small “advert” to other nodes.
If m is missing locally, request m from a node that sent an advert for m!



2.11 Advert-based Reliable Broadcast: Algorithm

Sender S:

Execute Bracha's algorithm for advert := H(m)

Execute the steps when receiving m

Node v:

if received ad = advert from w **and not** delivered:

advertisers.add(w)

if received m and H(m) = advert **and not** delivered:

broadcast advert

deliver m

delivered := true

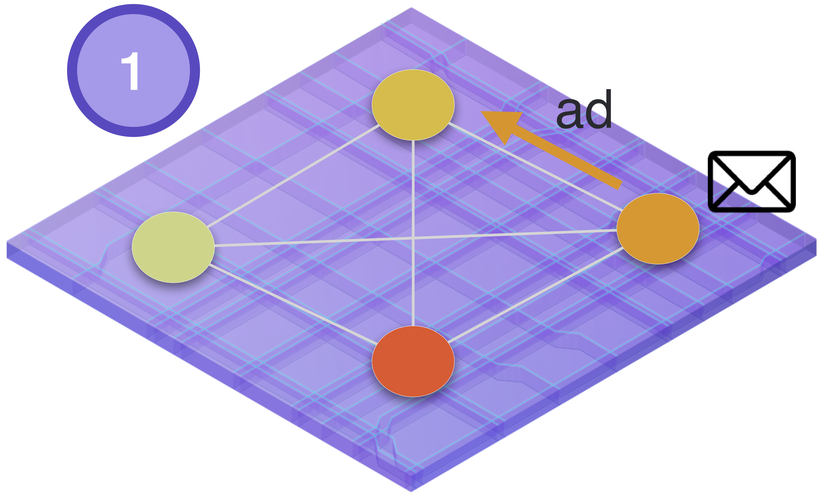
terminate after $(f+1)\Delta$ time

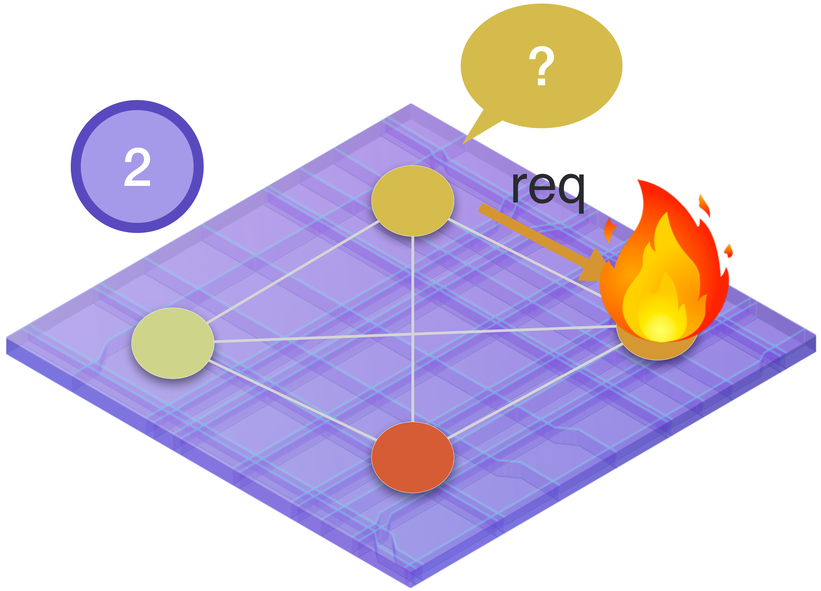
if received req(ad) from w $\notin$ senders **and** m $\neq \perp$:

send m to w

senders.add(w)

2.12 Advert-based Reliable Broadcast: Dealing with Failures

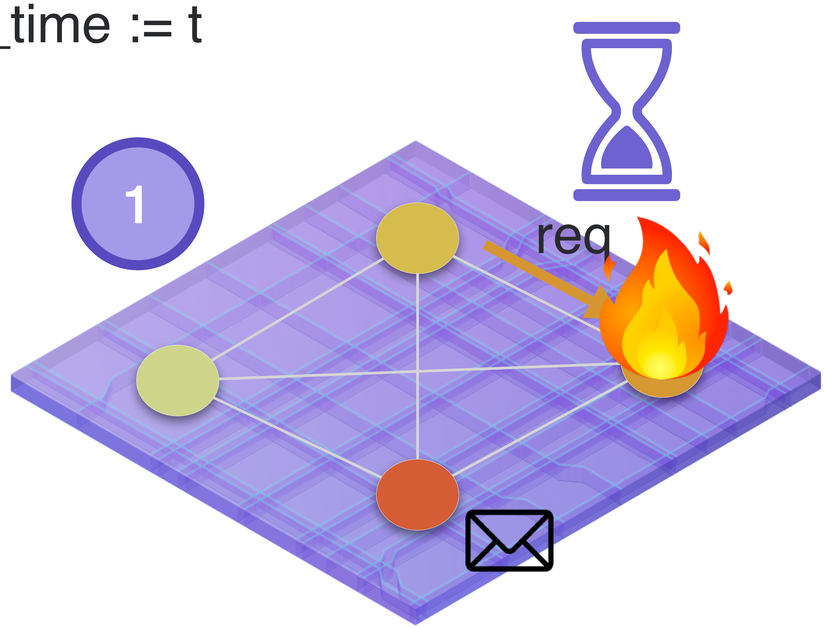


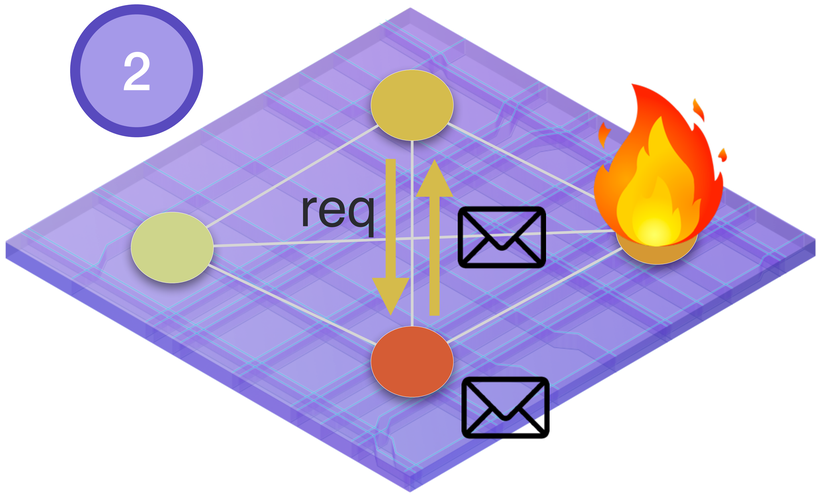


2.13 Advert-based Reliable Broadcast: Multiple Requests

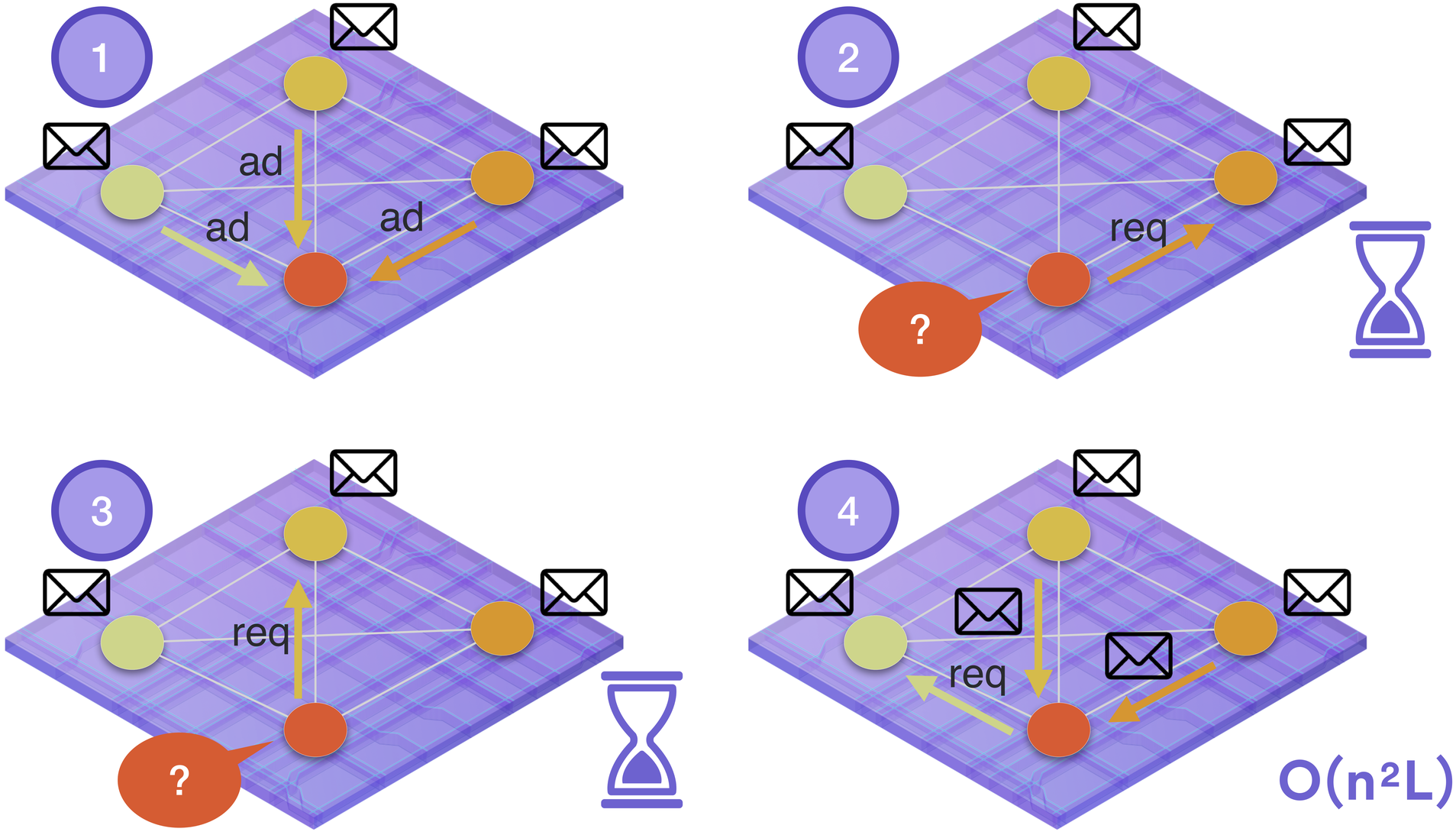
Initially, request_time = -∞, requested = {}

```
t := time()
if request_time + Δ ≤ t:
    w := choose_node(advertisers \ requested)
    send req(ad) to w
    requested.add(w)
    request_time := t
```



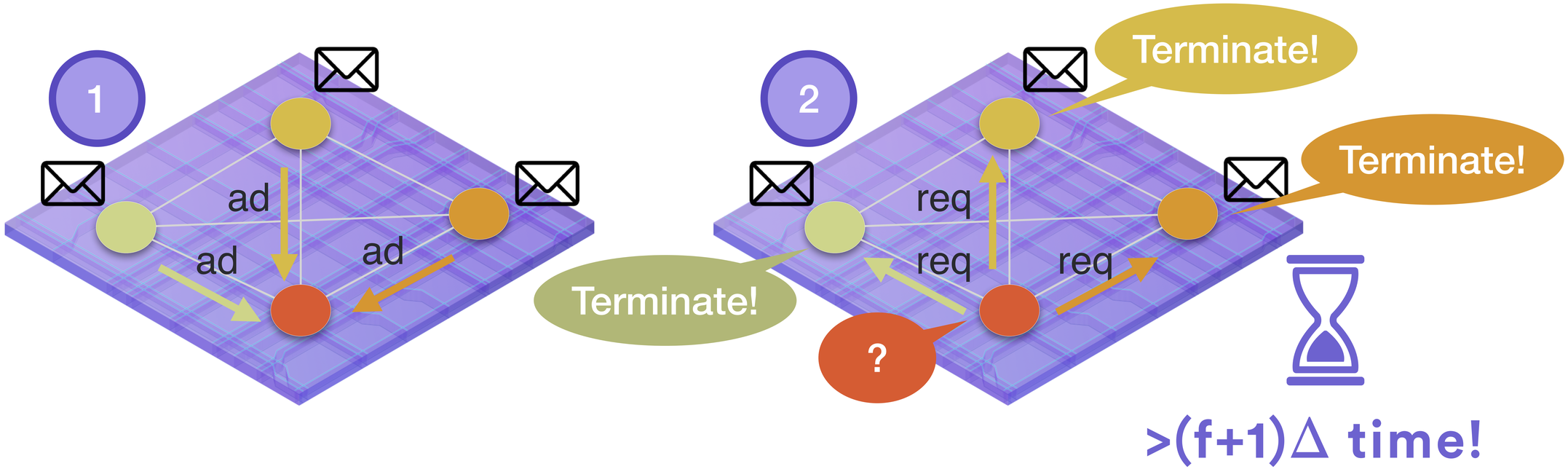


2.14 Advert-based Reliable Broadcast: Asynchrony?



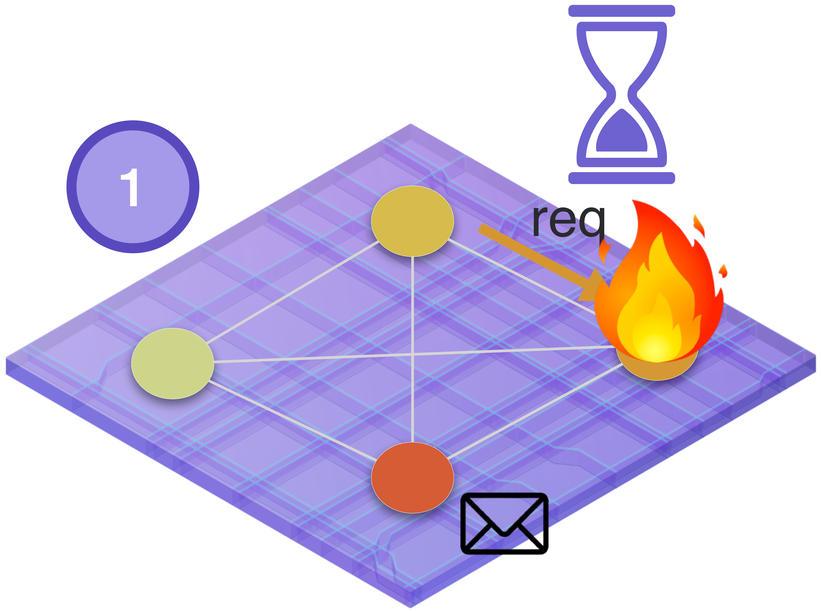
$O(n^2L)$ bits!

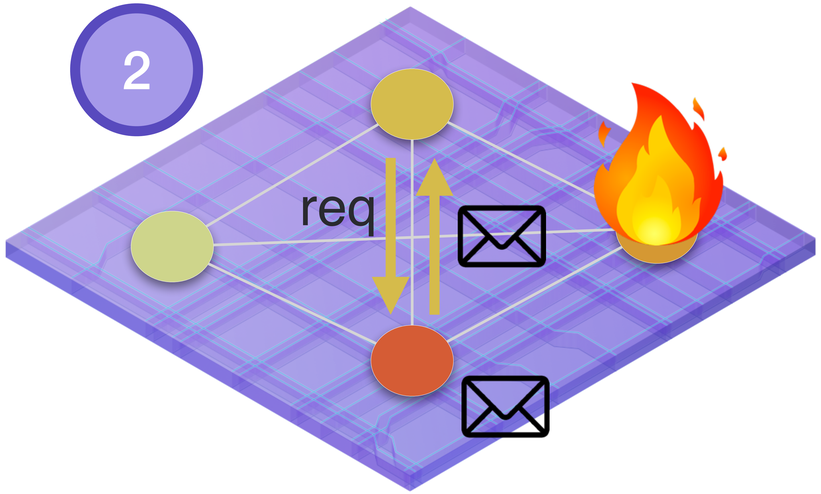
2.14 Advert-based Reliable Broadcast: Asynchrony?!?



2.15 Advert-based Reliable Broadcast: Synchrony

Theorem: For Δ:=2, the advert-based algorithm implements reliable broadcast in the synchronous communication model.
Proof (Totality): A correct node v delivers m at time t → v broadcasts ad := H(m)
→ Every correct node receives ad by time t+1 and requests the message
→ In the worst case, a correct node sends the request to f faulty nodes, which takes Δf time
→ Node v or another correct node receives the request by time t+2+Δf = t+Δ(f+1) time and returns m.





2.16 Advert-based Reliable Broadcast: Synchrony 2

Theorem: The advert-based algorithm has a communication complexity of $O(n^2H + (f+1)nL)$ for a message of L bits and an advert size of H bits.

Proof:

Executing Bracha's algorithm requires sending $O(n^2H)$ bits.

