INSOLAR

Insolar

Release 1.0

Sofia Yemelianova and collaborators

Sep 03, 2019

OVERVIEW

Note: Insolar's documentation is under development, its structure is not yet finalized. Expect updates soon.

To get a grip on how Insolar works, take a look at its *architecture overview*.

To connect to TestNet 1.1, go through *step-by-step instructions*.

CHAPTER

ONE

ABOUT INSOLAR

Insolar outlines a new vision for blockchain, in which key features such as network capacity and consensus are addressed in a way that will enable shared business processes and optimize contractual transactions for enterprise operations.

The Insolar platform offers flexible governance, which will allow users to choose between using an Insolar public network—creating a domain with its own rules—or hybridizing the public network and a private domain. Private domains can be stand-alone private networks with their own servers or permissioned networks with resources provided by ecosystem members. This flexibility enables distributed business networks where everyone can select the configuration and domain features that best meet their needs.

Insolar's smart contracts are designed with enterprise developers' needs in mind. They are streamlined to work with business logic, and they are easy to develop using Golang and, soon, Java and other JVM-based code. Moreover, the Insolar platform provides data safety features that enable businesses to run their operations on public networks if they choose while being compliant with regulations such as the EU's General Data Protection Regulation (GDPR) and other country-specific or company-specific cryptography standards.

Insolar's vision is to facilitate seamless low friction interactions between companies by distributing trust, thus accelerating and opening up new opportunities for innovation and value creation.

The Insolar platform is the most secure, scalable, and comprehensive business-ready blockchain toolkit in the world. Insolar's goal is to give businesses access to features and services that enable them to launch new decentralized applications quickly and easily, whether they need a minimum viable product or full-scale production software and to integrate those applications with existing systems.

CHAPTER

TWO

DISTINCTIVE FEATURES

The Insolar platform is built to satisfy enterprise requirements by combining distributed and cloud technologies and dozens of industry-first features. Insolar is currently designing the proposed platform in an incremental fashion allowing it to progressively grow into the ultimate decentralized collaborative environment for various kinds of industries, companies, governments, and communities. Insolar sees these many features and capabilities as a mandatory part of blockchain implementation and integrates them as part of the Insolar platform.

2.1 Federation of Clouds

Transparently connects multiple clouds based on Insolar technology, where each cloud runs and is governed independently (e.g., by a community, company, industry consortia, or national agency).

2.2 Cloud

Organizes and unifies software capabilities, hardware capacities, and the financial and legal liability of nodes to ensure the transparent and seamless operation of business services.

2.3 Globular Network

The backbone of a cloud. It is a set of protocols enabling the coordination of P2P networks of up to 1,000 nodes and a hierarchical network of up to 100,000 nodes.

2.4 OmniScaling

Is an integral feature that utilizes a multi role model of nodes, a multichain organization of storage, and an innovative approach to distributing work across the network by combining a network-wide membership consensus with deterministic role allocation, and with individual validation groups per each transaction. As a result, OmniScaling enables near-linear and dynamic scalability by CPU, storage capacity, and traffic (network throughput) for domains running within a cloud.

2.5 Domain

Enables different governance models, it defines policies for data and contracts, such as to allow public or permissioned models, or to apply national or industry standards.

2.6 Data Safety

Cost-efficient data safety and leakage prevention for shared cloud solutions through data scattering and density limits, atomic re-encryption, permissioned node access, signatures from nodes that have accessed data, and so on.

2.7 Capacity Marketplace

A special domain within a cloud that defines procedures for business services to do spot and long-term trading for CPU, storage, and traffic as commodities.

2.8 Separation of Business Logic

Helps to focus on what is essential for a business to operate and allows for the deployment of new business services as easy as creating a new mailbox or a website.

2.9 Business Templates

Come from a separation of business logic with the use of domains, making the reuse of business logic possible either as components or as full application templates for easy deployment.

2.10 Per-transaction Consensus

Allows validation and finality requirements to be defined as business logic components on a per-transaction basis to match the value and risk of transaction versus the speed and cost of validation.

2.11 Data and Execution Scattering

Together with atomic proxy re-encryption algorithms, data and execution scattering significantly reduce the impact of intrusions and data leakage for off-premise settings.

2.12 Support for Large and Long Transactions

Makes it easy to exchange documents, build complex business services, and use dynamic binding of services via marketplaces. It significantly reduces the complexity of development and deployment for the storage and processing of off-chain documents, while simultaneously increasing the consistency and integrity of blockchain solutions.

2.13 Integration and Compatibility

Provided via contracts and virtual machines, which can be attached to specific nodes and operate as integration gateways.

2.14 Native Support of Industry-standard Languages

Native support of contracts based on industry-standard languages, such as Golang and JVM, as well as support for distributed transactions and microservice-like integrations of contracts to enable the use of existing practices and skills.

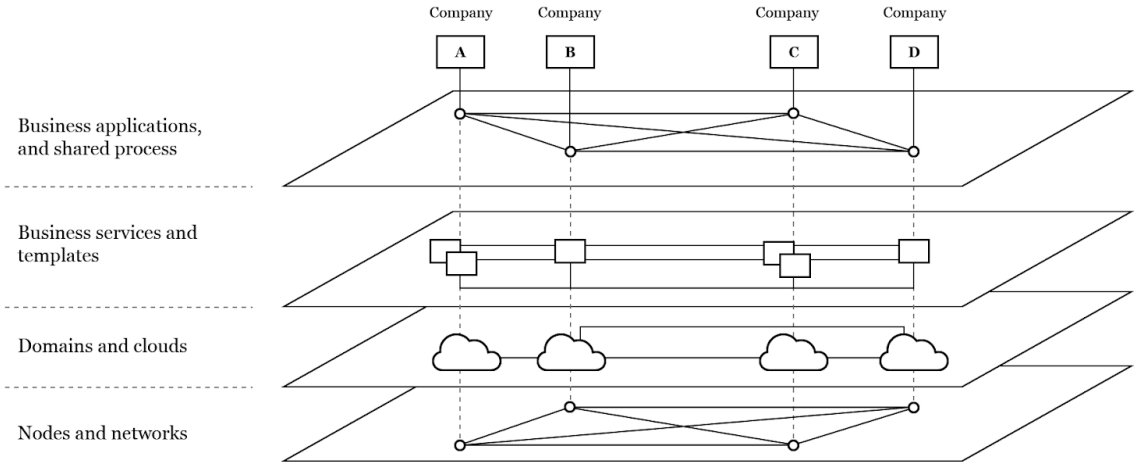
CHAPTER

THREE

ARCHITECTURE

Insolar aims to deliver an open and collaborative environment required to enable third-party companies to build and maintain templates and services, provide hardware capacities, and adapt services and functions to local practices and legal and regulatory requirements.

