TECHNICAL WHITEPAPER

ico@sphere.ee

CONTENTS

ABSTRACT

Sphere is a distributed light-weight network for smart-contracts execution and digital commodities management. It ensures instant transaction execution in designated geo-zone, provides intelligent scalability, guarantees network synchronisation and resilience, and eliminates the need of mining.

Sphere Network. Network core is an open source software written in Scala. It is based on Akka stack deployed in a cloud (Google Cloud or AWS). It maintains processing, verification, execution and chaining of the transaction into its own blockchain. Transaction processing throughput is from **30 000 per second within one geo-cluster**.

SPH tokens. SPH is a cryptocurrency, an investment gold backed token fixed by investment gold price on LBMA (London Bullion Market Association). Tokens are both an asset and the means of payment in the material world and can also play the role of an exchange instrument for the SPH owner. 1 SPH is equal to 1 ounce of gold under LBMA at the moment of purchase or sale.

Mining. Once computing power is used for transactions processing investor's income is formed as a fee for transaction. Unlike "traditional" Proof of Work of mining, SPH mining is absolutely efficient (investor is guaranteed to receive income from each transaction) and environment friendly.

Open API. Open API helps to integrate own marketplace into Sphere solutions ecosystem painlessly. Third-party integrations broaden and scale the Network increasing the number of transactions and thus investor's income.

Third-party integrations. As an open-source software, Sphere can be engaged as a scalable and resilient fintech tool for internal (B2B) and external (B2B, B2C, C2C) intelligent transactions (smart-contracts):

- **B2B.** Organisations can issue their own Sphere-compliment tokens over Sphere networks spawned in a private cloud. It may be used internally (for transaction costs reducing, financial control, accounting simplification and automatisation) or externally (for mutual settlements, smart-contracts, etc.)
- **B2C and C2C.** Sphere Platform (see the next chapter) is the reference implementation of B2C and C2C with Sphere, however, one can also deploy own Sphere-based online platform with own Sphere-compliment token to create a new marketplace.

Current whitepaper covers theoretical foundations and technical implementation of Sphere. For wider economical and legal rationale, please, check Legal whitepaper.

HISTORY

Since blockchain technology has been introduced, it had 2 major issues: scalability and trust (consensus). Persisting massive real-time data for million clients in distributed network implies massive async messaging over the wire and eventual data log consistency, which causes scalability problems in classic blockchain approach.

In classical **Proof of Work (PoW)** model, there's an existential scalability restriction, a mining. As block creation requires more computational resources, its issuing speed is confined with worldwide mining powers and network throughput.

In addition, PoW miners competition is absolutely inefficient, it causes ecological problems and appears to be pseudo-random mining race winner takes all merits for block, but in fact, larger mining pools will always apprehend. Sometimes even a block size is not optimal, but dictated by historical or economical reasons.

Proof of State (PoS) consensus model, from the other hand, switches from «hashing race» mining to «wallet mining» - the nodes carrying wallets with larger balances, are «heavier» and thus are more «trusted». It leads rewards centralisation and network problems due to unbalanced traffic towards «heavier» nodes.

SPHERE NETWORK

Sphere effectively resolves following modern blockchain networks issues.

Main features are:

- **Cloud native.** Sphere utilises all the advantages of cloud computing. Cloud platform provides all the tools for seamless and rapid development and cluster provisioning with Akka Cluster: high availability cluster (HA), intelligent task dispatching, auto-scaling and load balancing, monitoring, self-healing, concurrent networking with no idle time.
- **Efficient data sharing** across the network with **Akka Distributed Data**. Due to efficient biased gossip protocol, a consistency across Distributed Data is gained quickly. With consistency control tools and relying on special CvRDTs (convergent replicated data types) it simplifies concurrency control and data sharing across computing clusters.
- **Efficient data-transfer protocol.** Sphere gRPC high-performance, open-source universal RPC framework that utilises **protobuf**, an industrial standard protocol for structured data serialisation.
- **Scalable.** Proven scalability up to **30k transactions per second**.
- **Safety.** All Full nodes are **legally identified** entities or persons, verified by Estonian police and government and authorised by Sphere's legal partners. All operations, including consensuses gathering are happening under regular clients surveillance.
- **Fair participation rewards.** Full node holders immediately receive their fee for transaction verification and execution.