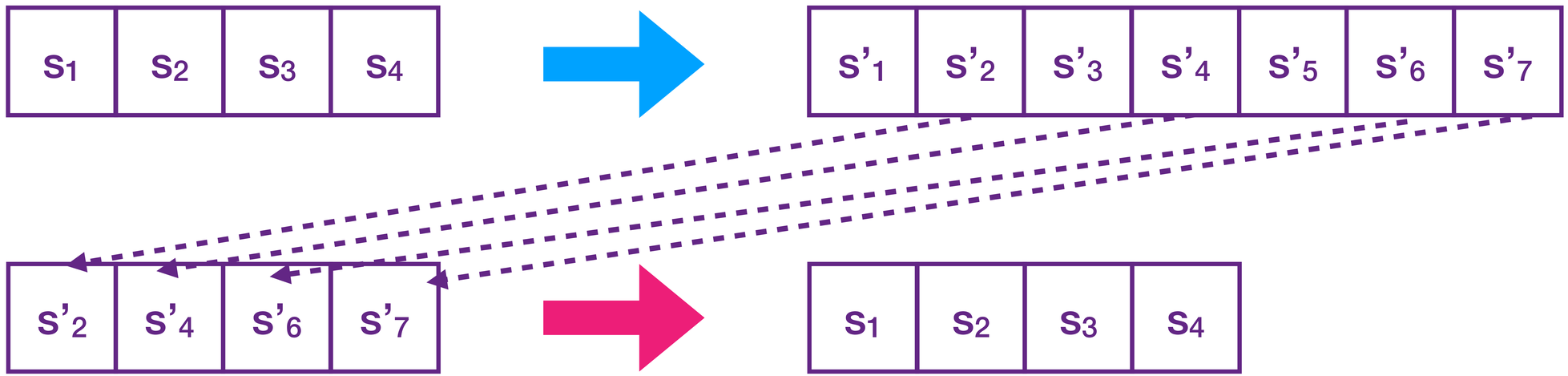
Adverts are broadcast once and every request is sent to every other node at most once $\rightarrow$ $O(n^2H)$ bits.

Every correct node receives the message from one other node $\rightarrow$ O(nL) bits.

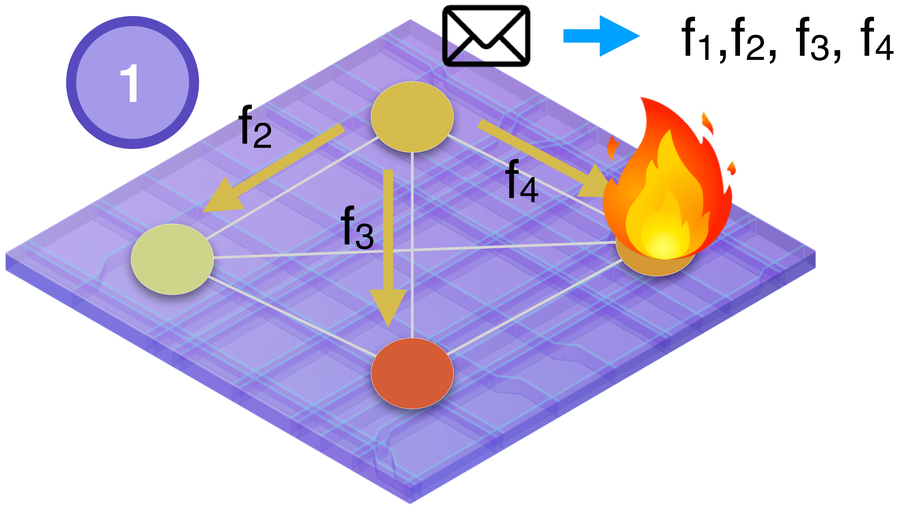
However, all faulty nodes can request m from every correct node $\rightarrow$ O(fnL) bits.

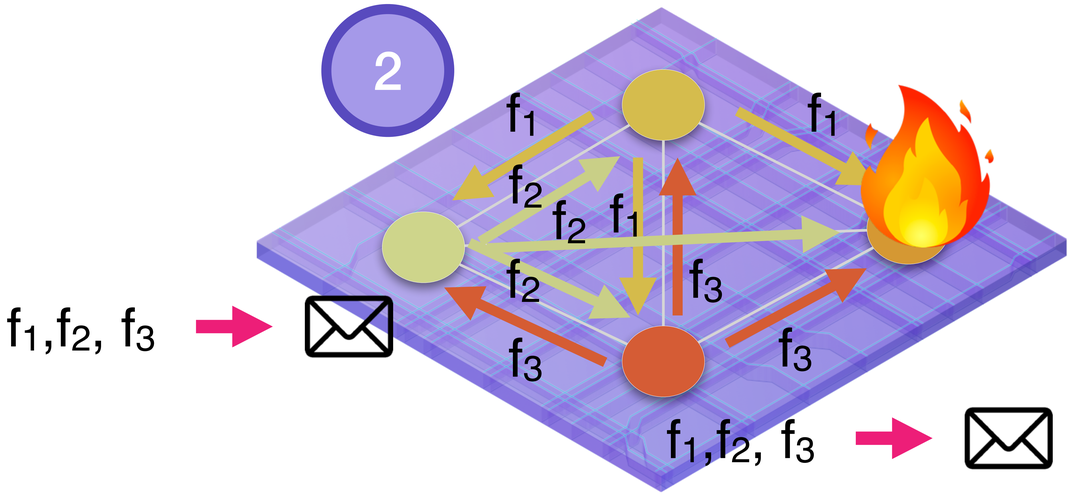
2.17 Coding-based Reliable Broadcast: Erasure Codes

Erasure code: A (n,k)-erasure code is a code that transforms a message m of k symbols into a message with n > k symbols such that m can be recovered from a subset of the n symbols.



2.18 Coding-based Reliable Broadcast: Algorithm (Idea)





Size of m: L bits
Size of f_i: L/(f+1)≈ 3L/n bits

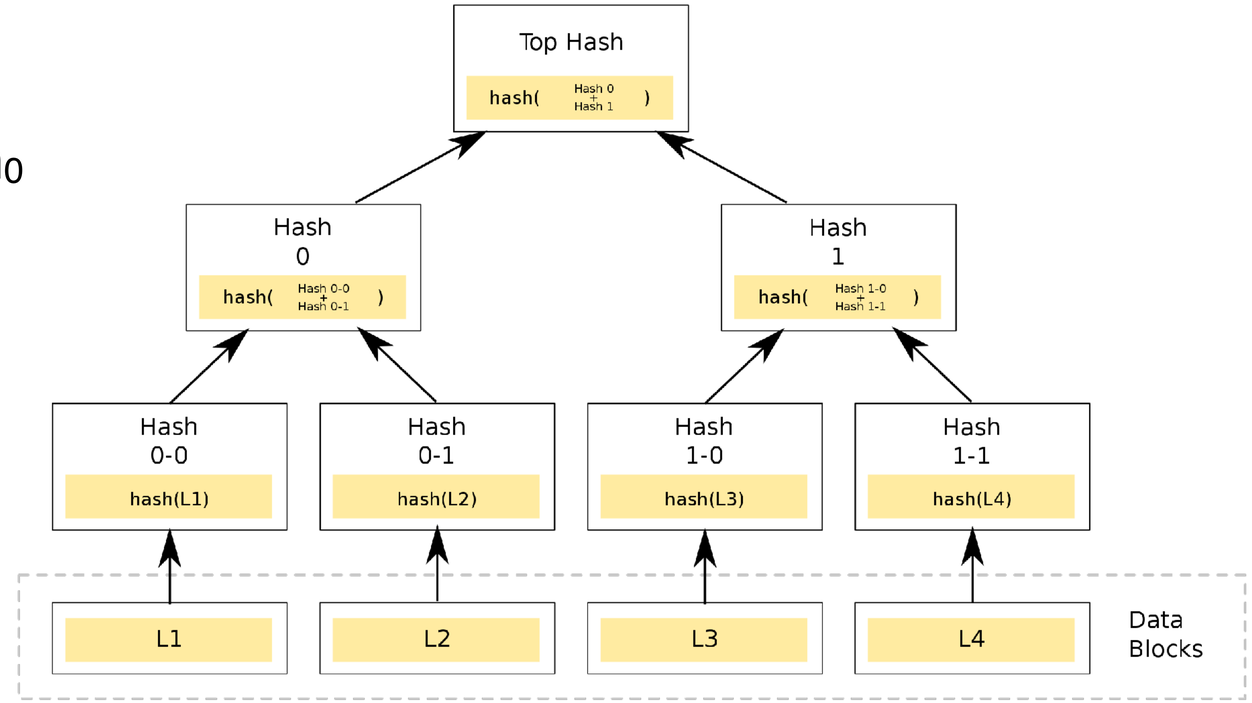
2.19 Coding-based Reliable Broadcast: Algorithm

Sender S:

$(f_1, \ldots, f_n)$:= get_fragments(m)
h_0 := get_merkle_root_hash($f_1, \ldots, f_n$)
Execute Bracha's algorithm for root_hash := h_0
for j ∈ {1, …, n}:
    P_j := get_merkle_proof($f_1, \ldots, f_n$, j)
    send (f_j, P_j, j) to v_j

Node v_i:

if received (f, P, j) **and** root_hash $\neq \perp$:
    if valid(f, P, root_hash, j):
        F := F ∪ {f}
        if i = j **and not** broadcast f before:
            broadcast (f, P, i)



Merkle tree (source: Wikipedia)

2.20 Coding-based Reliable Broadcast: Algorithm 2

Node v_i:

if |F| = f+1 **and** m = ⊥:
    m := recover_message(F)
    (f_1,…, f_n) := get_fragments(m)
    h_0 := get_merkle_root_hash(f_1,…, f_n)
    if h_0 = root_hash:
        if **not** broadcast f_i before:
            P_i := get_merkle_proof(f_1,…, f_n, i)
            broadcast (f_i, P_i, i)
    else:
        root_hash := ⊥

if |F| = 2f+1 **and** root_hash ≠ ⊥ **and not** delivered:
    deliver m

2.21 Coding-based Reliable Broadcast: Analysis

Theorem: The coding-based algorithm is a reliable broadcast algorithm for $n > 3f$ in the asynchronous communication model.

Proof (Totality): A correct node v delivers m.

→ Since $|F| = 2f+1$, v must have received fragments from $\geq f+1$ correct nodes.

→ All correct nodes must eventually receive $\geq f+1$ fragments.

→ All correct nodes eventually reconstruct m and broadcast their fragments.

→ All correct nodes eventually receive $\geq 2f+1$ fragments and deliver m.

Theorem: The coding-based algorithm has a communication complexity of $O(n^2\log(n)H + nL)$ in the asynchronous communication model.

Proof: The initial Bracha broadcast requires $O(n^2H)$ bits.

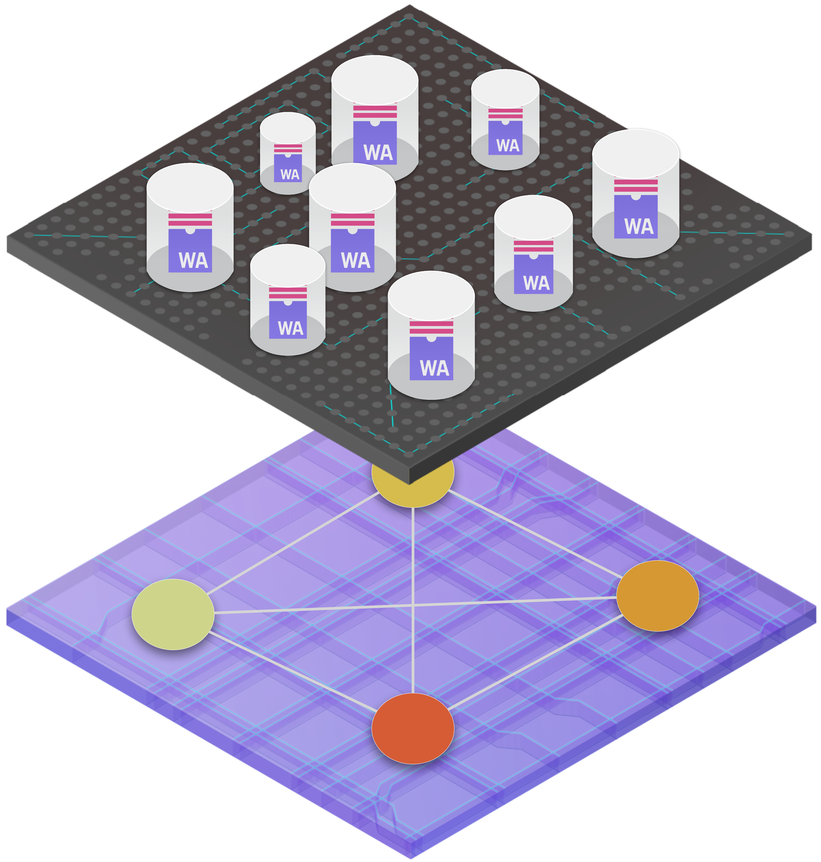
A Merkle proof has size $O(\log(n)H)$. Each node broadcasts its fragment with a Merkle proof

→ $O(n \cdot n \cdot (\log(n)H) + O(n \cdot n \cdot L/n) = O(n^2\log(n)H + nL)$.

Note: This bound is not far from the lower bound $\Omega(n^2 + nL)$!

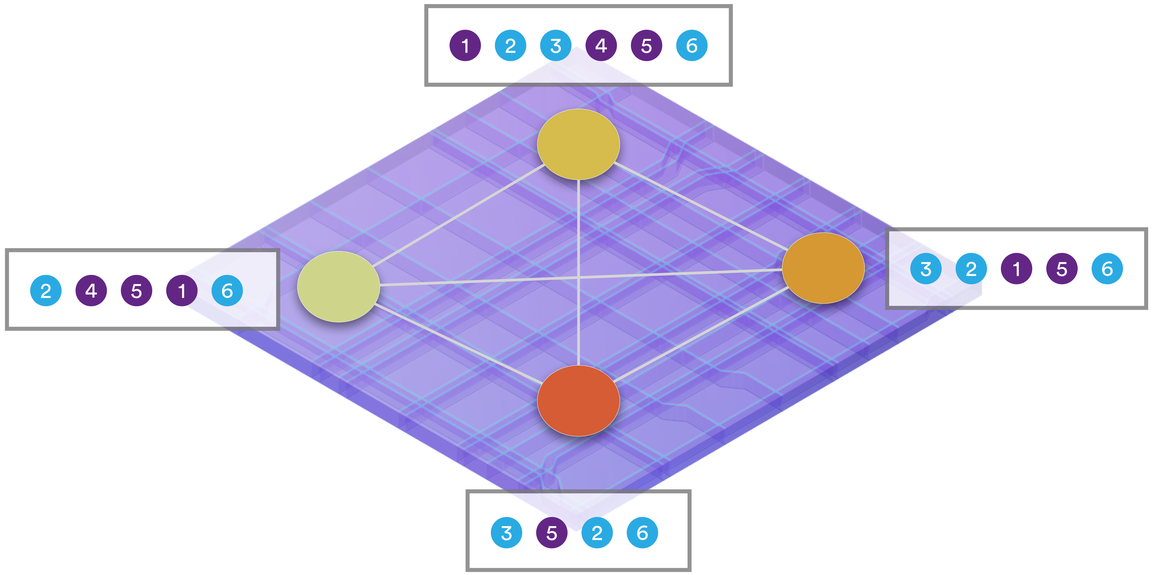
3. Consensus Layer

3.1 Consensus Orders Messages



3.2 Consensus Orders Messages

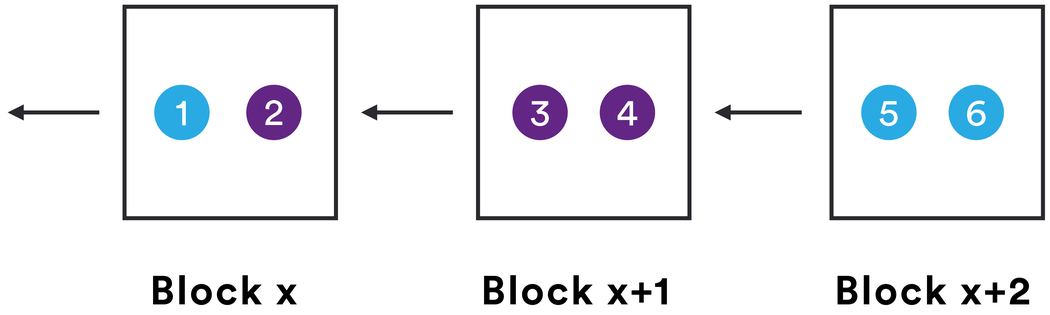
● Message (user → canister) ● Message (canister → canister)



Nodes may receive input messages in different orders, but must process them in the same order, for example: 1 2 3 4 5 6

3.3 Consensus Properties

Messages are placed in **blocks**. We reach agreement using a blockchain.