Below is an illustration of the layered architecture that facilitates such a collaborative environment. The use of multilayer architecture makes platform design a challenging task, but with proper use it enables building complex solutions with better control of development risks and (later) of ownership costs. Insolar is currently designing the proposed platform in an incremental fashion allowing it to progressively grow into the ultimate decentralized collaborative environment for various kinds of industries, companies, governments, and communities.



The architecture is split into four layers:

- At the top layer are applications (*contracts*) owned by and tailored for companies who serve other companies.
- The next layer represents business services and templates (*domains*) for business applications provided by vendors.
- At the third layer is the *federation of clouds*. Their infrastructure can also be public and offered by governments or even communities as a public good (crowd-sourced computational resources).

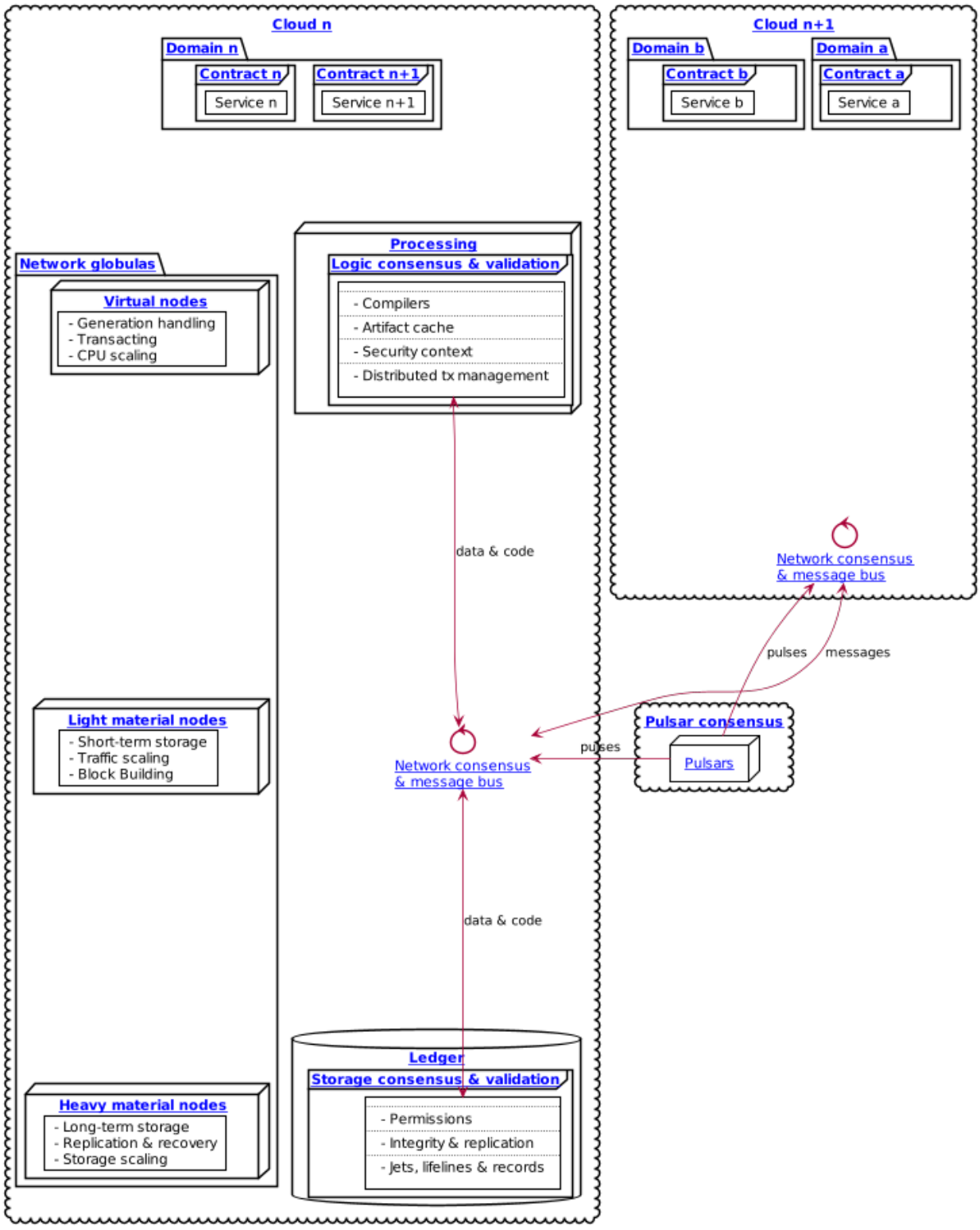
- At the bottom layer, there are providers of *hardware capacities* organized into national and/or industrial compute & storage resources.

Let's take a closer look at the *three bottom layers* since they are the Insolar's development focus.

3.1 Architecture Diagram

Below is the platform architecture diagram aimed to address the aforementioned interconnected layers. The architecture has multiple components and consensuses to address the complexity and variety of requirements.

Components in the diagram are *clickable*, the links will lead you to respective definitions.



All components communicate via messaging to achieve respective *consensuses* and use *pulses* to stay in sync. Let's decompose the architecture to learn the key design concepts.

3.2 Clouds and Their Federations

Clouds organize and unify software capabilities, hardware capacities, and the financial and legal liability of *nodes* to ensure seamless operation of business services. The Insolar Platform transparently connects multiple clouds and each cloud is governed independently, e.g., by a community, company, industry consortia, or national agency. Thus, multiple clouds can unite into a federation on the Insolar network.

The cloud itself establishes governance of both network operations and business logic. Therefore, it is a dual entity that controls:

- The *network* and components deployed during *node* setup, such as:
  - bootstrap configuration;
  - globula discovery and split-protection protocols;
  - node activation and deactivation protocols with the list of currently active nodes and blacklisted ones;
  - real-time detection protocols of execution fraud.
- A special *domain* that is stored by the cloud itself and carries rigid configuration and rules such as:
  - procedures for registering and deregistering nodes;
  - postexecution fraud detection procedures;
  - compensation and penalization procedures;
  - marketplace rules for processing capacity.