HOW SPHERE NETWORK WORKS

There are three fundamental entities in Sphere Network:

- **Client Nodes** (client-side applications communicating with Full Nodes via sRPC)
- **Full Nodes** (computing clusters)
- **Archive Nodes** (blockchain replicators)

Clients Nodes. Effectively, Client Node is a thin client. As a client-side application Client Node could be installed anywhere - mobile phone, bare metal server, cloud, etc. It requires only a Sphere wallet to become a Client Node. Client to Full node communication utilises **sRPC**, smart-contracts API layer over gRPC with TLS/SSL security.

Full Nodes. Full Nodes are clusters of virtual computing machines (nodes) grouped by geographical principle (5 clusters over 4 continents). Full Nodes cluster is spawned on Google Cloud or AWS zones.

Each Full Node is equivalent to specific geo-cluster. Geo-cluster means reducing network overhead. A transaction between geo zones is bound to the issuer locality. For example transaction from the New York to the UK, is served by US geo-cluster.

Full Node is mainly responsible for smart-contracts (transactions) computation lifecycle:

- listening to Client nodes
- filling own memory pool with transactions
- computing transactions from memory pool

- creating a new block
- emitting a block to the blockchain
- adopting a block to the blockchain
- network state management
- carrying recent blocks in memory (blockchain cache)

Full Node is rather complex, so we will take a closer look on them in the next chapter.

Archive Nodes. Archive Nodes are merely carriers of all Sphere blockchain history inside top-level Sphere blockchain (**The Surface**, see below). Under the hood it is an actor built with Akka Persistence over LevelDB.

It is is both available for direct download from Cloud Storage or AWS S3 or as a BitTorrent file. The downloadable compiled file is Scala application built on top of LevelDB, which offers query API and automatic data accumulation functionality to keep track live changes in Sphere blockchain.