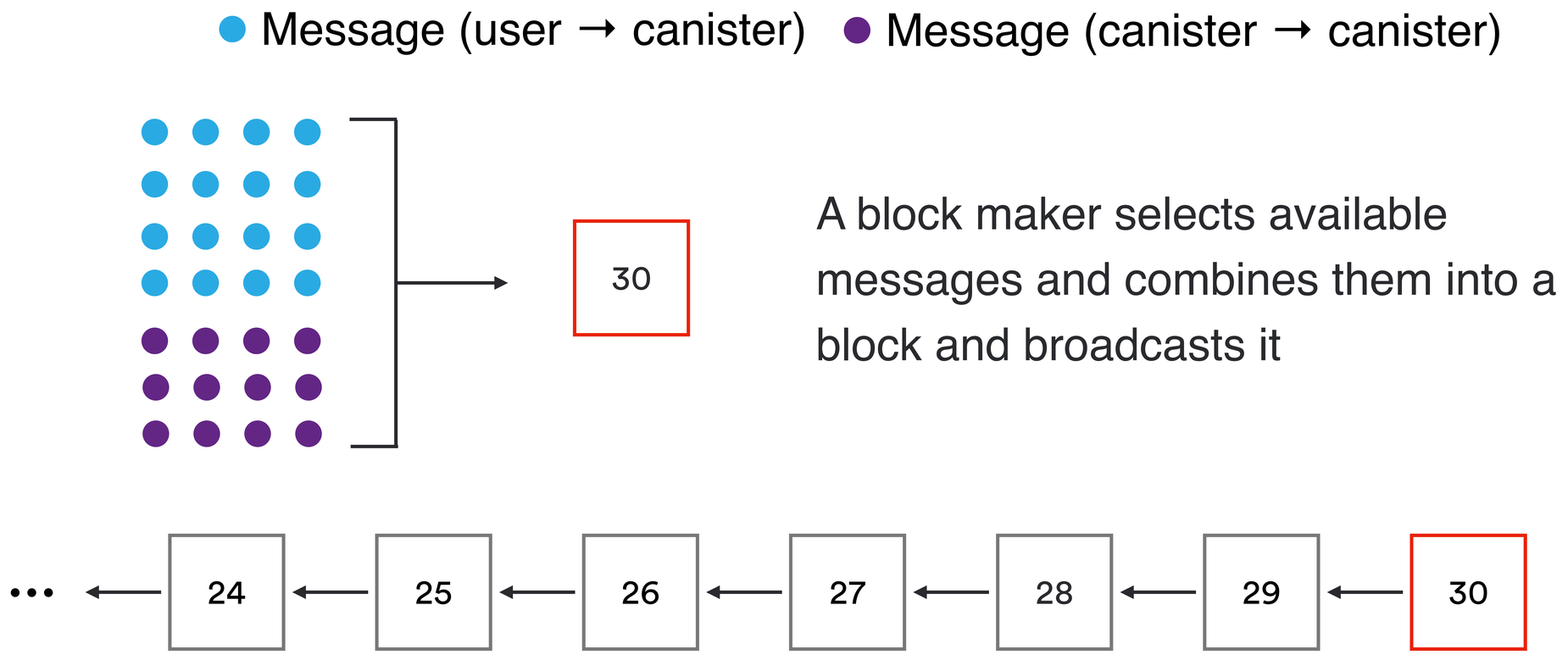


We use $n = 4, f = 1$ in examples

The following properties must hold even if up to $f < n/3$ nodes misbehave

- **Agreement**: For any i, If two (honest) nodes think that the *i*-th block is agreed upon, they must have the same block
- **Termination**: For any i, at some point every (honest) node will think that the *i*-th block is agreed upon
- **Validity**: all agreed upon blocks are valid

3.4 Block Maker



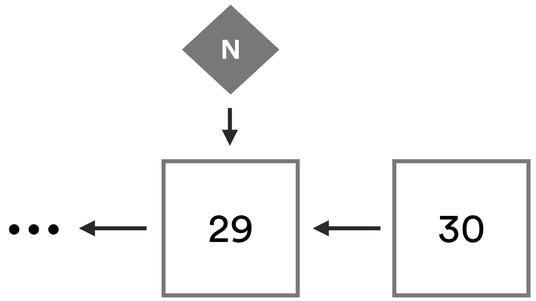
Note: We need more than one block maker in each round, otherwise the IC would not be fault tolerant!

3.5 Notarization

The notarization process ensures that a *valid* block proposal is published for every block height

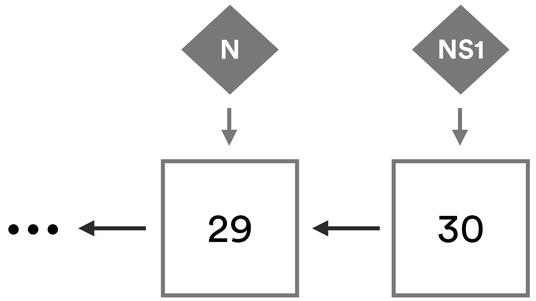
Step 1

Replica 1 receives a block proposal for height 30, building on some notarized height 29 block



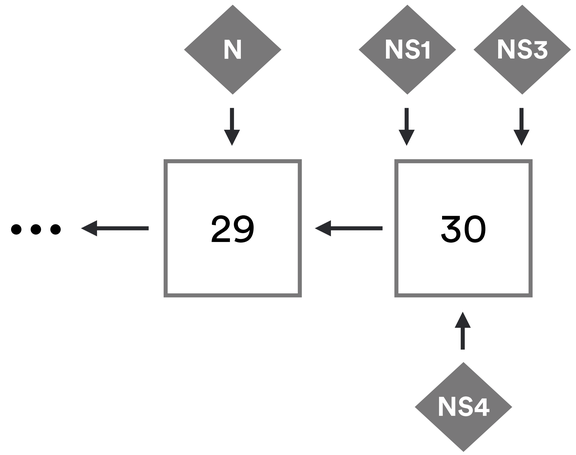
Step 2

Replica 1 sees that the block is valid, signs it, and broadcasts its *notarization* share



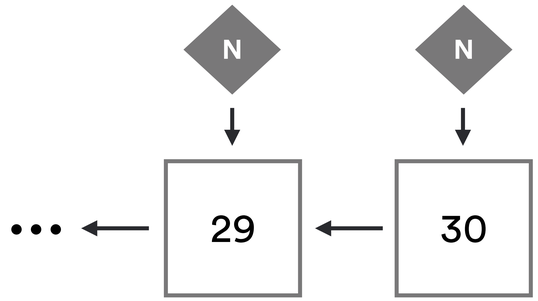
Step 3

Replica 1 sees that replicas 3 and 4 also published their notarization shares on the block



Step 4

3 notarization shares are sufficient approval: the shares are aggregated into a single full notarization. Block 30 is now notarized, and notaries wait for height-31 blocks

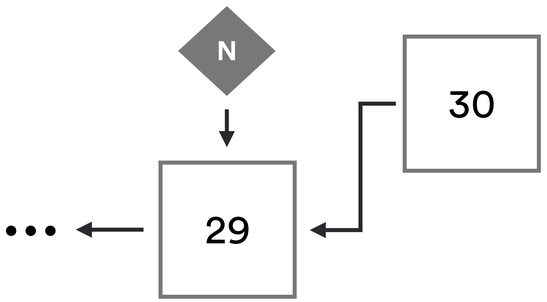


3.6 Notarization

Replicas may notary-sign multiple blocks to ensure that at least one block becomes fully notarized

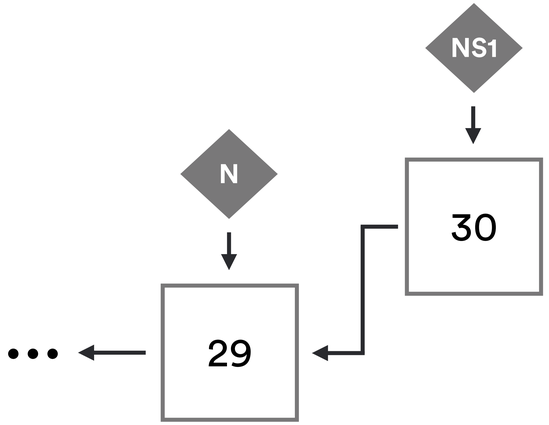
Step 1

Replica 1 receives a block proposal for height 30, building on some notarized height 29 block



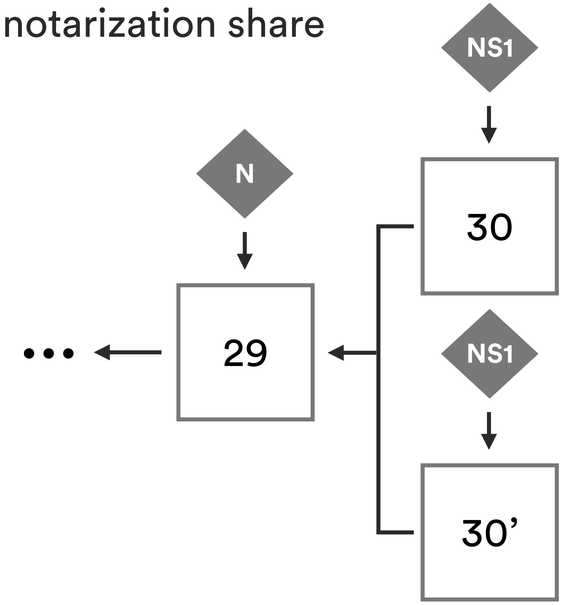
Step 2

Replica 1 sees that the block is valid, signs it, and broadcasts its *notarization* share



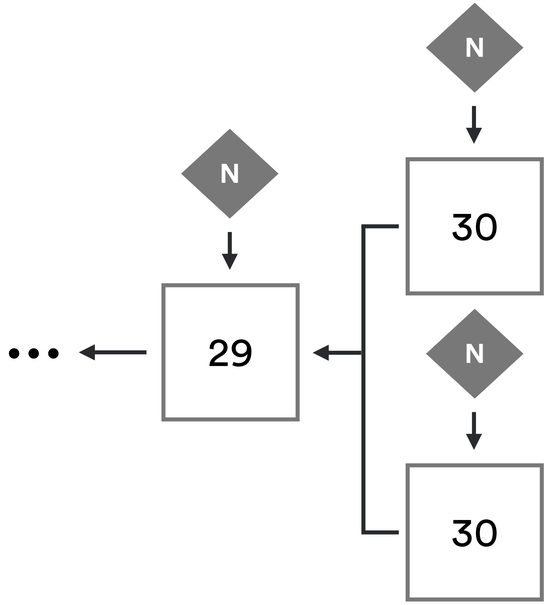
Step 3

Replicas 1 sees another height 30 block, which is also valid, and it broadcasts another notarization share



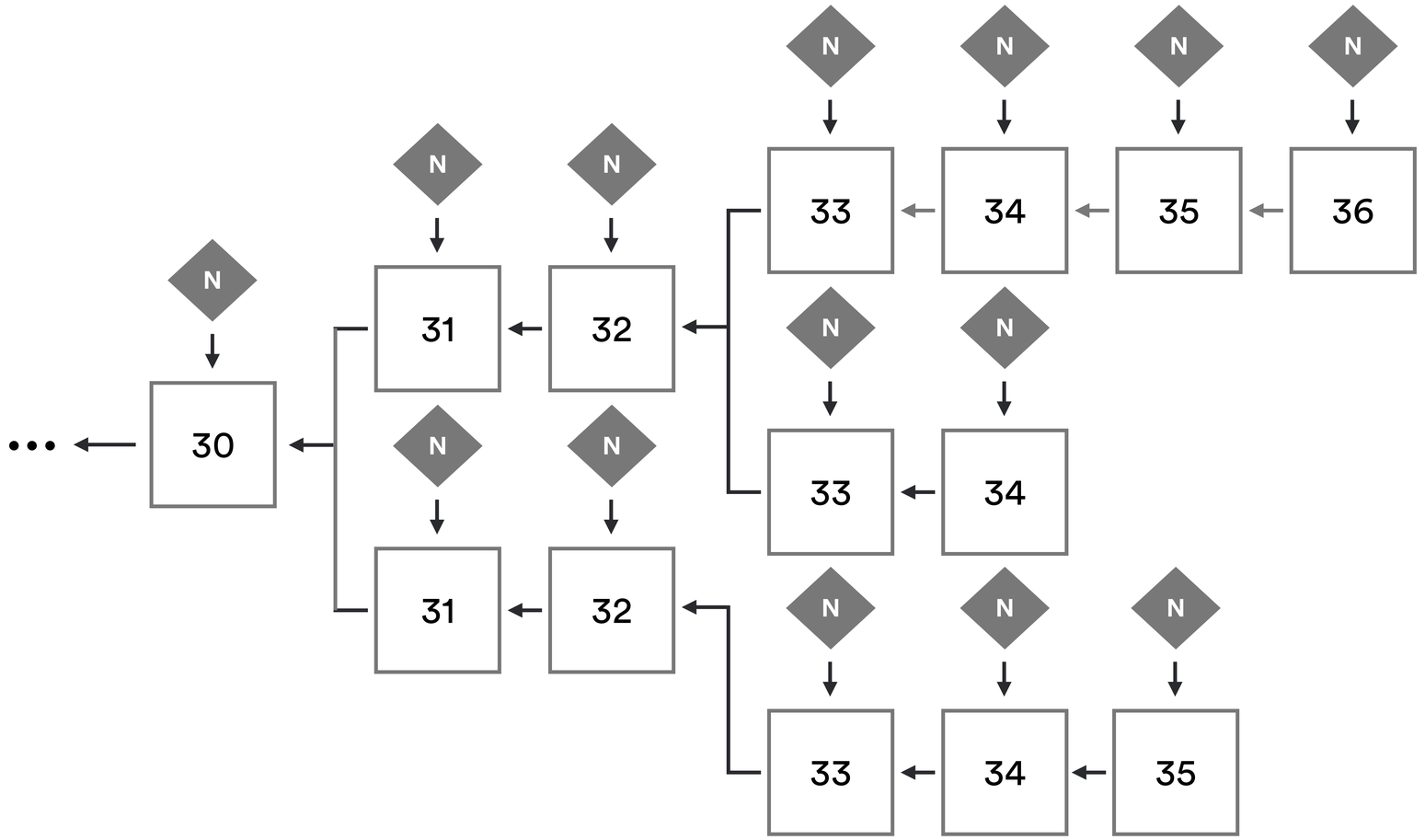
Step 4

Both height 30 blocks get enough support to become notarized



3.7 Notarization

Multiple notarized blocks may exist at the same block height



3.8 Random Beacon

At every height, there is a Random Beacon, an unpredictable random value shared by the replicas

Step 1

Replica 1 has Random Beacon 29 and wants to help constructing Random Beacon 30

Step 2

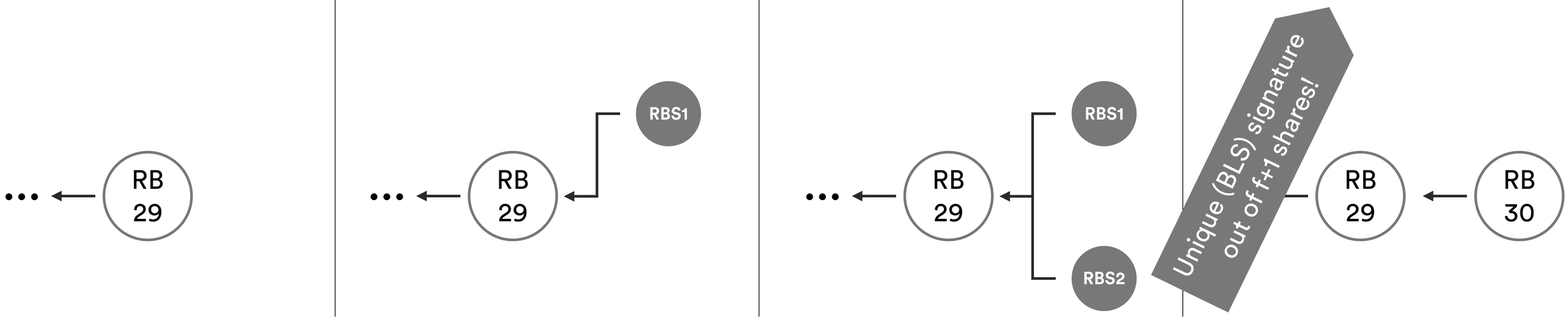
Replica 1 signs RB29 using a threshold signature scheme, yielding a share of random beacon 30