3.3 Domains

Domains establish governance of contracts and nodes, thus, acting as *super contracts* that can contain *objects* and their history (*lifelines*) and can apply varying policies to the lifelines contained within. Policies can differ with regards to particular rules:

- Changing the domain itself.
- Access to/from other domains for lifelines.
- Logic validation, e.g., consensus, number of voters.
- Code mutability – possibility of changing the code and change procedures.
- Mutability of object history contained in the lifeline. These rules allow to implement GDPR or legal action via authorization requirements defined by the domain.
- Applicability of custom cryptography schemes requested from the cloud that deploys them.

3.4 Globulas

Globula is a network of up to 1,000 *nodes*. It can run as a truly decentralized network with consistency established by a leaderless, pure BFT-based consensus mechanism, a *globula network protocol*.

Insolar also supports larger node networks of up to 100 globulas (a total of 100,000 nodes) that behave transparently across such networks in accordance with whichever contract logic is in place. Such networks rely on the *inter-globula network protocol* with leader-based consensus.

3.5 Multi-Role Nodes

Insolar utilizes a multi-role model for *nodes*: each node has a single *static role* that defines its primary purpose and a set of *dynamically assigned roles*. Dynamic role allocation functions enable the *omni-scaling* feature of the Insolar Platform.

3.5.1 Static Roles

The node's static role defines what kind of resource and functionality are delivered by that node to the network, and how the network uses such nodes. The network recognizes four static role categories:

- *virtual* – performs calculations;
- *light material* – performs short-term data storage and network trafficking;
- *heavy material* – performs long-term data storage;
- *neutral* – participates in the network consensus (not in the workload distribution) and has at least one utility role.

Static role correlates with the type of resource the node can provide to the cloud, and is a part of the *omni-scaling* feature of the Insolar Platform. All static role categories are detailed below.

Neutral nodes

Neutral nodes participate in the *network consensus* but do not receive any workload automatically distributed by the Insolar network. Neutral nodes serve particular functions:

- API exposure,
- block explorer support,
- discovery support,
- key management.

Virtual nodes

Virtual nodes are stateless, fast, easy to join and leave, and do not need data recovery. On the Insolar network, virtual nodes do the following:

- receive and handle requests to execute contracts;
- *execute and validate contracts*;
- read the latest *contract* state and generate updates (i.e., new *records*) for material nodes;
- enable CPU scalability;
- handle contract-related data encryption when provided with access to relevant key storages.

Light material nodes

Light material nodes are stateful and they automatically collect hot data and indices upon restart. On the Insolar network, light material nodes do the following:

- build blocks;
- manage data access and do audit;
- provide caching for recent data;
- enable scalability of network throughput;
- perform data retrieval and storage operations for *virtual nodes*;
- redirect requests to relevant material nodes when the required data is not available;
- maintain indices of the most recent records, attribute indices, and other functions;
- deduplicate and recover requests in case of virtual node failures;
- assist *heavy material nodes* by serving as temporary backup and cache for individual blocks;
- serve as integrity validators, recovery sources, proof-of-storage approvers, and handover voters;
- collect and register *dust* (e.g., service inconsistency reports, long operations, logs).

Although light nodes can add dust, in case of *lifelines*, they can only add records on behalf of relevant *virtual nodes*. This is enforced by signatures and their checks during new *block validations*.

Heavy material nodes

Heavy material nodes are stateful and require recovery and content revalidation (proof-of-storage), both periodically and upon rejoining the network. On the Insolar network, heavy material nodes do the following:

- provide long-term data storage and scalability of storage capacity;
- store all data received from *light material nodes* (and, in turn, from *virtual nodes*);

- check data integrity but are unable to introduce or change data or form a block;
- ensure the required level of block replication and the maximum data density (scattering) to reduce the impact of data leakage from a single material node (heavy or light).

Heavy material nodes differ significantly from other nodes – they store lots of data and must take additional measures to mitigate the following risks:

- losing (or corrupting) data but not having enough copies, or
- data leakage caused by the accumulation of too much data on a single node.

Heavy material node's implementation is simplified for the TestNet 1.1 and will gradually extend during the development of Insolar's enterprise version.

Moreover, additional network protocol is implemented to maintain backups and archival storage nodes without burdening the main Insolar network consensus.

3.5.2 Dynamic Roles

In addition to the node's static role, it can be equipped with dynamic ones – roles able to change.

Virtual nodes can have the following roles and respective responsibilities:

- **Virtual executor** handles operations on a *lifeline* and builds new *object* states.
- **Virtual validator** verifies virtual executor's actions from previous *pulses*.

Light material nodes can have the following roles and respective responsibilities:

- **Material executor** forms new *blocks* and grants access to previous blocks.
- **Material validator** checks the block's validity and consistency.
- **Material stash** caches hot data and relevant indices (current states of all *objects*) and syncs the indices among other stash nodes.

In essence, all the nodes take part in two kinds of *execution and validation* procedures, depending on their dynamic roles: **virtual** and **material**. *Heavy material nodes* rely on validation performed by light material ones.

A node can have multiple dynamic roles, e.g., a virtual node can be selected via the *entropy* to be an executor for one *lifeline* and a validator of another.

Dynamic roles are designed to:

- enable dynamic and straightforward scaling of the network;
- require minimal preparation to become operational;
- get new workload allocations while dynamic roles of all the nodes change with every *pulse*.

3.5.3 Delegated and Utility Roles

In addition to static and dynamic roles, nodes can take on delegated and utility roles that serve additional functions: caching, inter-globula coordination, and node joining.

3.6 Contracts

The Insolar's main principle is that everything is a *contract* on the Insolar Platform. Contracts are stored as *lifelines* in the *ledger* and are based on general-purpose programming languages such as Golang or Java. They allow existing practices, libraries, and development environments to be used straightforwardly.

A contract developer may focus solely on the contract logic and calls of other contracts, while such details as location & implementation of other contracts are managed transparently by the platform. Every contract has *domain-level* managed rules that define the contracts handling:

- policies for code updates,
- validation requirements,
- inbound or outbound call permissions.

In addition to *governance* with logical rules, domains can also be deployed in separate *clouds* for stronger network security and data inspection on network edges, while contract/business logic can dynamically tune validation performed by the Insolar Platform to balance **costs**, **risks**, and **performance** by adjusting *quantity* and *quality* (stake or liability levels) of *validators* involved.