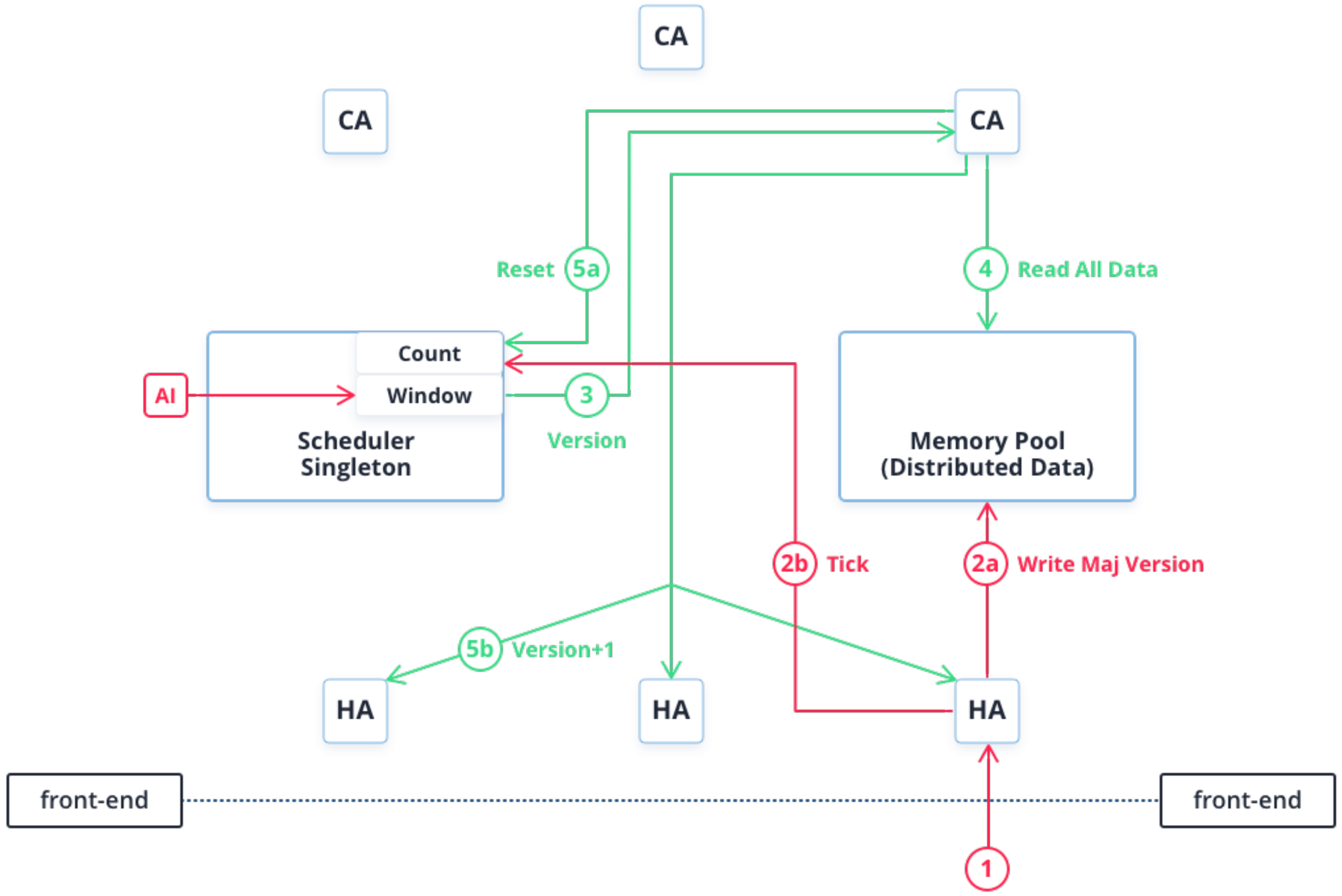
FULL NODES

Full Node is auto-scaled cluster of dedicated servers carrying actors under Akka Cluster provision. A joint network of Full Nodes conforms an internal network mesh across continents.

Full Node topology

Full Node itself is designed as a reactive unit (actor) and relies on actor concurrency model. Full Node is a resilient, self-healing Akka Cluster, that carries other units:

- Replicator actor is a service actor for Distributed Data control provided by Akka out of the box. Distributed Data shares data between nodes in an Akka Cluster, so it is eventually consistent. Consistency levels control is provided out of the box.
- Memory Pool (memPool) implements ORMultiMap data structure handled by Distributed Data. Memory Pool is a shared map for all transactions came from outside world. The key is memPool generation (version), whereas the value is GSet (growing Set) with transactions' code.



Pic 1: single Node

- **Front-end actors (FA)** are units based on Akka HTTP listening to Client nodes requests (sRPC) within own geo-zone. Operations:

  1 Once Front-end actor receives a request, it **assigns current generation (version)** to the data (generation counter is GCounter);

  2a Front-end actor stores versioned data into the memPool with consistency WriteMajority (2n+1/2 VM in a cluster, thus even if a VM carrying this Front-end actor dies and/or fails to commit a transaction into the memPool, a Front-end actor of another VM will eventually do that);

  2b Front-end actor sends the **tick** to Scheduler actor.

- **Scheduler actor** is a singleton (one instance per cluster) actor handling memPool drainage e.g. dispatching computation of a certain memPool generation. Operations:

  2b Scheduler increments its atomic counter (PNCounter);

  3 Scheduler assigns a **task** to be computed to the least loaded Computational actor. It happens a under special condition, a **window moment** — whether its atomic counter is near an transactions-per-block optimum, or it has been a timeout (equivalent to Kafka's linger.ms).