Step 3

Replicas 1 sees that replica 2 also published a share of Random Beacon 30

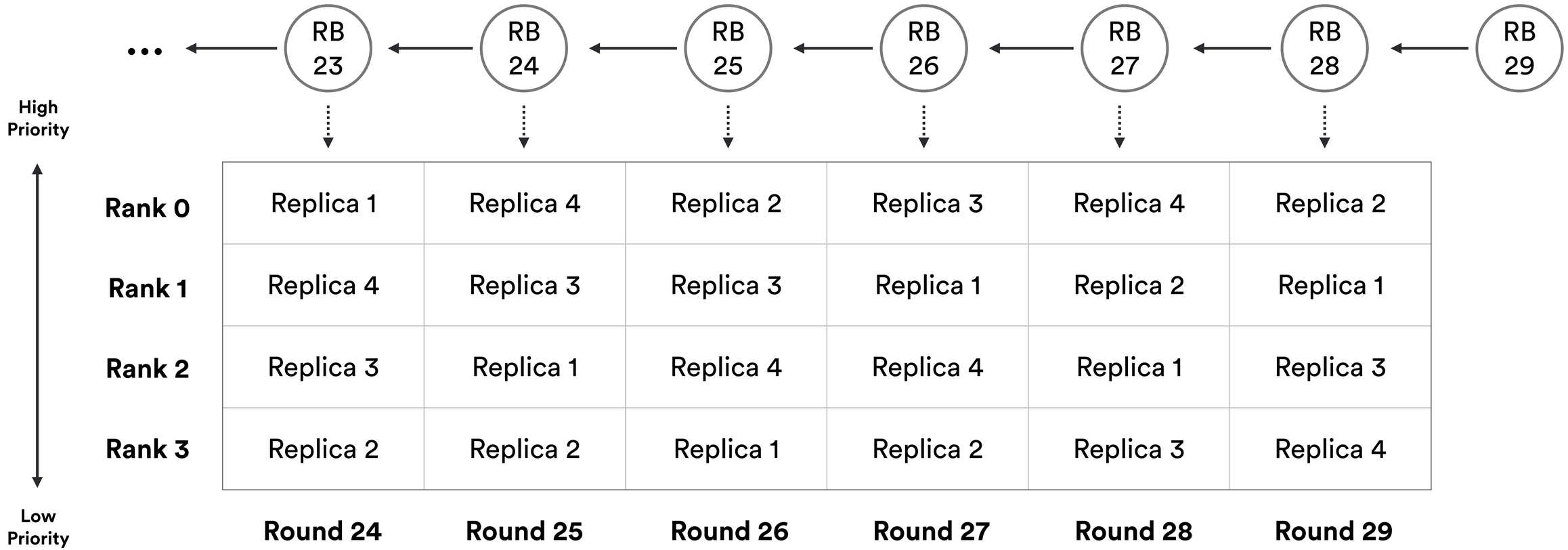
Step 4

2 random beacon shares are sufficient to reconstruct a full threshold signature, which is Random Beacon 30



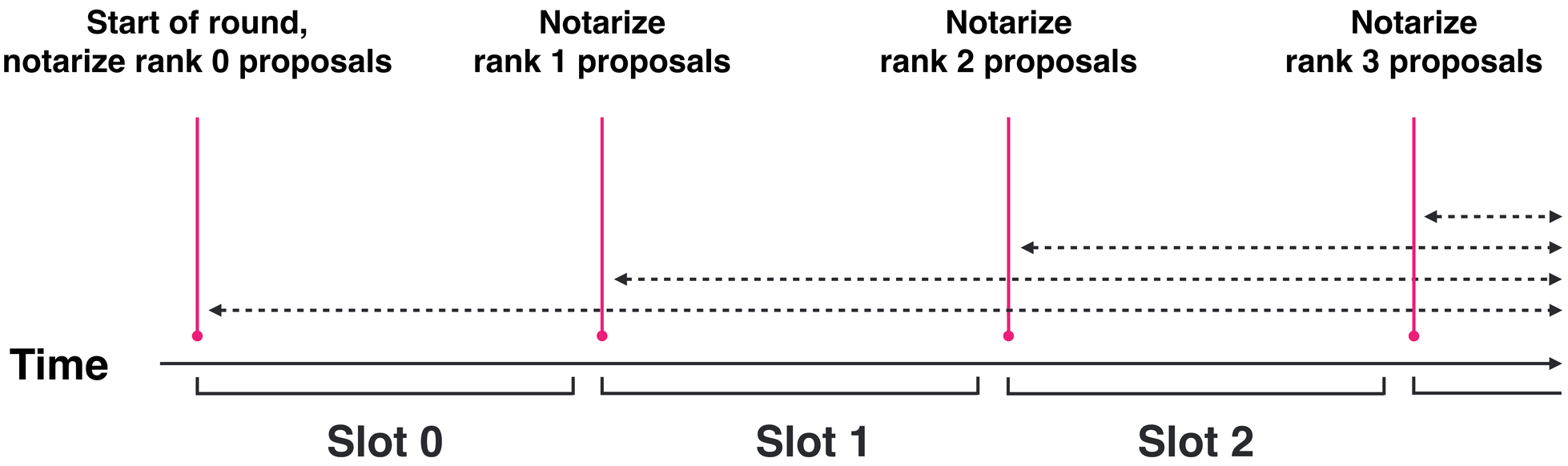
3.9 Block Maker Ranking

The Random Beacon is used to rank block makers



3.10 Notarization with Block Maker Ranking

Rounds are divided into time slots defining when block maker proposals are considered

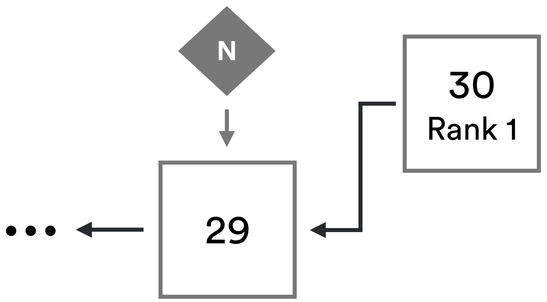


3.11 Notarization with Block Maker Ranking

The block ranks can reduce the number of notarized blocks

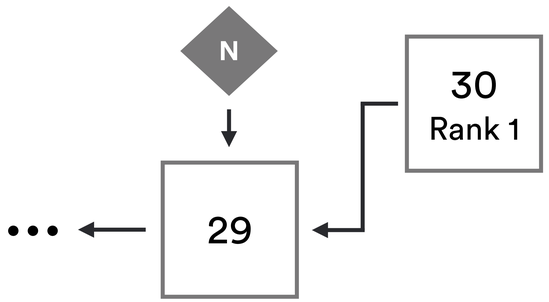
Step 1

Replica 1 receives a rank-1 block proposal for height 30, building on some notarized height 29 block



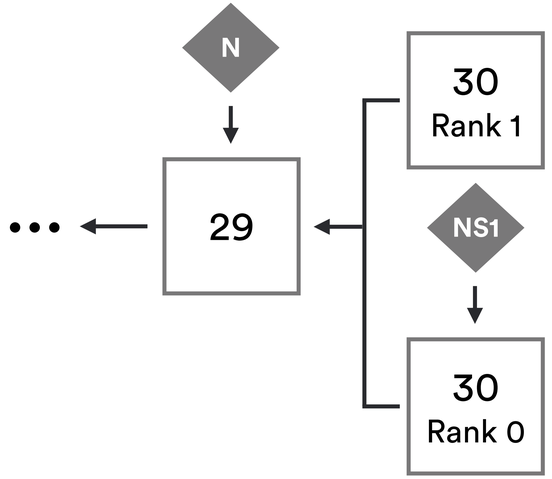
Step 2

Replica 1 is still in time slot 0, so not willing to notary-sign a rank-1 block yet



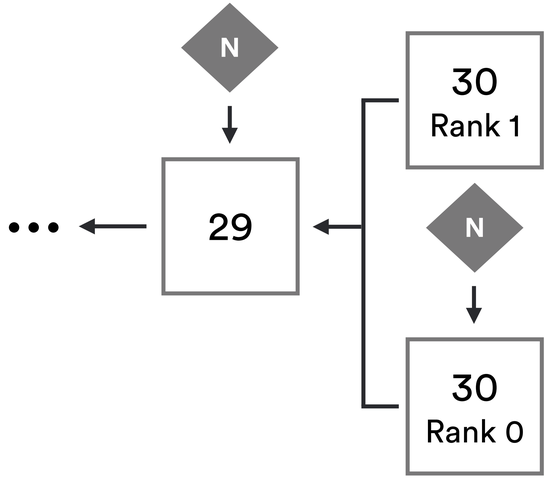
Step 3

Replicas 1 sees a valid rank-0 height 30 block, and it broadcasts a notarization share



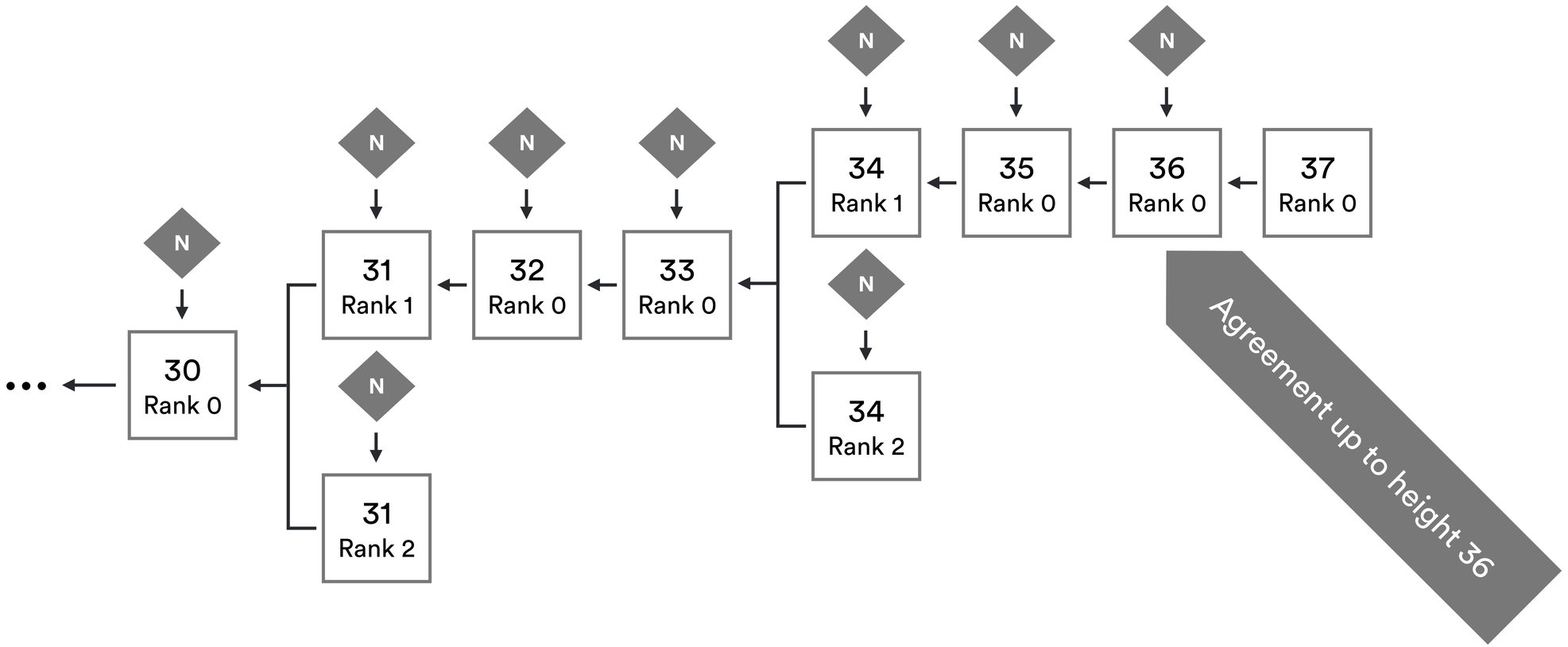
Step 4

Eventually, only the rank 0 block becomes notarized



3.12 Notarization with Block Maker Ranking

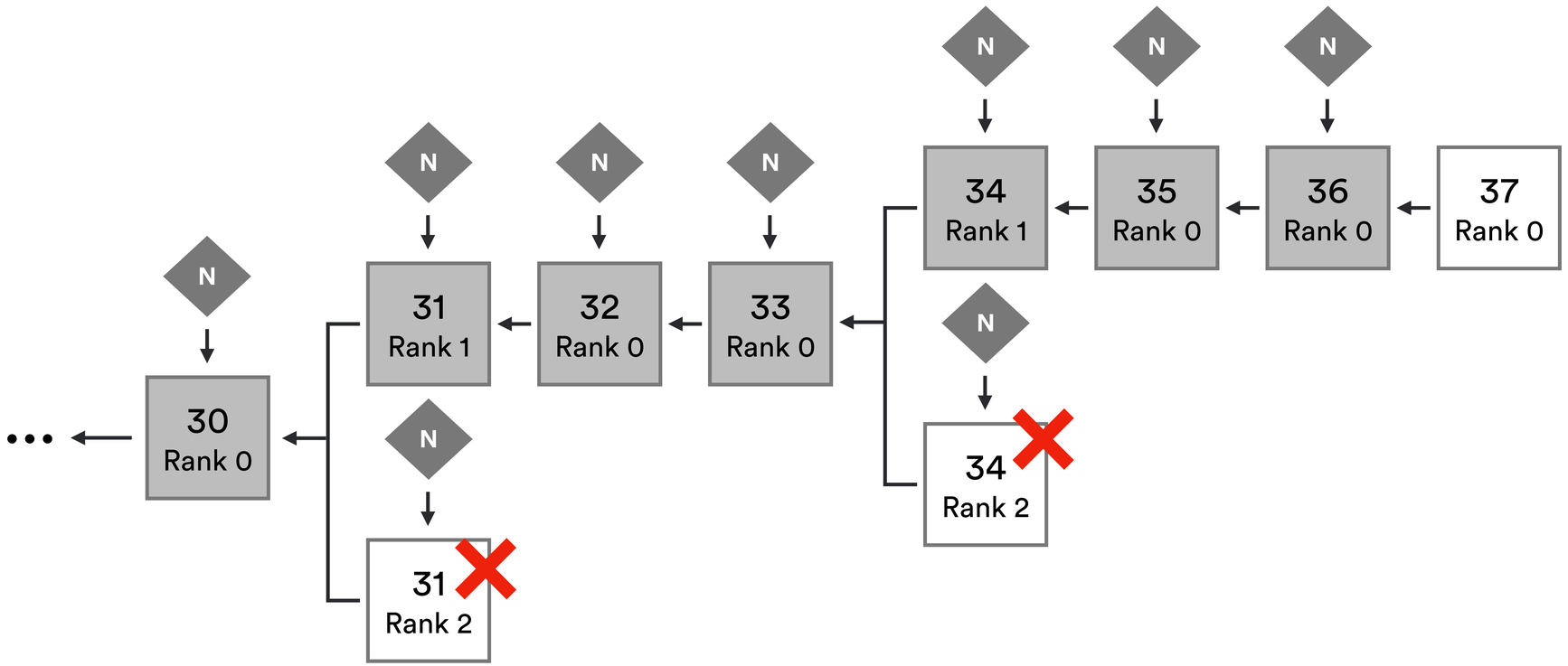
One notarized block *b* at a height *h* = Agreement up to *h*



How can we detect this…?

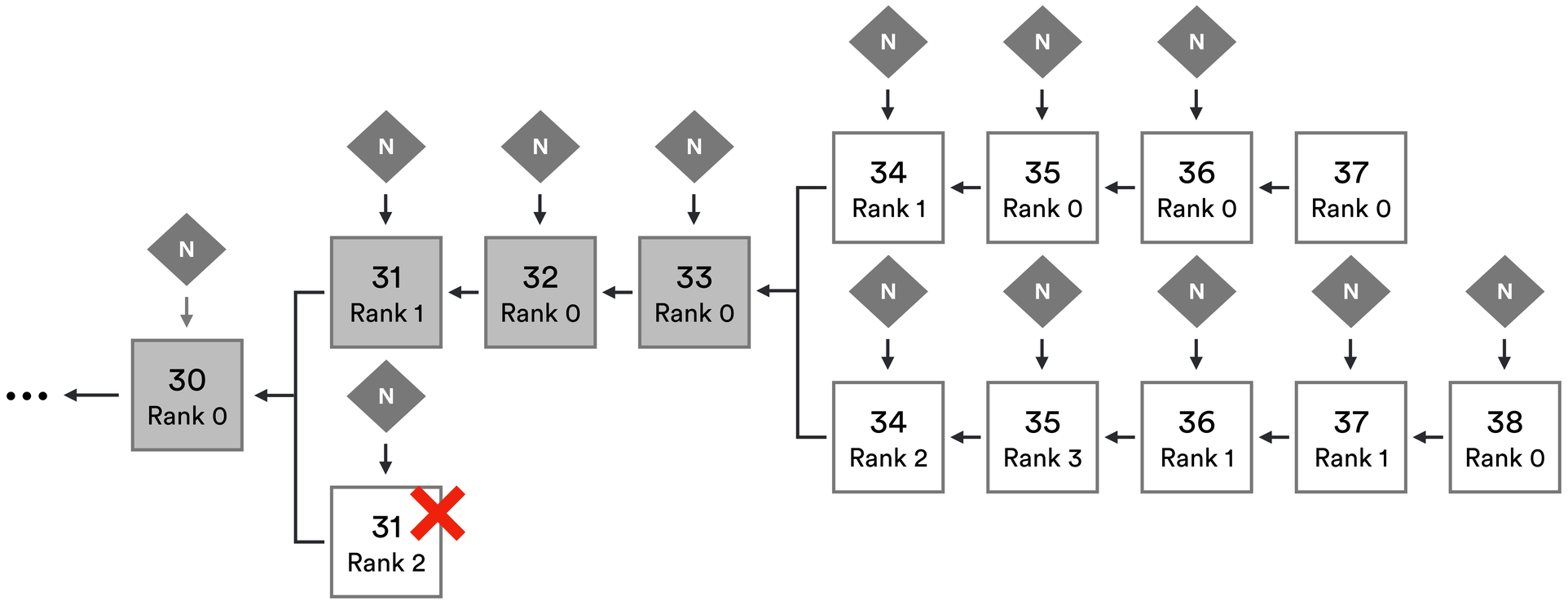
3.13 Notarization with Block Maker Ranking

Synchronous communication → Forks can be removed



3.14 Notarization with Block Maker Ranking

Partially synchronous communication → Forks cannot be removed!