Contracts also have individual time tracking and resources which can be subsequently connected to custom billing procedures and prepaid (or on-spot) allocation of *hardware capacities*. Moreover, the *ledger* that stores contract data applies strict controls on the following:

- Data access by requiring signatures from *nodes* that need the access;
- Scattering of versioned data across multiple *storage nodes* to significantly reduce risks of fraud, intrusions, or data leaks.

Furthermore, Insolar guarantees to execute any contract and ensures duplicate calls will not emerge in case of hardware, system, or network failure.

For practical enterprise use, Insolar contracts can store and transfer large data *objects* with the following benefits:

- on-chain, without the need for additional systems integrations;
- with algorithms to provide *network traffic*, *CPU*, and *storage* scalabilities.

3.6.1 Contract Determinism

As the platform already reduces determinism via network messaging, Insolar applies relatively relaxed requirements regarding the determinism of *contracts*. As such, a method invocation:

- on the same *object* state,
- with the same parameters,
- and on the same *pulse*;

Should:

- produce exactly the same results,

- consume roughly the same amount of *CPU resources*.

Contract execution methods that run longer than one full pulse must be explicitly declared with an *execution duration* policy.

A contract that does not produce the same results under given conditions will not pass *validation*. In this case, all expended efforts will be at the cost of the party that deploys the contract (as opposed to the caller). Insolar records information on spent efforts in *sidelines* and can track assigned limits, however, the actual billing and payment execution must be handled by *governance logic* (i.e., by other contracts).

Although *virtual nodes* are used to isolate contracts incompatible with security or governance rules, the new contract's code can only be introduced to Insolar as source code, with compilation and static inspection performed by *nodes* in accordance with an applicable *governance model*.

To provide contract execution determinism, Insolar utilizes its *network consistency*.

3.6.2 Network Consistency

Insolar uses the *network layer* to ensure view consistency across the whole network. The next step is to facilitate the efficient and secure execution of contracts across all *virtual nodes*.

To this end, Insolar:

- *sets apart the functionality* requiring different resources and permissions,
- distributes workloads across all available/active nodes of the Insolar network using entropy.

As a result, all nodes have:

- the same *entropy* value,
- a list of active *nodes*.

Insolar does not use node workload statistics to provide network consistency, instead, it implements pseudo-random workload distribution.

The reason is simple: a trustful workload factor in distributed systems requires full visibility and operations aggregation but they still do not guarantee smooth workload distribution when workloads fluctuate faster than the average duration of a workload control cycle (aggregate statistics – balance – execute).

Pseudo-random workload distribution can cause distribution anomalies within a workload control cycle but it provides a relatively smooth distribution on longer timescales, without the need for full visibility and operations aggregation.

Such a workload distribution and the entorpy-based allocation functions for *dynamic roles* are the core instruments that enable the *omni-scaling* feature of the Insolar Platform. This feature provides a balance in accordance with client's needs.

Processing costs can be traded off against:

- **Uninsured risks**. Suitable for situations where a cheaper transaction is executed but fewer validators verify said transaction, meaning greater risk of loss.

- **Processing speed**. It can be increased to the detriment of operational risk:
  - frequent transactions could be processed without awaiting validation, or
  - validations may be batched together and processed following some delay, leading to the possibility of resource-consuming rollbacks.

3.7 Execution & Validation

The Insolar Platform works on the principle of actions executed by one node, validated by many.

The number of selected validators can be determined in accordance with the *business process* at hand and, since validators in shared enterprise networks will have liability and legal guarantees, this works as transaction insurance.

As described in the *network consistency section*, validator selections are *not* based on voting; instead, they are part of the *omni-scaling* feature. Insolar uses the active node list and *entropy* generated by consensus of the *globula network protocol*, and then applies deterministic allocation functions for *node roles*. This avoids wasting efforts on numerous per-transaction and network-wide consensuses.

Since Insolar sets apart functionality using *node roles*, it has two sets of execution & validation procedures: **virtual** and **material**.

3.7.1 Virtual Execution & Validation

Nodes with *virtual static roles* carry out **virtual** execution & validation:

1. The network selects (determines based on *entropy*) a specific virtual node to become a *virtual executor*. Upon receiving the request, the executor:
   1. Registers the request within the current *pulse*.

      In case the request arrives to a 'busy' virtual executor, it can delegate the execution of an *object* to other virtual nodes (not necessary to virtual executors). Moreover, multiple requests can be executed within the same pulse when opportunistic execution/validation is allowed by the caller or by the called object.
   2. Executes the request on the *object* (contract).
   3. Collects the results of outbound calls.
   4. Provides *lifeline* and *sideline* updates for validation by other nodes.
2. Once the executor's status expires, the network selects *virtual validators* from the list of active *virtual nodes* on a new *pulse* (new entropy), meaning executors cannot predict which nodes will validate transactions, thereby avoiding a collusion scenario.
3. Each virtual validator:
   1. Checks that the request is legitimate.
   2. Executes the request on the *object* (contract) a second time.
   3. Checks that the request returns the same response given the *same arguments*.

    4. Checks that the request performs the same outbound calls.

4. Lastly, the outbound calls validation is stacked into a single validation round as validators use signed results collected by previous executors.

A single virtual executor can execute long requests that span several pulses. To do this, the virtual node that started the execution asks current executors in each pulse for tokens that give the execution permission.

3.7.2 Material Execution & Validation

Nodes with *light material static roles* carry out **material** execution & validation:

1. The network selects (determines based on *entropy*) a specific light material node to become a *light material executor*. Upon receiving data requests from the virtual executor in the current *pulse*, the light material executor:

    1. Manages data access for *contracts*.
    2. Performs data retrieval and storage operations for *virtual executors*.
    3. Builds a new *block* from the *lifeline* & *sideline* updates sent by the virtual executor.
    4. Splits (or merges) *jets* if required.

2. Once the executor's status expires, the network selects *material validators* from the list of active *light material nodes* on a new *pulse* (new entropy), meaning executors cannot predict which nodes will validate transactions, thereby avoiding a collusion scenario.
3. Each material validator checks that the light material executor has formed the last *block* correctly. The block must have:

    - Correct hashes.
    - Correct order of new *records* in the affected *filaments*.
    - No contradictions between records in the filaments.

    In addition, each validator ensures that the executor made the right decision to split (or merge) the corresponding *jet*.

Upon each pulse, every light material node sends the data they formed to *heavy material nodes*. However, light nodes keep hot data and share hot indices among a number of *light material stash* nodes.