- **Computational actors (CA)** are units waiting for a task (e.g. certain memPool generation calculation) to be assigned by Scheduler actor. Operations:

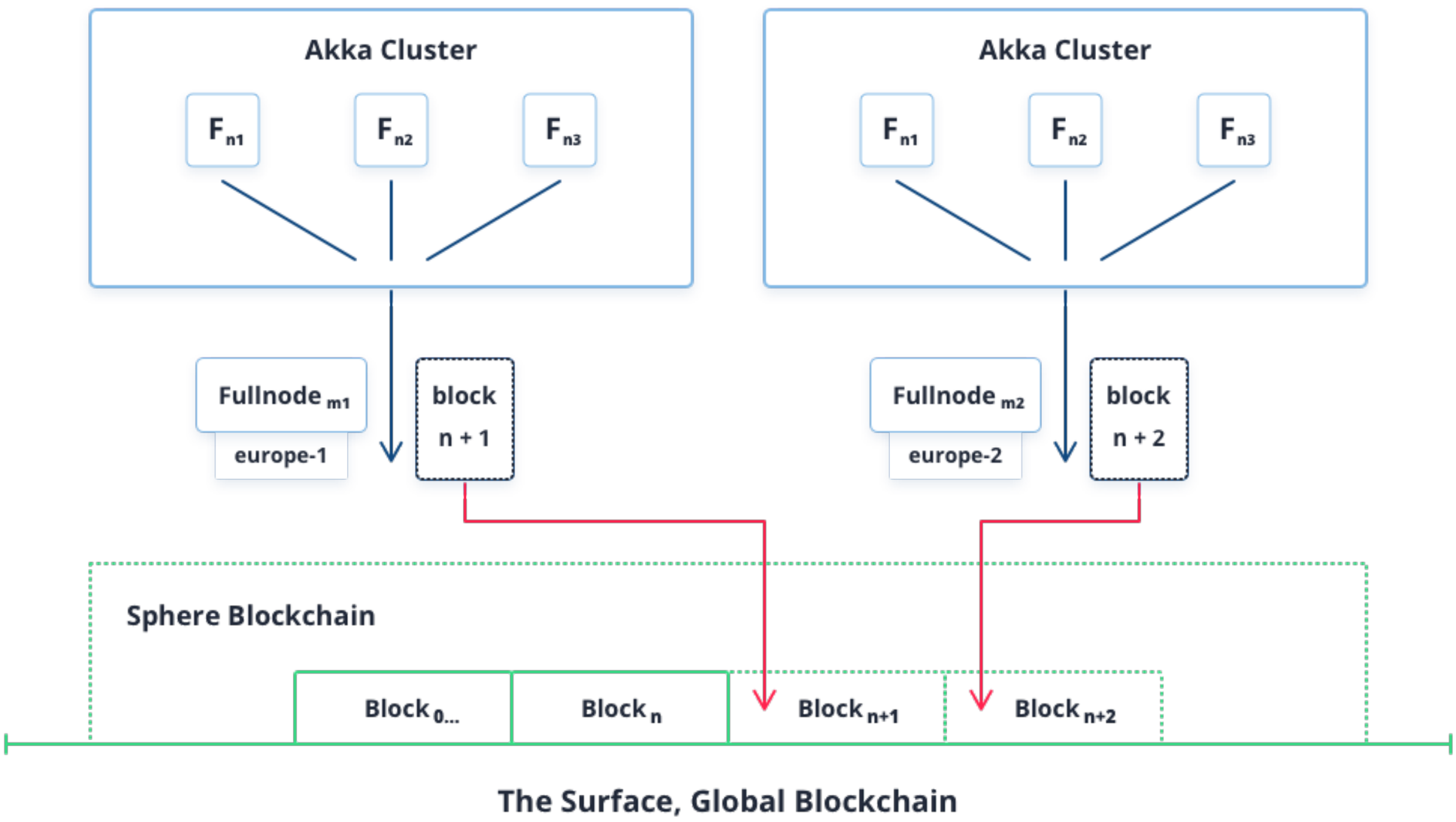
  4 Once the task is received, Computational actor reads specific generation (version) from memPool with ReadAll consistency and commences transactions calculations;