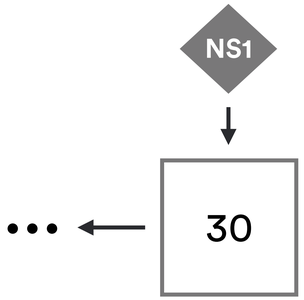


3.15 Finalization

Replicas create finalization shares if they did not sign any other block at that height

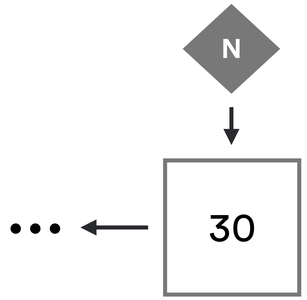
Step 1

Replica 1 notary-signs block *b* at height 30



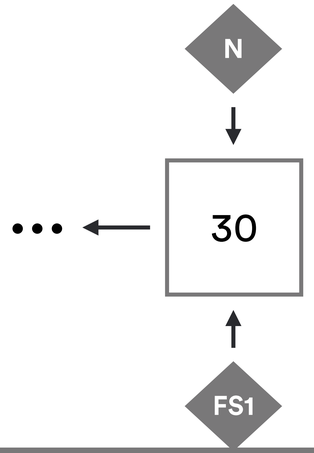
Step 2

Replica 1 observes that block *b* is fully notarized and will no longer notary-sign blocks at height ≤ 30



Step 3

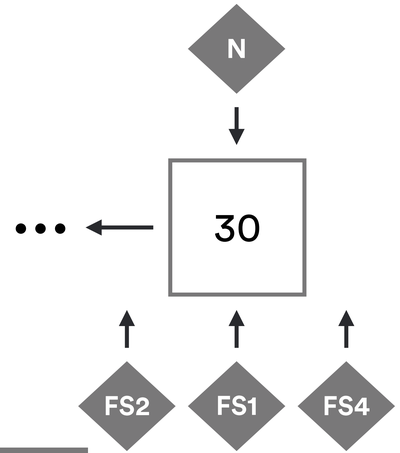
Since replica 1 did not notary-sign any other block than block b, it signs block *b*, creating a finalization-share on *b*



Replica 1 did not notary-sign any height 30 block other than b

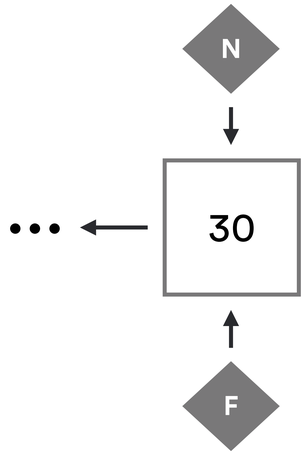
Step 4

Replicas 2 and 4 also cast finalization shares on block *b*



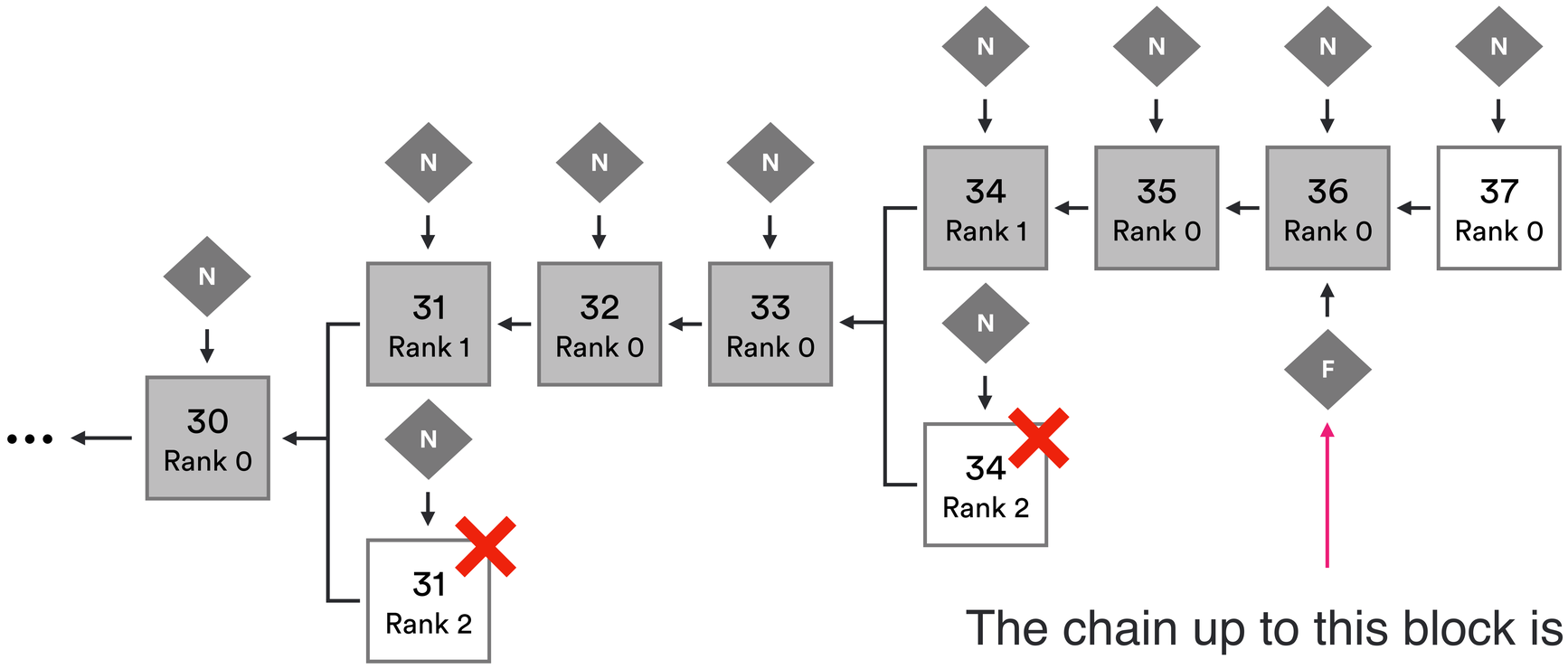
Step 5

3 finalization-shares are sufficient approval: the shares are aggregated into a single full finalization



3.16 Finalization

Finalization on block *b* at height *h* = Proof that no other block is notarized at height *h*



3.17 Agreement

If block b at height h is finalized, then there is no notarized block b’ ≠ b at height h.

Proof:

1. A full finalization on *b* requires ***n-f*** replicas to finality-sign (by construction)
2. At least ***n-2f*** of the **n-f** replicas that finality-signed *b* must be honest (by assumption that ≤ **f** replicas are corrupt)
3. An honest replica that finality-signed *b* did not notary-sign any other block at height *h* (by construction)
4. At least ***n-2f*** replicas did not notary-sign any height *h* block other than *b* (by 2. & 3.)
5. A full notarization requires ***n-f*** notarization-shares (by construction)
6. The ***n-(n-2f) < n-f*** remaining replicas that may have notary-signed a block b’ are not sufficient to reach the notarization threshold of ***n-f*** (by 4. & 5.)

3.18 Termination

For every block height h, at least one block is notarized.

Proof:

1. An honest replica will eventually broadcast a block b (its own or a received block from a replica with a lower rank).
2. All honest replicas will eventually receive block b and notary-sign it unless another block b’ is already fully notarized.
3. If there is no such block b’, all honest replicas will eventually receive notarization shares at least from all (**n-f**) honest replicas and thus notarize block b.

3.19 Termination

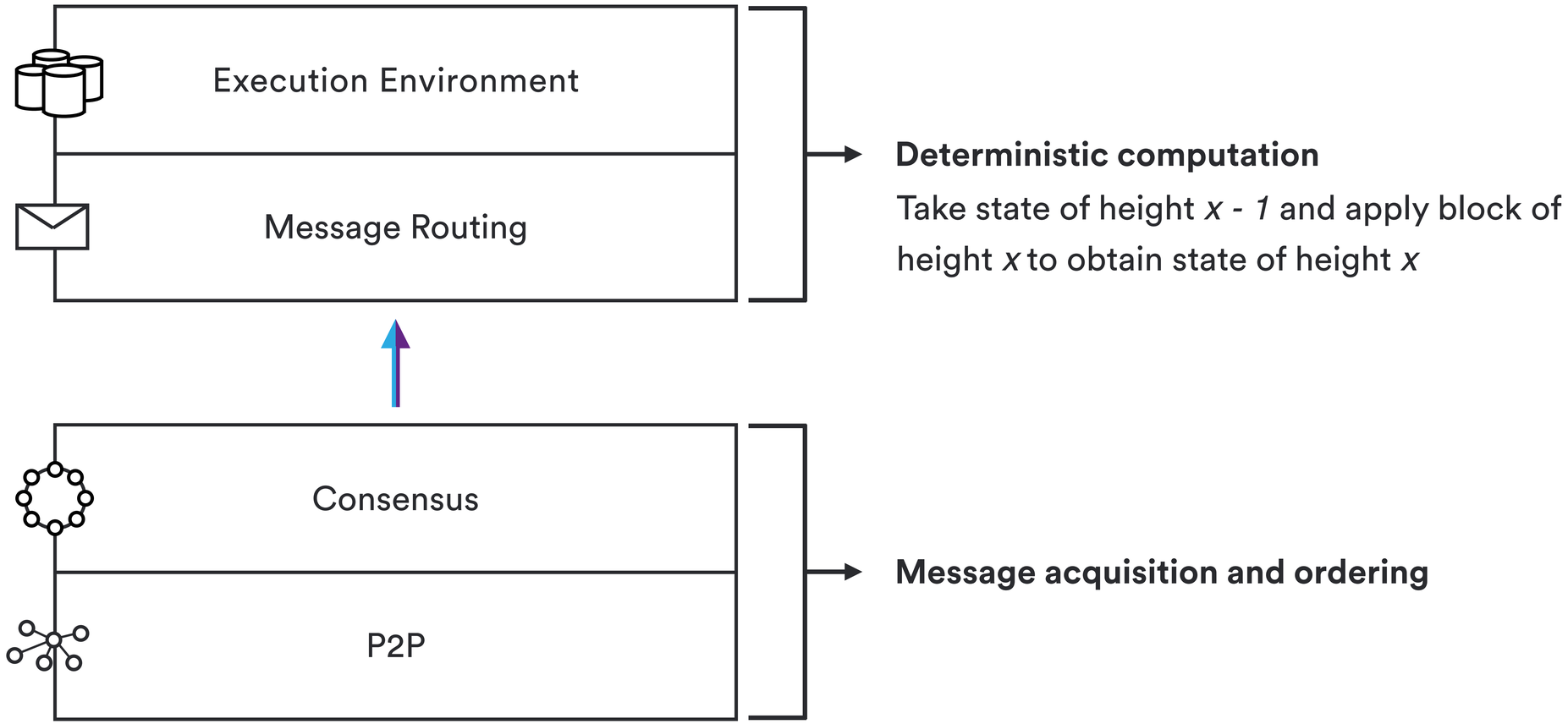
If communication is synchronous for 3 time units at the beginning of the execution for block height h and the rank-0 block maker is honest, then its block will be finalized.

Proof:

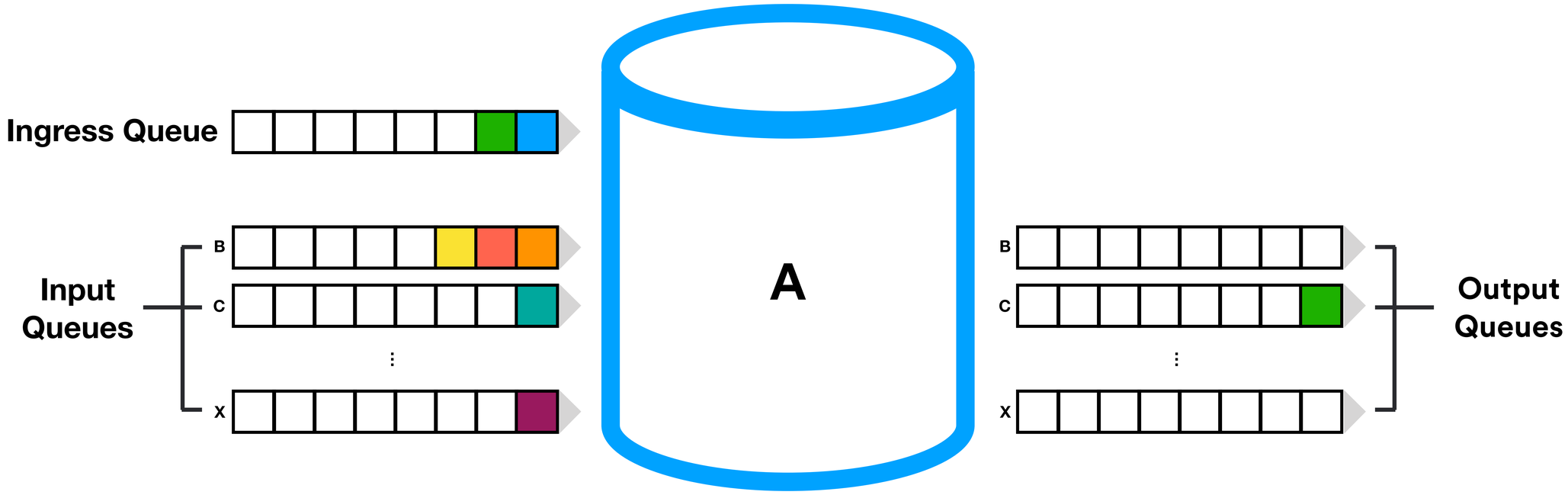
1. If the honest rank-0 block maker broadcasts its block b at time t, it is notary-signed by every honest replica by time t+1.
2. Every honest replica receives at least **n-f** notarization shares by time t+2 and notarizes block b.
3. No other block proposal is accepted by time t+2, therefore every honest replica broadcasts its finalization share for block b.
4. By time t+3, every honest replica receives at least **n-f** finalization shares for block b and finalizes block b.