Light material stash nodes are nodes which have been *light material executors* for a number of past *pulses*. The number is called a *stash history limit* and its default value is 5 but it is configurable within a *cloud*. Thus, stash material nodes provide caching for recent data.

3.8 Consensuses

Consensus procedures vary in their degree of control by business logic, with two consensus procedures available:

- **Domain-defined consensus**: procedures that are a set of Raft-like protocols with *entropy-controlled* voter selection. These protocols are applied to an *object* after a series of changes. Such protocols can be chosen at the *domain* level and configured at the transaction level.
- **Utility consensus**: procedures – a set of protocols – that cover various platform operations not directly operated or required by business logic, including network consensus, pulsar consensus, and traffic cascade.

Different sets of consensus procedures affect every action applied to *lifelines*: *logic*, *storage*, *network*, and *pulsar* consensuses.

3.8.1 Logic Consensus

Ensures that actions applied to an *object* were performed correctly considering the object's state, input parameters, and external dependencies (calls).

For more information on logic consensus, see the *virtual execution & validation section*.

3.8.2 Storage Consensus

Ensures that:

1. *Nodes* which participated in logical consensus had allocated roles.
2. *Records* generated by the nodes are structurally and referentially valid.

For more information on storage consensus, see the *material execution & validation section*.

3.8.3 Network Consensus

Ensures *node* availability and synchronization of time and state among nodes and provides consistent allocation of *dynamic roles* to nodes. There are two consensus protocols behind the network consensus:

- **Globula network protocol**: a truly decentralized BFT-like protocol without any consensus leader that establishes the consistency of a globula (a smaller network of up to 1,000 nodes).
- **Inter-globula network protocol**: a leader-based protocol that extends the GNP and establishes consistency among globulas of the Insolar network (up to 100 globulas or 100,000 nodes).

The network layer of Insolar deals with the consistency of network node's view and *pulse* distribution. Pulse is a signal carrying entropy (randomness) that triggers the production of a new *block*.

The entropy's consistency and the set of active nodes on the network are vital for the methodology of executed by one node, validated by many. Nodes are selected from the active node list to perform *different functions*, while entropy and consistency ensure behavioral consensus across all nodes. *Validator* nodes are selected only on a new pulse to ensure that *executor* nodes cannot collude with validators.

In addition to the aforementioned consensuses, *pulsars* can have their *own*.

3.9 Pulsars

Pulsars running on a pulsar protocol represent a separate logical layer that is responsible for network synchronization and provides a source of randomness (*pulses*). Interoperability of *nodes* within a single *cloud* depends on pulses and all nodes must be on the same pulse to process new requests or operations.

Pulsars can run either on the same network or an entirely separate one. Cases of the former include:

- private networks that can implement a dedicated server;
- cross-enterprise and hybrid networks that can use a shared network of pulsars yet run individual installations of Insolar networks;
- and public networks that can use trusted pulsar nodes or run the pulsar function on other nodes.

In case of multiple pulsars on the network, their consensus generates the *pulses*.

3.9.1 Pulsar Consensus

Clouds define the pulsar selection rules and they can vary significantly. On enterprise networks, servers that complete no other operations manage the selection, whereas on public networks, it may be a random subset of 10 to 50 nodes with high uptime. Other configurations are also possible for different network types.

Default *pulse* generation is based on BFT-consensus among pulsars, where *each member contributes* to entropy and *none can predict it*. The pulsar protocol enables entropy generation in a way that prevents individual nodes from being able to predictably manipulate the entropy through vote withdrawals.

This protocol does not include negotiations related to pulsar membership or pulse duration – such parameters are considered as preconfigured or preagreed. The default pulse duration is 10 seconds.

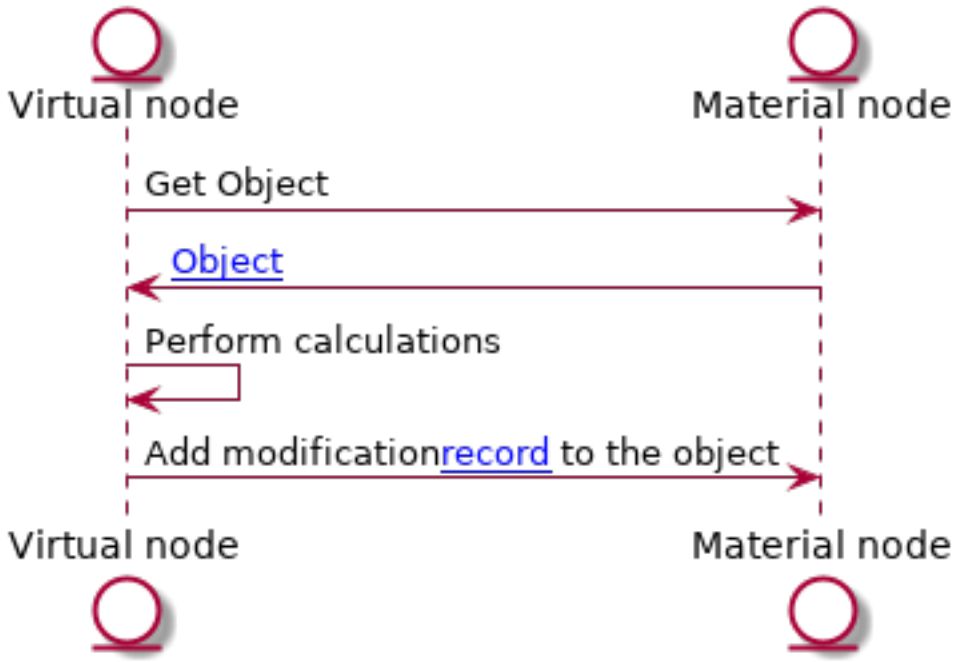
As a consensus result, pulsars distribute the collaboratively-generated entropy signed by every pulsar to every node on the network.

3.10 Ledger

Ledger is a common term for distributed storage, a network of nodes that store data.

As described in the *static roles section*, material nodes are responsible for storing data and providing it on requests for *virtual nodes*. Virtual nodes create and sign new information and pass it to material nodes to store. So, material nodes do not create or modify information (*objects*) with the exception of specifically defined meta data.

A typical *object* workflow is as follows:



3.10.1 Records

Data is stored in the ledger as a series of immutable *records*. All records are created and signed by *virtual nodes*. Each record is addressed by its hash and a *pulse* number. Records can contain a reference to another record, thus, creating a chain. An example of a chain is the *object's lifeline*. Each *material node* is responsible for its own lifelines determined by their hashes.