  5a Once all the transactions are verified, calculated and grouped into a Transactional Merkle tree, Computational actor issues a block and notifies Scheduler to clean up own atomic counter once that block is accepted and committed into the Surface (top level blockchain);

  5b Computational actor notifies all Front-end Actors to increment their shared generation GCounter with WriteMajority consistency;

***** literal suffix in operation number 2a, 2b, 2.. means that operations happen concurrently

The Surface



Pic 2: Surface

The Surface is **top-level blockchain handler** operating across multiple Full Nodes (geo-clusters). It accepts blocks issued and accepted by every Full Node.

Blocks ordering is guaranteed by a special data structure that carries:

- **HashSet** for blockchain full history (ever-growing custom data structure that extends GSet)
- **register** (LWWRegister) for new-coming block
- **register** (LWWRegister) for current geo-cluster State Radix Merkle tree header.

As a composition of Distributed Data CvRDTs, Surface guarantees updates atomicity and true data synchronisation using version vectors (Lamport's logical clock implementation) under the hood.

State (wallet states) handling and migration

Each wallet state is represented by **State actor**. One may think about about the State actor as a middleware triggered once **Computational actors** collapse **Transactional Merkle tree** sub-tree calculation. Thus is handles the most actual state concurrently with **Transactional Merkle tree computation**.

State actor persistence is guaranteed by Akka Distributed Data ORMultimap where key is wallet_id and the value is another ORMultimap of balances: actual (float) and frozen (map-like structure for ESCROW smart-contracts).

Each State actor update triggers **State Radix Merkle tree** update (a data structure similar to Ripple's Hash Tree), that is needed for enforcing blockchain fidelity and **wallet migration** facilitation.

The important note here is that **State actor is bound to geo-cluster locality**: it always has registration in certain cluster. Multiple registrations are not possible i.e. only one geo-cluster carries actual state of the certain wallet.

However, it is possible to **reassign State actor locality** — the process called **State migration**. Migration is basically a classic **2PC** (two phase commit) process: once the leader for host geo-cluster elected, it asks other host geo-cluster Nodes to accept (migrate and persist) a new **State actor** with all the underlying data layer. Once transaction is committed, the donor geo-cluster will release the state actor and erase the underlying data.

The migration decision is the responsibility of **AI module**.

AI module

AI module is a detached, off-network actor running **Sphere Statistical Engine** powered by the Spark MLlib. Key responsibilities:

- define better window moment predicates
- define better conditions for State actor migration decision

One may think about the **AI module** as a lazy listener that tracks all the network events and makes a recommendation (with WriteMajority consistency) about what conditions are optimal to perform transaction blocking (memPool drainage) and what is the case when State actor migration decision pays off. The main method is regression analysis (linear, polynomial, GAM).

SMART CONTRACTS

As it was mentioned above, Full Nodes API is **sRPC**, a smart-contracts API. In Sphere **Smart Contract** is merely a RPC attached to the transaction e.i. it is a part of the source code and thus does not require resources for compilation.

There are two pre-built Smart Contracts available:

- refundable / non-refundable **transaction**, which is a simple transaction with optional metadata for Client app
- **ESCROW**, a safe transaction with automatic funds freezing and release at the moment the deal is closed

One can introduce own Smart Contract by the following procedure:

1. Create a Smart Contract according to Sphere guidelines (must be written in Scala or Java, must have unit tests, must obey packages import constrains, etc.).
2. Ensure it passes Smart Contract **Integration tests suite**.
3. Create a **pull request** to sphere/smartcontracts repo.
4. Once pull request is accepted, the Smart Contract will be deployed by automatic CI/CD pipeline and become available via sRPC.