4. Message Routing and Execution Layer

4.0 The Four Layers of the Internet Computer Protocol

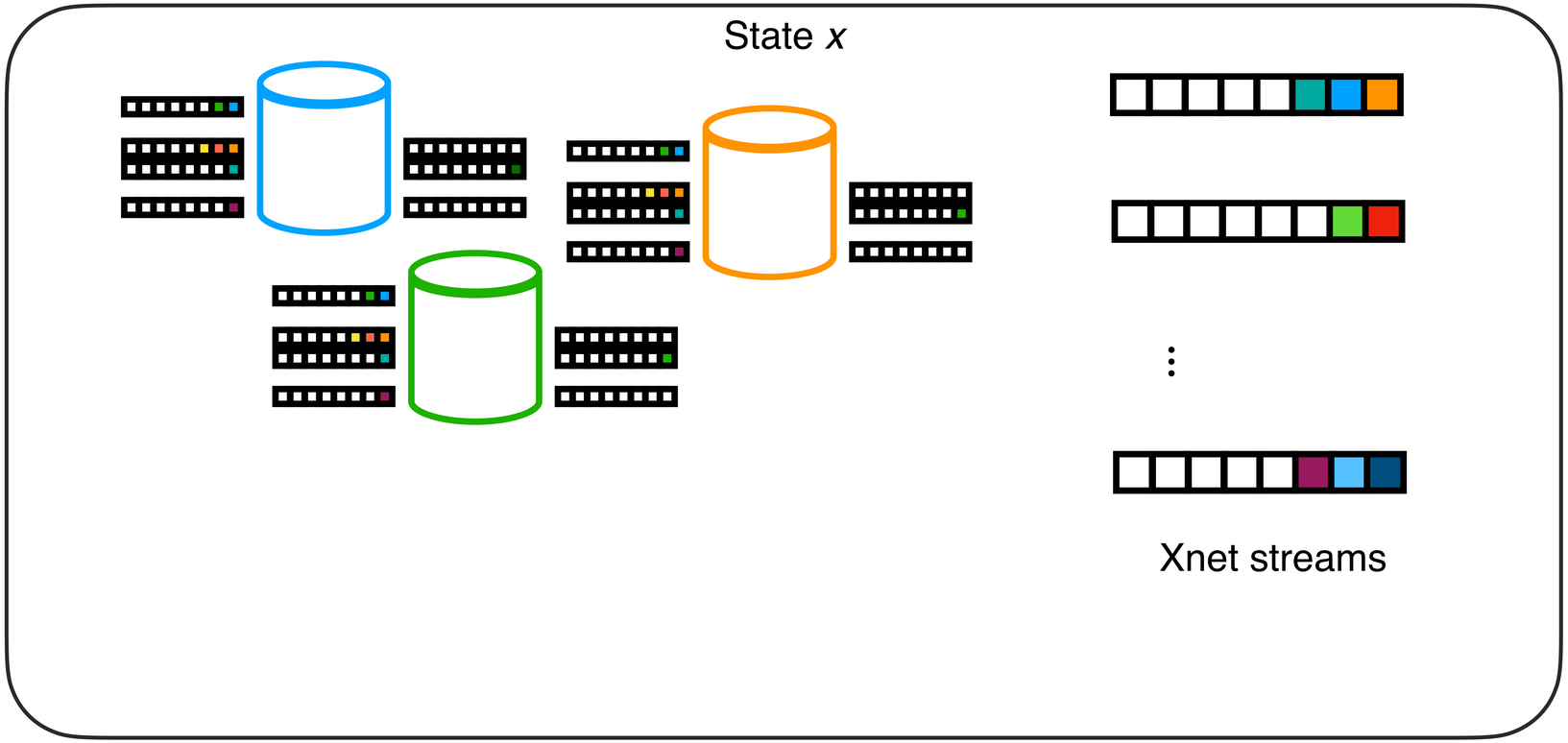


4.2 Canisters and their Input and Output Queues

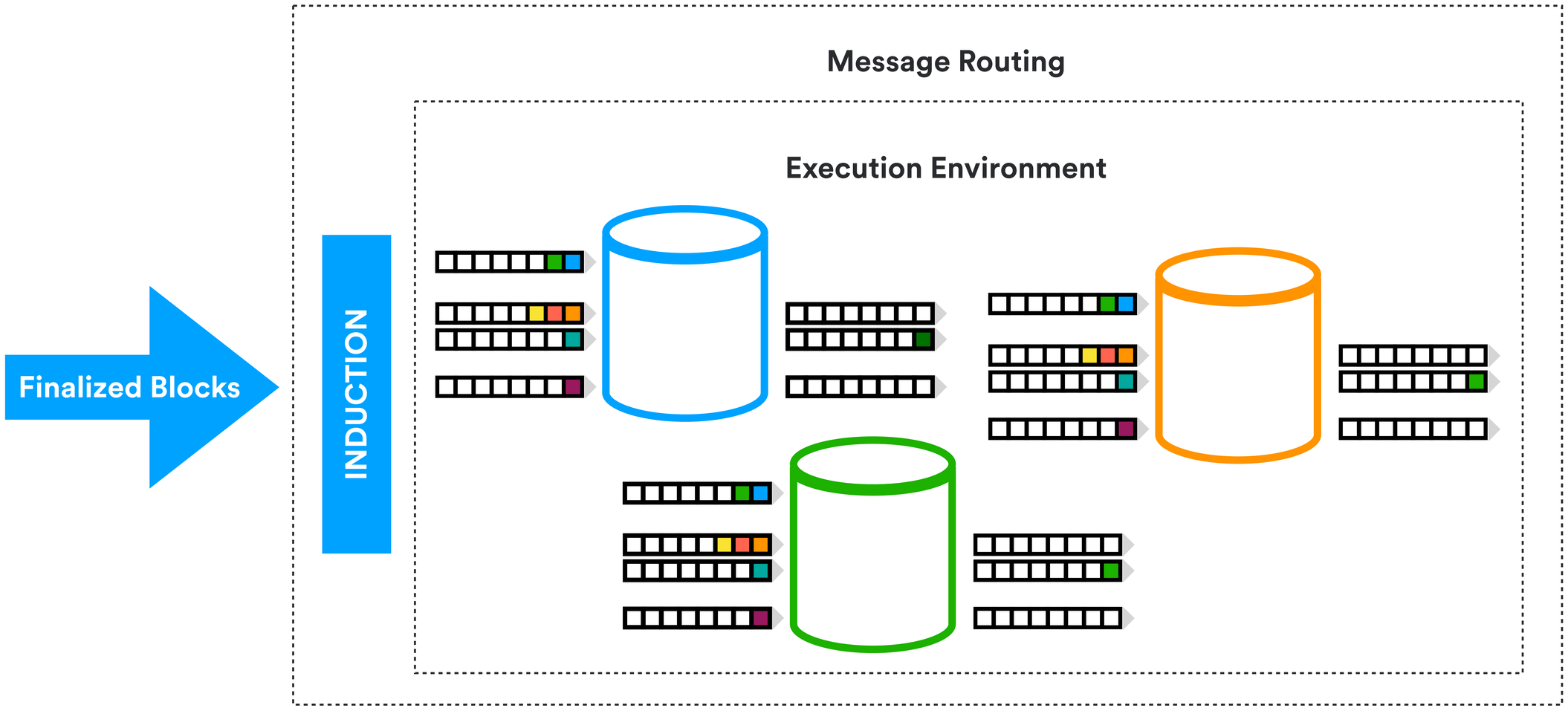


Input & Output Queues of Canisters

4.3 Subnet State

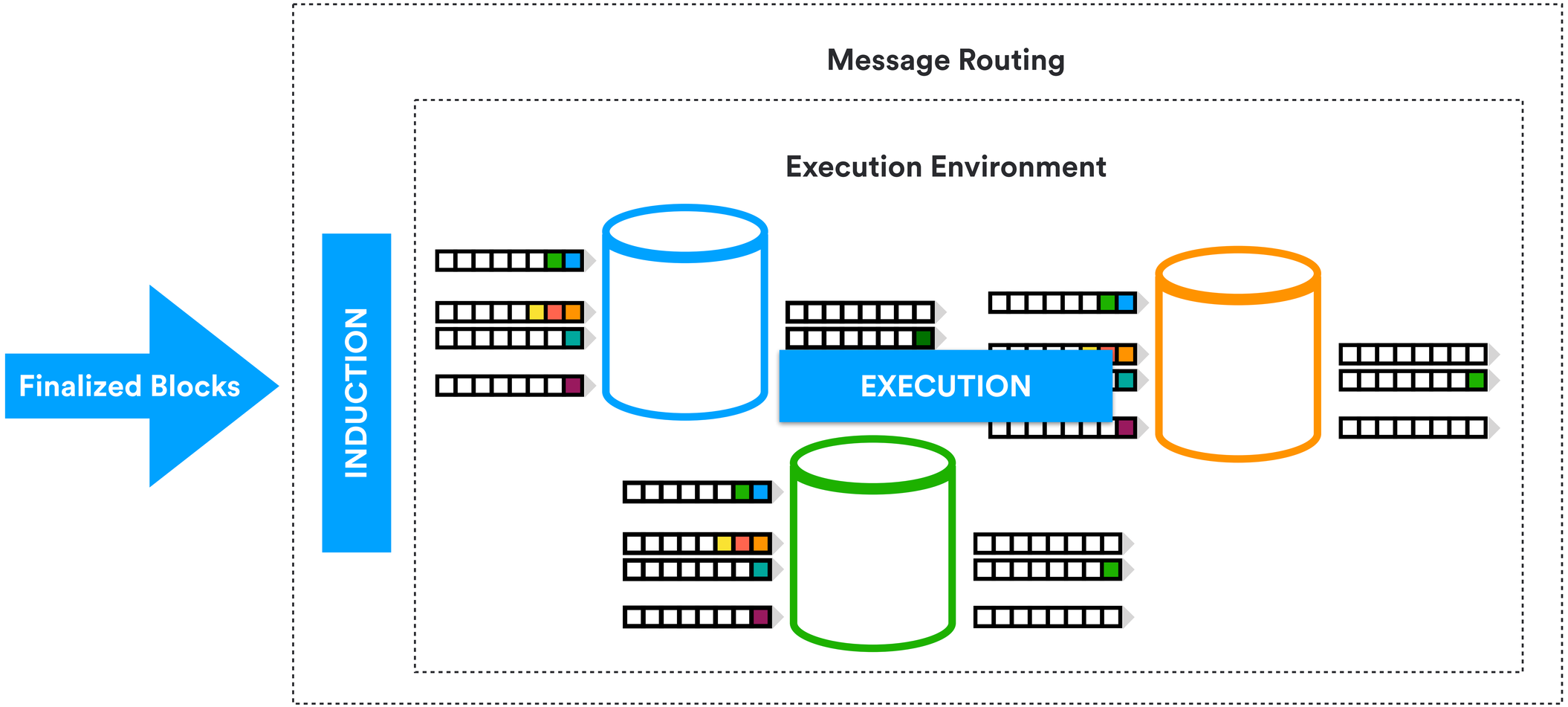


4.4 High-Level Intuition of a Deterministic Execution Cycle



Take messages from finalized blocks
and put them into input/ingress queues
of the respective canisters.

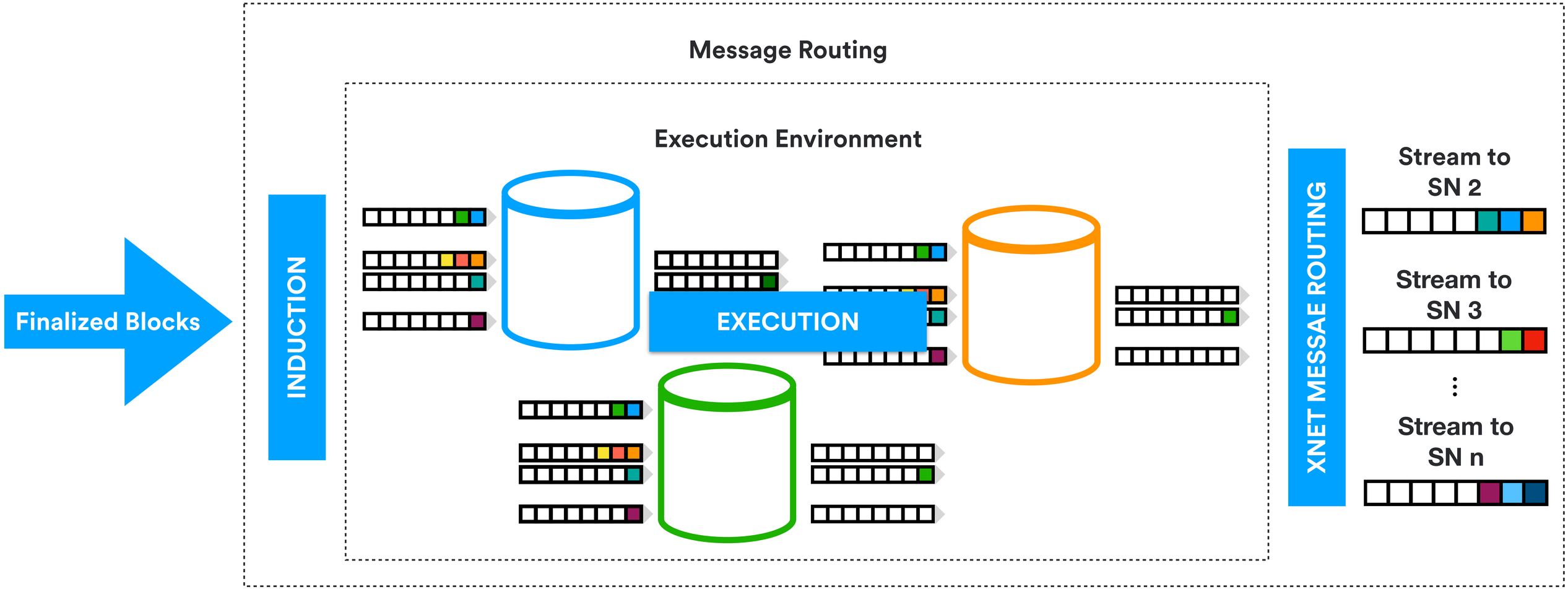
4.5 High-Level Intuition of a Deterministic Execution Cycle



Take messages from finalized blocks and put them into input/ingress queues of the respective canisters.

Take messages from input/ingress queues, execute them and put produced messages in output queues

4.6 High-Level Intuition of a Deterministic Execution Cycle

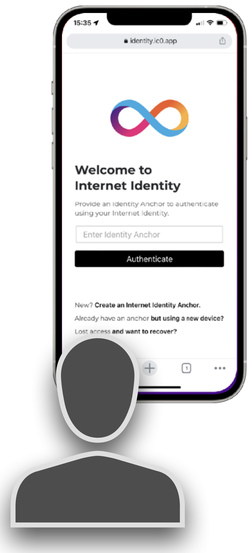


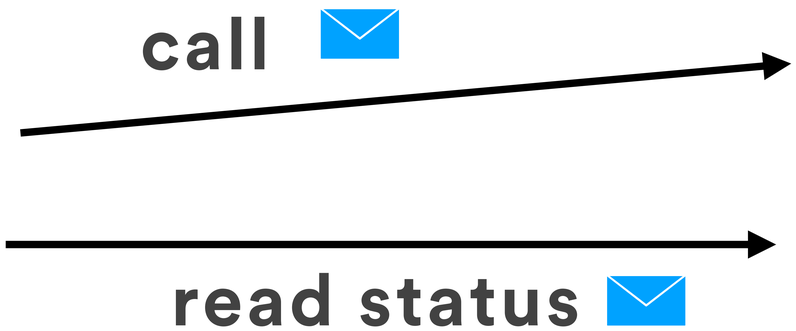
Take messages from finalized blocks and put them into input/ingress queues of the respective canisters.

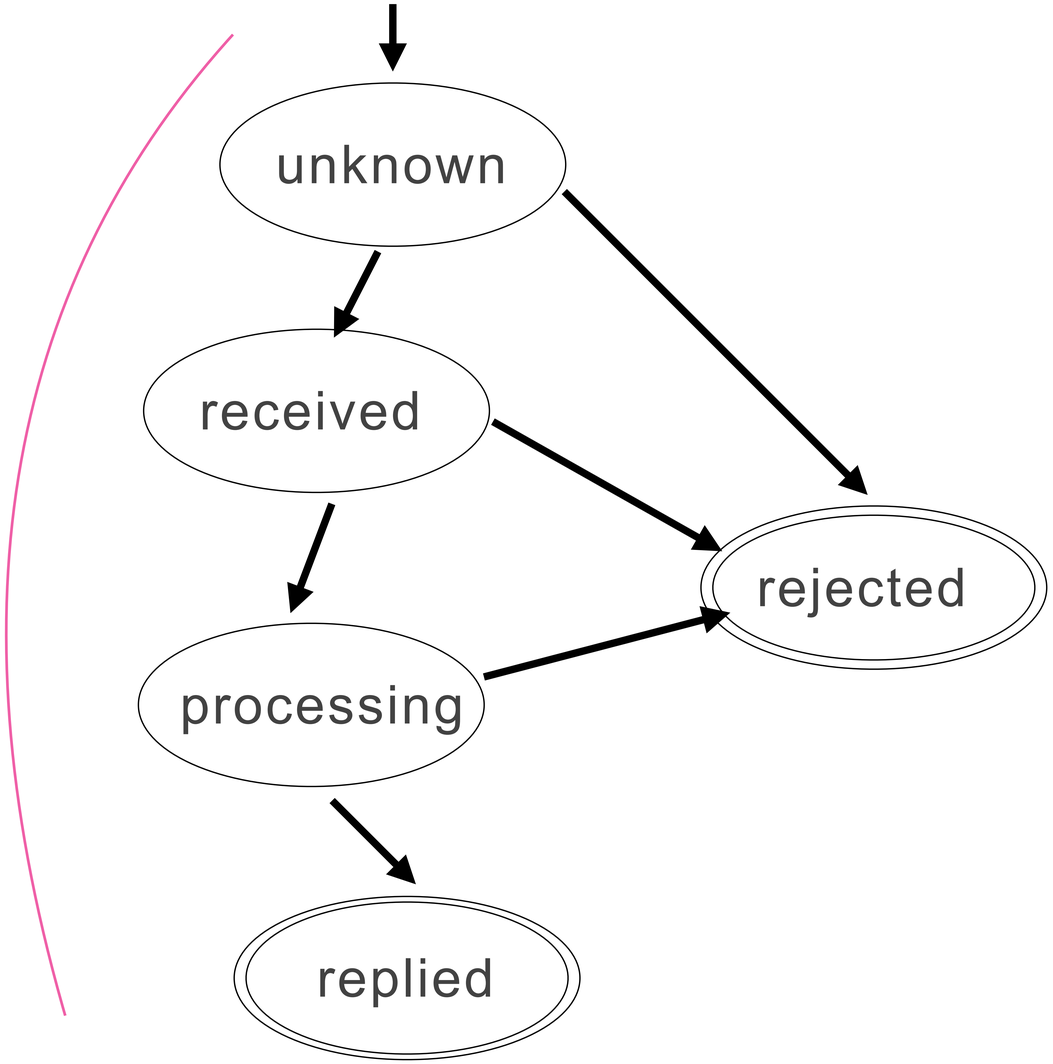
Take messages from input/ingress queues, execute them and put produced messages in output queues