In the Insolar's key-value storage, the key is a fixed structure – a combination of a pulse number and a value hash. The value can be one of several types:

- *Record* – immutable structured data unit. Can form chains if each record references a previous one in succession.
- Index – meta information about record chains, e.g., pointers to the latest record in a chain. Represents an *object*.
- Blob – immutable payload. Used to store (potentially big) chunks of serialized data, e.g., object's memory. Usually, records refer to blobs to store application data.

3.10.2 Requests

Each operation performed by *virtual nodes* is registered as a request in the ledger. Request is a single *record* that contains information necessary to perform an operation. Each request belongs to an *object* and is affined to it.

3.10.3 Results

Each operation performed by *virtual nodes* has exactly one result. Although an operation can have many side effects (*records* stored in the ledger), result represents a summary of that operation. So, each finished request has its own result, i.e., result references its request. A request without an associated result stored in the ledger is a *pending* one.

3.10.4 Objects

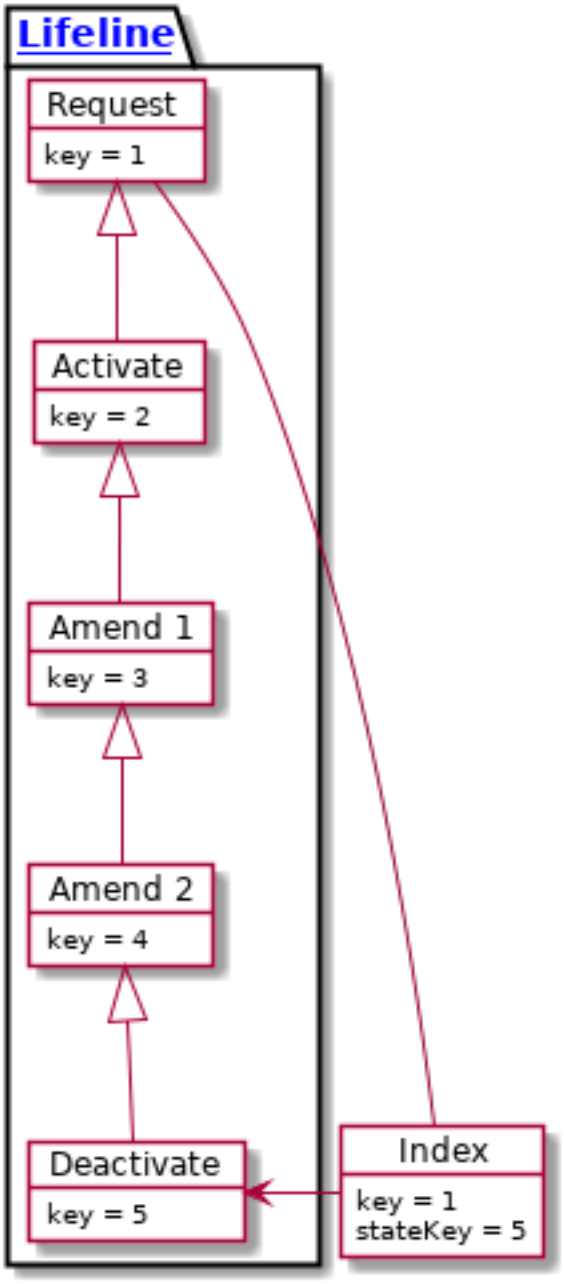
Objects (contracts) are fundamental application building blocks. Borrowing OOP terminology, an object is a class instance. In other words, an object is a series of *records* that can be accessed via an index.

Each record represents an object's state at a certain point. The state can contain the object's memory at the point. Memory is a binary blob stored in the ledger and a contract can put any data it needs into it.

In a blockchain, objects cannot be modified, only appended by another record. Therefore, object states can be one of the following types:

- **Activated** – the *object* has been initialized. This is the first state of any object and it contains initial memory.
- **Amended** – the object's memory has been modified. Contains new memory.
- **Deactivated** – the object has been "removed" from the system. Since data cannot be removed from the chain, objects are simply marked as *removed*.

A succession of object records (states) is called a *lifeline*:

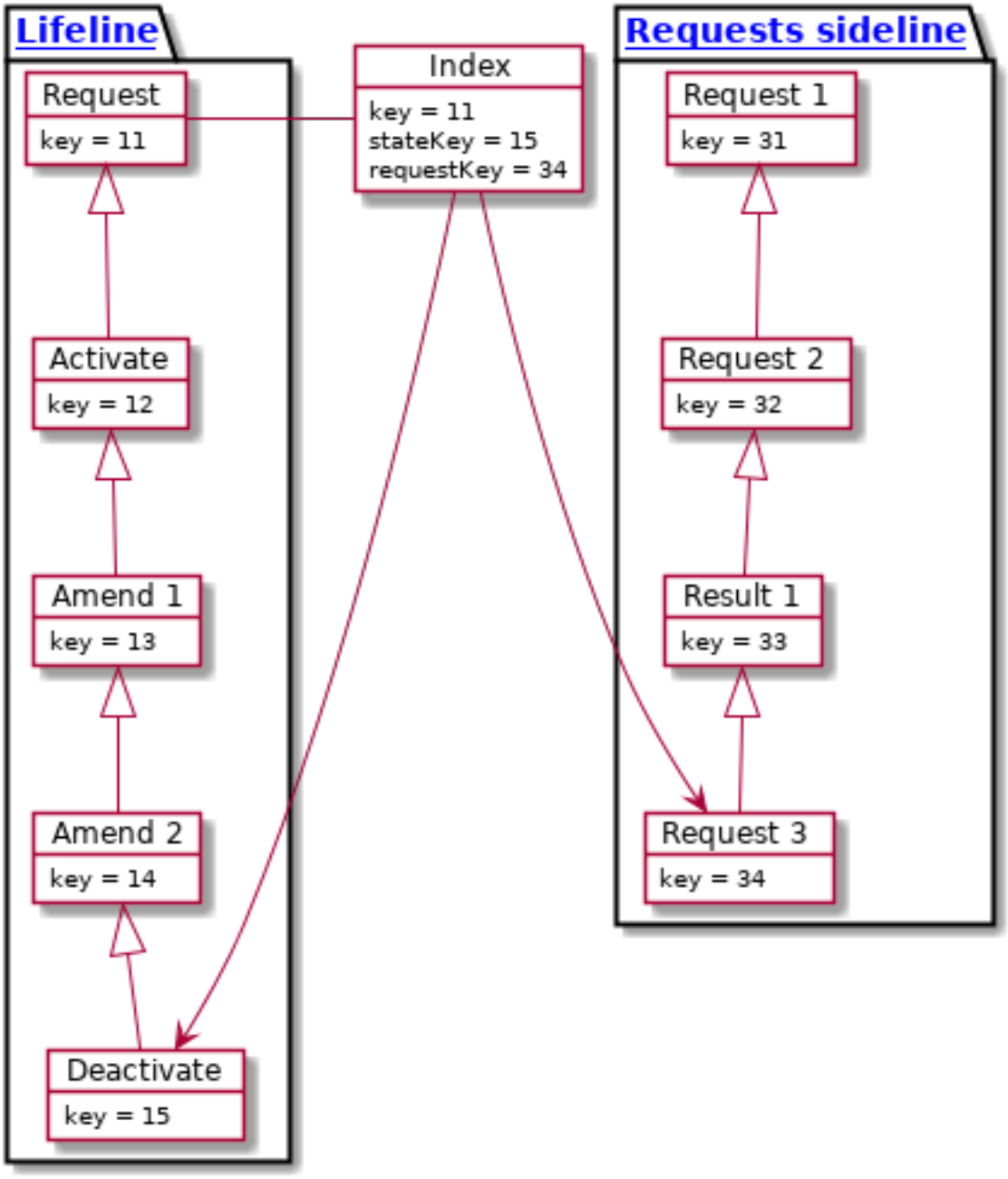


An object is assembled from a lifeline via its index. As stated above, index is a collection of pointers to object's records (states, requests, etc.). So, to get an object, all we need is its index. The ledger stores multiple versions of the object's index depending on the pulse.

To preserve consistency, each operation is performed on a particular object's version. To get an object to execute on, a *virtual node* sends an operation request based on which the object's version is calculated. This way, two concurrent operations can be performed on different versions of said object.

Object's lifeline is not the only chain, though. The ledger stores any requests that belong to an object in a *sideline*. The general term for all the chains (lines) is a *filament*. So, a more complex

object structure including all filaments is as follows:



Object's Address

Object's address is more complicated than that of a simple *record*. An *object* consists of many *records* but should have only one address. So, the ledger considers the address to be a pointer to

the creation request's record. The object's index can be found via this address.

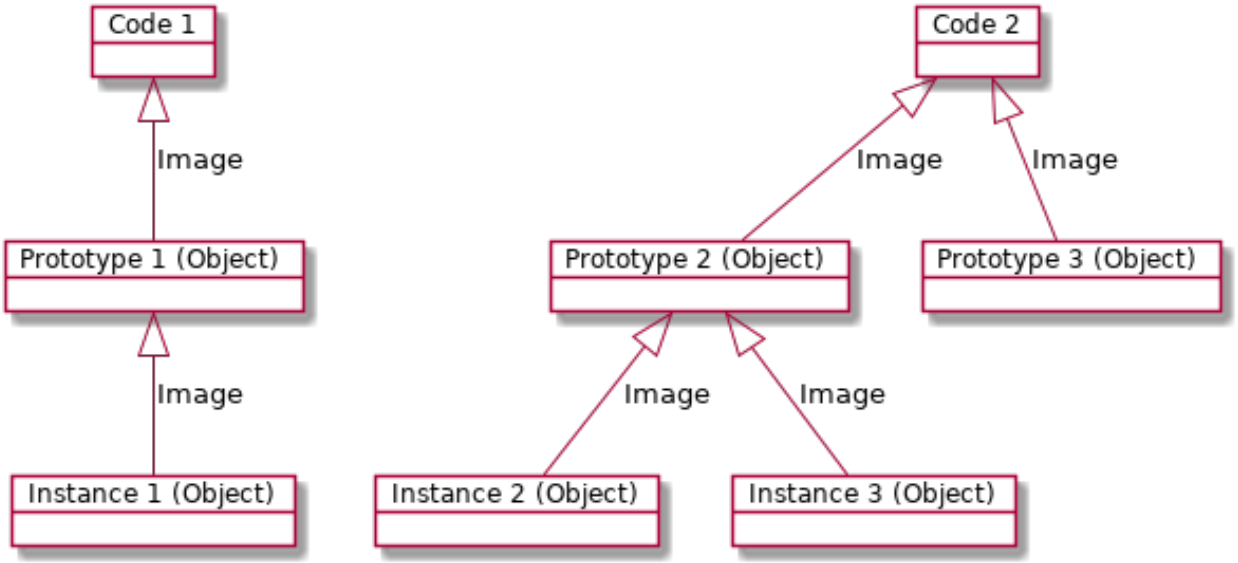
3.10.5 Relations

Objects have relations to other entities and to each other. Most of those relations are references in the object's *activation record*.

Key figures in those relations are:

- **Object**. Directly references a prototype. This reference cannot be changed during the object's lifetime, although multiple objects can have the same prototype. Serves as an *instance* of a prototype.
- **Prototype**. Special kind of *object* that acts as a template for building other objects. It contains default memory and directly refers to relevant code.
- **Code**. Single immutable *record* which contains code for *virtual nodes* to execute. They perform operations on the referenced object. The same code can be referenced by multiple prototypes.

Relations between the entities are as follows:



Since both prototype and object are technically *objects*, they contain a reference to either:

- prototype in case of an object, or
- code in case of a prototype.

The general term for this reference is an *image*. In other words, object's image is its prototype, and prototype's image is its code.

CHAPTER

FOUR

GLOSSARY

Below is a list of terms (and their analogies) Insolar uses to designate its key concepts.

Pulse Tick that opens up a new block and is an event that ends the current time *slot* and starts a new one. Every pulse carries a new entropy/seed. Pulses are numbered in a monotonously increasing sequence.

Slot Time period between two consequent *pulses*. Different slots may have different time duration.

Jet Shardchain that keeps *records* of a subset of *objects* (*lifelines*) contained by a *cloud*. The shardchain has nodes allocated to store related records.

Material Jet *Jet* that has no dependencies on other jets and is meant to provide persistence and access to data.

Virtual Jet Logical group of affined objects for nodes allocated to process requests related to these objects. For example, each *lifeline* is considered an individual virtual jet. Such jets rely on material ones to operate: e.g., material jets store data while virtual ones do various calculations and validations of that data.