Take messages from output queues, route them in cross subnet streams (respecting ordering)

4.7 Ingress Message Status

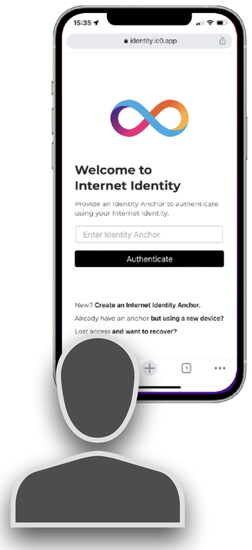


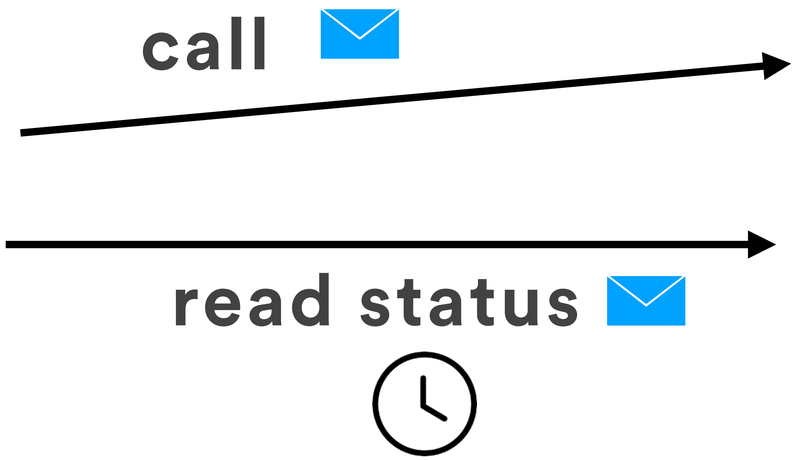


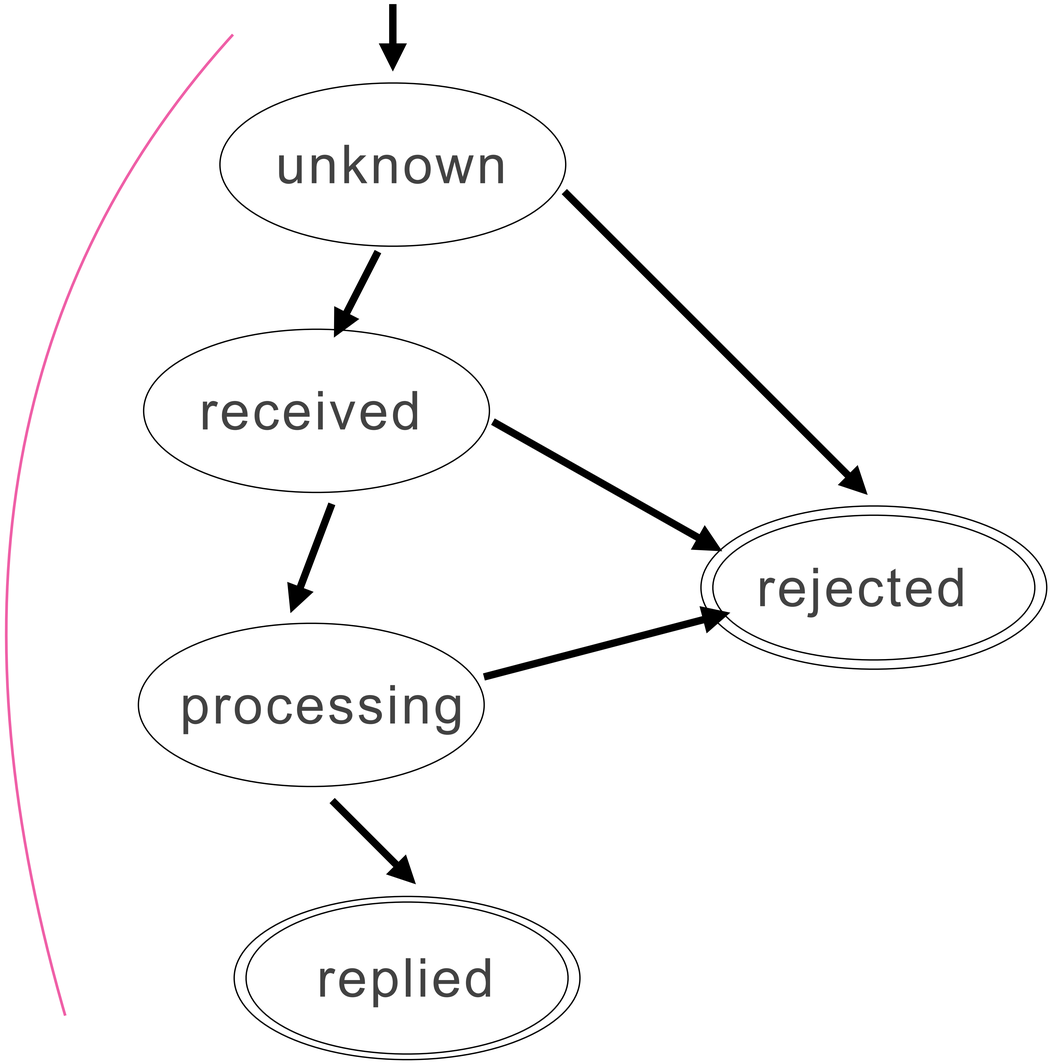


- Status and responses signed by the subnet

4.7 Ingress Message Status

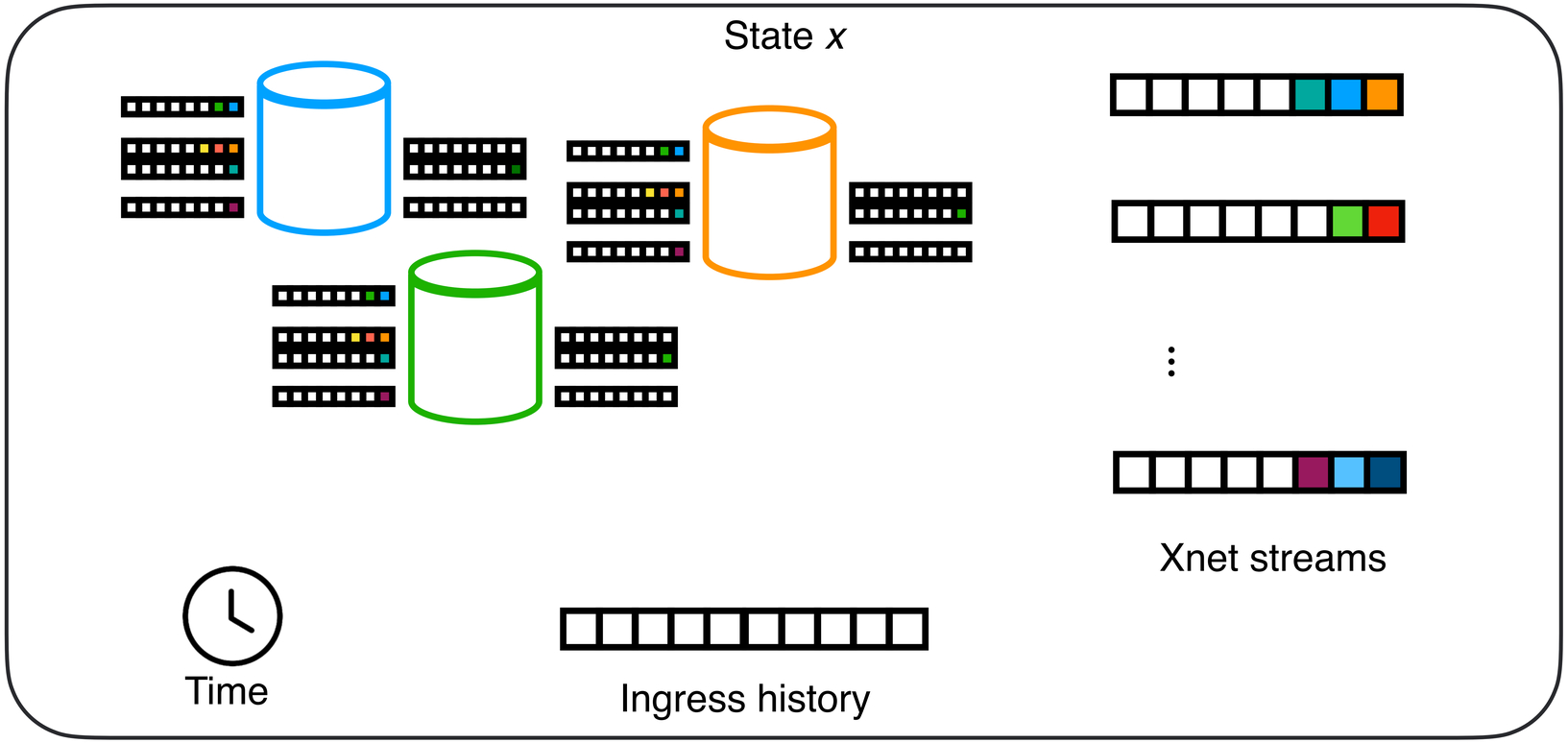




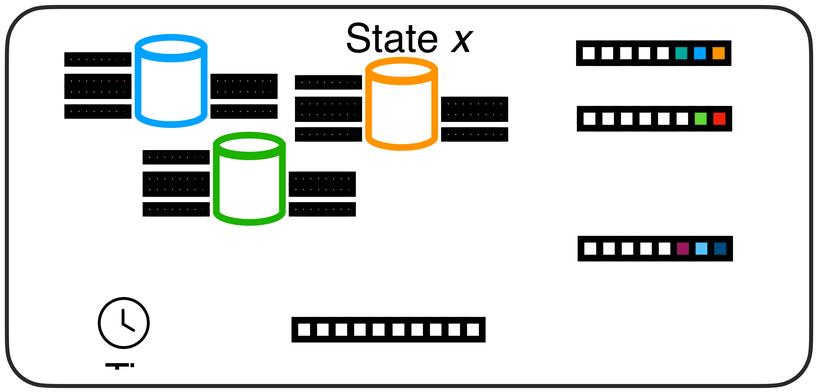


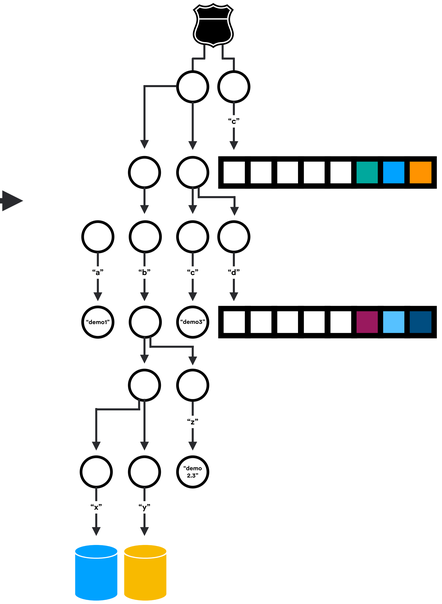
- Status and responses signed by the subnet
- Message expiry

4.8 Subnet State



4.9 Certification after every execution round





Merkle Tree

2f+1 threshold signature

Only one certified state can exist per height

Purpose:
a) output certification
b) non-determinism prevention
c) consensus and execution speed adaptation

4.10 Extend Block and Validity Conditions

Block

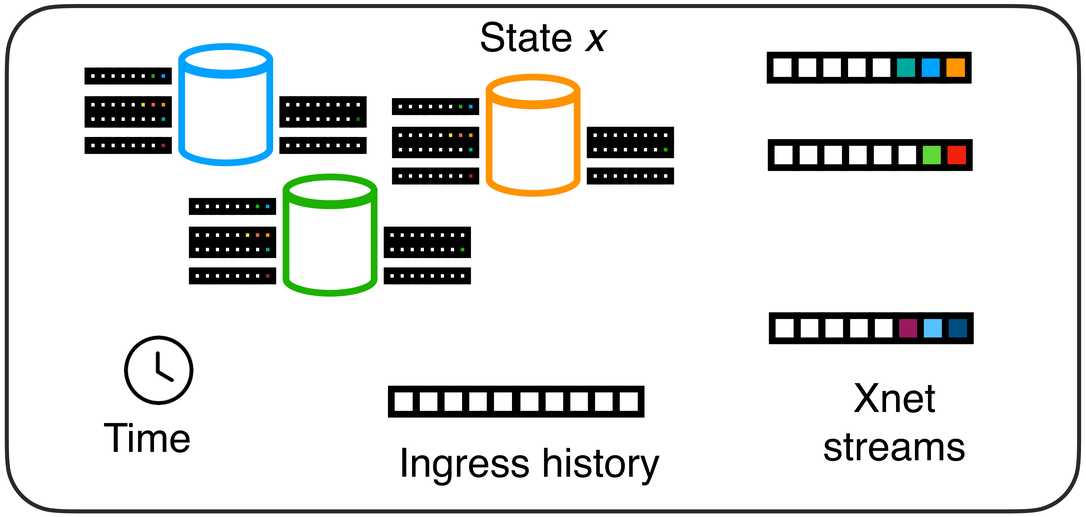
- ingress payload
- xnet payload
- time
- certified height

- block time > previous block time and < local time
- for each message m in ingress payload
  - m doesn't occur in earlier blocks
  - m is signed correctly
  - m's expiry time < block time
- for each message m in xnet payload
  - m doesn't occur in earlier blocks
  - m is signed correctly
  - no xnet stream messages are skipped
- certified height <= local certified height

} **MR data**

4.11 Execution Cycle

upon `process_payload`$(time, ingress, xnet)$ called:
  $state.height := state.height + 1$
  $state.time := time$
  `insert_into_input_queues`$(ingress \cup xnet, state)$
  for $m \in B.ingress$ **do**
    $state.ingress\_history[m] :=$ `RECEIVED`
  end for
  while execution time left **do**
    $m :=$ `pop_from_input_queues`$(state)$
    if $m \in state.ingress\_history$ **then**
      if $state.time < m.expiry$ **then**
        $state.ingress\_history[m] :=$ `PROCESSING`
      else
        $state.ingress\_history[m] :=$ `REJECTED`
      end if
    end if
    `execute`$(m, state)$
  end while
  `prune_ingress_history`$(state)$
  `certify_state`$(state)$ //sets $certified\_height$ eventually
  `xnet_communication`$()$
end upon



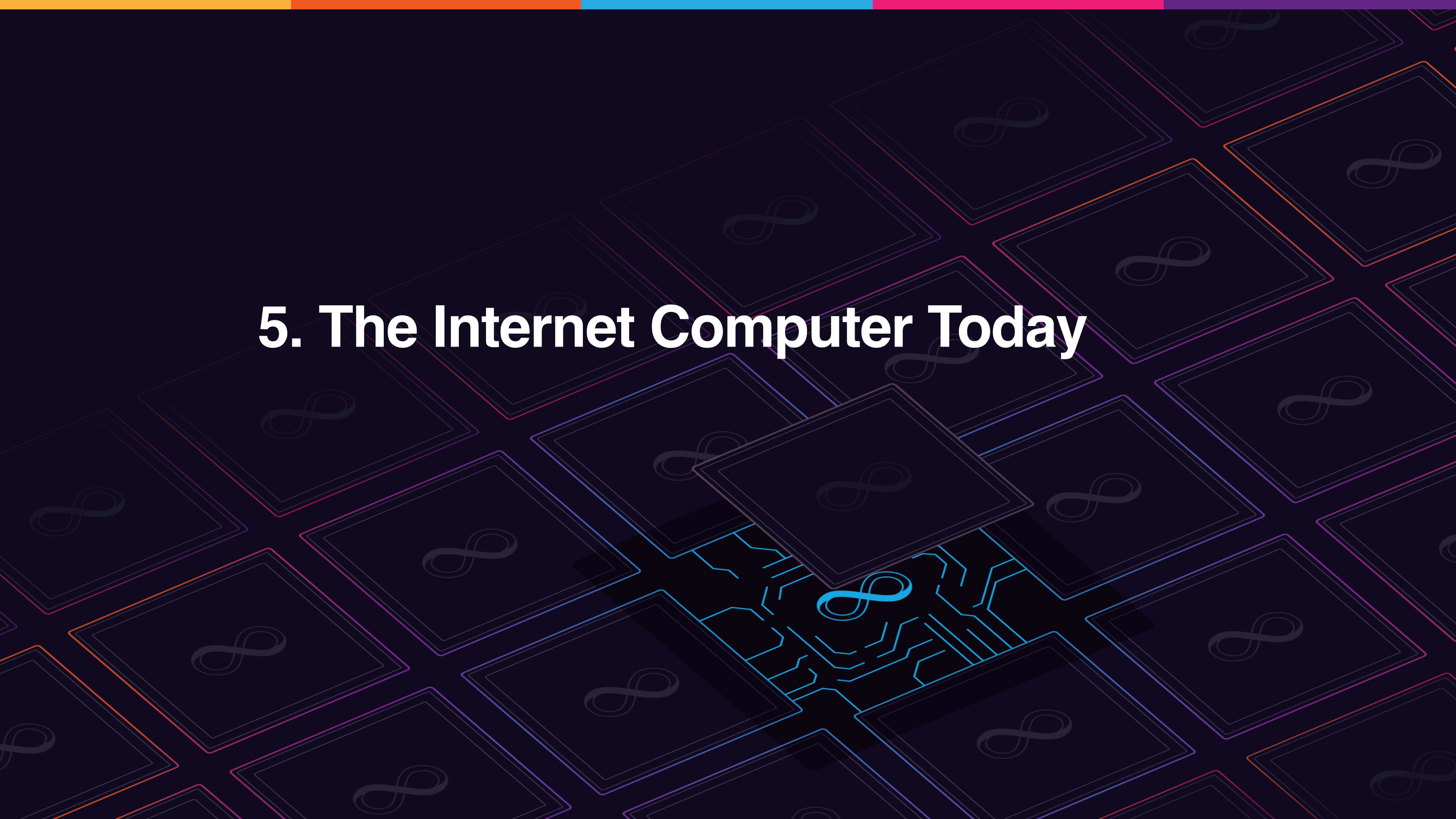
4.12 Unique Execution Before Expiry

An ingress message enters the status processing at most once before its expiry or never. Responses have been computed and signed by at least f+1 honest nodes

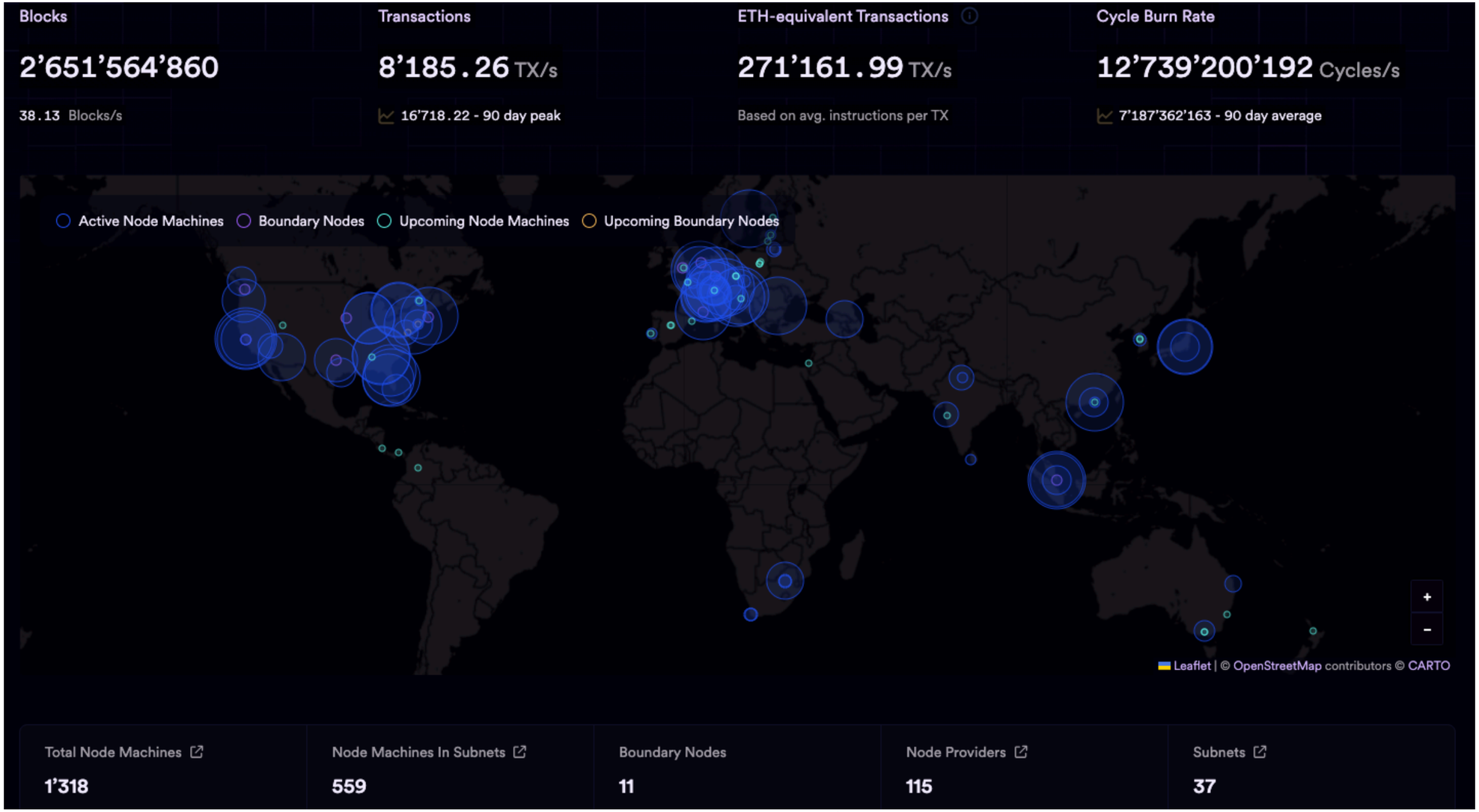
Proof sketch:

1. Time and message in finalised blocks are valid, i.e., > previous block time and expiry > block time respectively, otherwise they wouldn't have been notarised by 2f+1 nodes.
2. No ingress message occurs more than once in blockchain by the same argument, therefore it can be added to its canister queue at most once.
3. Before starting execution (and thus transition) to status processing the expiry time is checked again.
4. 2f+1 signatures are necessary to certify the state, therefore f+1 honest honest are involved in each computation and certification signature.

5. The Internet Computer Today

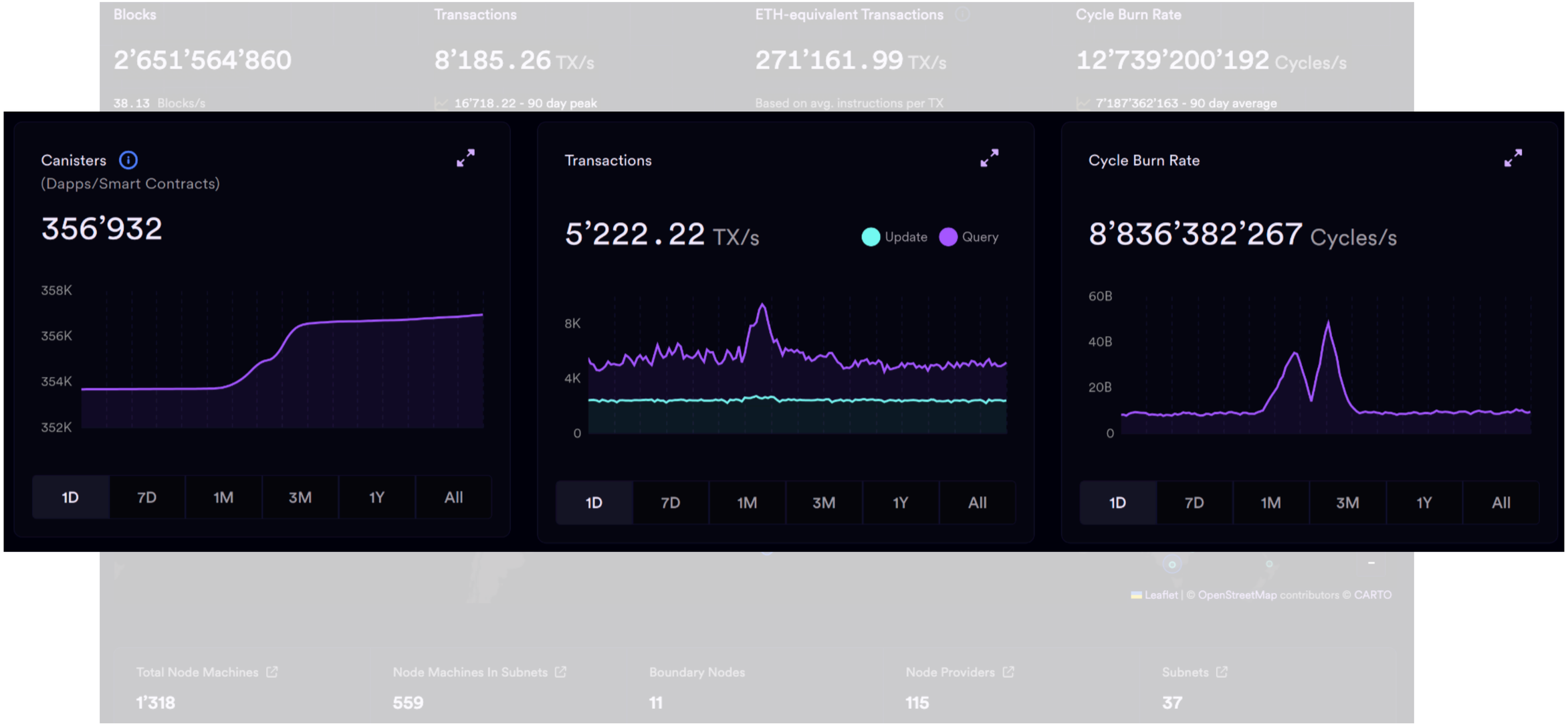


5.1 Live Since May 2021!



https://dashboard.internetcomputer.org

5.1 Live Since May 2021!

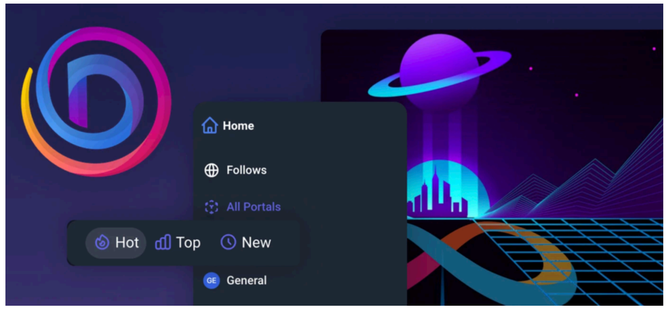


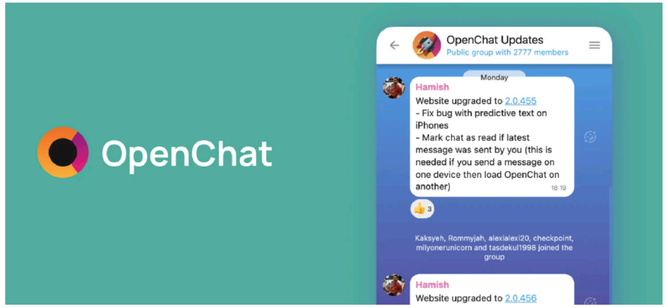
https://dashboard.internetcomputer.org

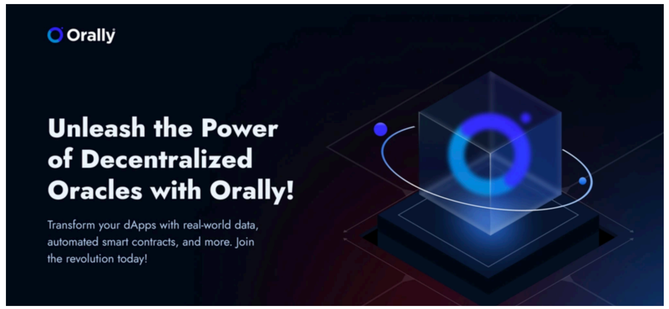
5.2 Many Distributed Systems Problems

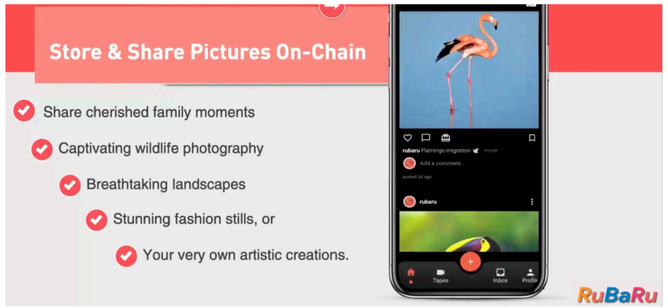
- Disseminating messages among all nodes in the same subset
- Exchanging canister and control messages between subnets
- Scheduling and concurrent execution of canister messages
- Catching up after a node has been offline for a while
- Handling churn (adding and removing nodes)
- Guaranteeing consistency (different users need a consistent view of data and operations)
- Upgrading to next protocol version
- Creating new subnets
- Load balancing
- ...

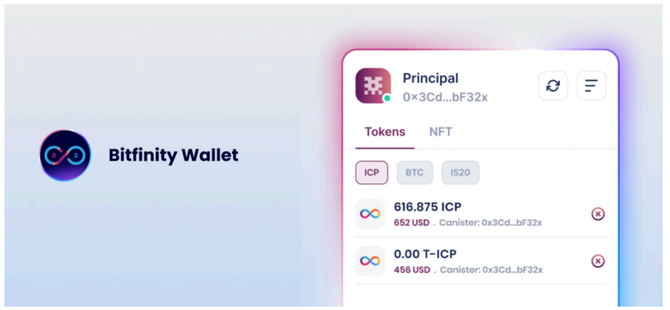
5.3 Growing Blockchain Ecosystem











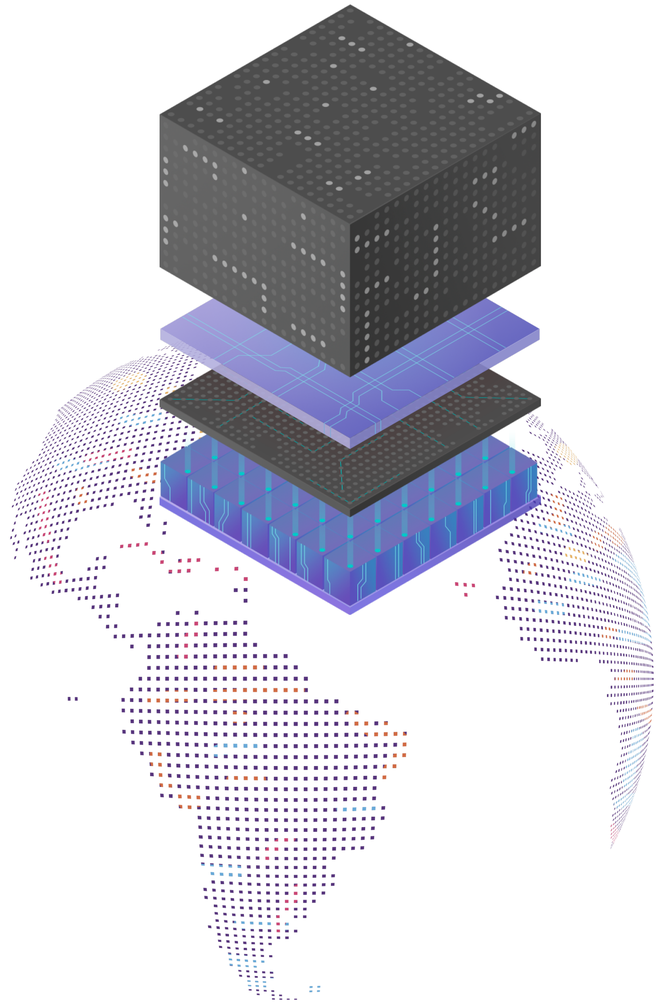
https://internetcomputer.org/ecosystem

5.4 Internet Computer vs. …

			
Average block time:	1 block / 10 minutes	1 block / 15 seconds	38 blocks / second
Finality:	1 hour	15 minutes	1-3 seconds
TX per second:	7	15	33,000 (write) 2,300,000 (read)
Validation data:	524 GB	1300 GB	96 bytes

5.5 More information

- Website: here
- Technical Library: here (videos of talks) and here (blogposts)
- 200,000,000 CHF Developer Grant Program here
- Developer Docs and SDK: here




DFINITY