Jet Drop Block of a shardchain with adjoined blocks of relevant *sidechains* that keep records produced at a specific *pulse*. Alternatively: Set of *records* representing changes of *lifelines* (and their *sidelines*) in a *jet*, all happened within a *slot*.

Domain Decentralized application (dApp) that governs access, consensus, mutability, and other capabilities for other dApps. Alternatively: Collection of *objects* and related policies (construction, referencing, logical consensus, etc.). Domain also chooses a *cloud* to provide storage and processing for objects. Domain itself is a descendant of an *object*.

Object Smart contract, dApp, application object, addressable element of application logic or data. Resides within a *domain*. Object is stored as a *lifeline*. The first *record* of a lifeline is a permanent identifier of the object. Objects can be of different types and their lifespan is virtually unlimited and usually controlled by or through the object itself or by the object's domain.

Record Similar to a transaction record; Insolar has a variety of record types. Record contains information on request, response, state control, maintenance, etc. All records fall into two main categories depending on their usable lifetime – a period during which the record can be used under normal circumstances:

- *Permanent* records are generated by an application and its business logic. The logic controls the record's usable lifetime, e.g., legal documents must stay for a period of action limitation.
- *Dust* records are associated with a permanent record and are generated either by application or system logic. The usable lifetime of such a record is limited by maintenance procedures and usually measured in days, e.g., logs or transaction control records. Dust can also be used to identify complex forms of fraud or infringements; such will be registered as permanent records and the original dust records will be archived or removed.

Filament Linked sequence of *records* related to an entity (*object*, operation, etc.). Stored as a unidirectional linked list, from older to earlier records. Filament is identified by the reference to its head (the first/earliest record) and every filament's record has an affinity field that refers to the head. Filament is a general term that denotes different sequences of records. For example:

- object's state changes (*lifeline*);
- requests (received, pending, answered, validated);
- listings (delta and full snapshots of child object's listings).

Lifeline Chain of *records* (*filaments*) for a single object with relevant *sidelines* that represent auxiliary information about the object. Alternatively: Lifeline is all history and virtually all future changes of an *object* or of a *dust*. Lifeline never forks and belongs to a single domain. The first record (head) of a lifeline is an activation record and a permanent identifier. The last record (tail) is the latest state. Lifeline contains:

- main filament (object's state);
- additional filaments (states of requests);
- indices, listings, etc., related to the object.

Presence of additional filaments does not mean a fork to the object's state as they carry only additional information.

Sideline Sidechain (*filament*) for auxiliary information such as logs.

Dust Auxiliary object or record; a log record stored within a sidechain.

Dustline Removable *sideline* for a dust record.

Cloud Blockchain network (group of *nodes*) under the same node membership policy. The nodes provide storage and processing methods (including storage consensus) supported by the cloud (e.g., a specific blockchain implementation). Nodes can have different roles within the cloud.

Node Server that serves hardware capacity to a cloud. Node also represents an authorization (e.g., by putting a stake) of a *member* to participate in a *cloud*. Node is both a unique account and a unique address within a cloud. Node is always associated with a single member and with one or more *hosts*. Node performs *only one* role within a cloud but a member can have multiple nodes.

Host Entity of a physical/transport network, e.g., a server. Each host can have multiple network addresses, can be in different networks, and use various transport protocols and encryption standards.

Member External entity (user) registered within a special *domain*.

Network Set of *hosts* sharing same security, reliability, and network performance profiles and able to directly exchange data under at least one transport protocol and at least one encryption standard supported by all hosts. In other words, all hosts within the same network are able to use P2P communications without violation of security and other policies.

CHAPTER

FIVE

INTEGRATING WITH INSOLAR

Note: Upon the MainNet release, this section will be expanded to include the appropriate integration instructions.

To join or set up an Insolar network, check the *hardware requirements* and:

- *Connect to TestNet 1.1*. Participation in this network is permissioned, with participants invited by the Insolar Core Development Team based on their ability to fulfill the respective SLA.
- *Set up a network* locally for development and test purposes. The local setup is done on one computer with no particular system requirements, and the 'network nodes' are simply services listening on different ports.

5.1 Hardware Requirements

The recommended setup for a proof-of-concept private Insolar network is to consist of **at least 5 nodes** that may be deployed both on virtual or physical servers in a data center.

The minimal hardware requirements for all servers are as follows:

Processor	RAM	Storage	Network bandwidth
4 cores (8 recommended)	16 GB	50 GB	1 Gbps

Note: The storage capacity may need to be expanded depending on the size of the data to be stored.

Insolar runs on Linux, e.g., **CentOS**.

5.1.1 TestNet 1.1

Preferable hardware requirements for virtual nodes on TestNet 1.1 are as follows:

Processor	RAM	Storage
40 cores	32 GB	256 GB SSD

All servers wishing to join the Insolar test network must have **public IP addresses**.

5.2 Connecting to Test Network

To connect to Insolar TestNet 1.1:

1. Skim through *known issues and limitations* and check the *TestNet 1.1 hardware requirements*.
2. *Set up and connect* a node.

5.2.1 Known Issues and Limitations

Note: Issues below will be addressed in future releases.

On TestNet 1.1:

Node Maintenance

- Only computational (virtual) nodes are available to external participants. Data storage is provided by Insolar nodes.
- All discovery nodes are hosted by Insolar. Other nodes use them to reconnect to the network.
- In the unlikely event of short-term storage (light material) nodes having to reconnect, multiple errors may occur for a few pulses.
- Only one long-term (heavy material) node and one pulsar are deployed. If either of the nodes is missing, the network will go down but, upon the node's restart, will recover.
- Storage node crash may lead to data loss.
- Under certain conditions, the node's process (`insolard` daemon) may exit and its Docker container will restart it automatically. Insolar may also ask node holders for assistance with a manual restart.
- Nodes joining the network produce errors when other nodes are leaving the network.

Security & data consistency

- Smart contract validation is disabled. Therefore, any execution result returned by a virtual node is treated as verified.
- Distributed transactions are not yet implemented. This can lead to decorrelated object changes. For example, an interrupted 'money' transfer from one wallet to another decreases the source wallet's balance but leaves the target wallet's balance unchanged.

- Operations executed during pulse changes will be declined with the `Incorrect message pulse` error.
- Currently, all the data is stored and transferred unencrypted.
- All network messages are signed but signature checks are disabled.

Performance

- A simplistic rate limiter is implemented for light material nodes, so they reject incoming requests when the number of pending requests reaches a certain limit. This results in an exponential backoff on our benchmark tool – the retry interval increases exponentially.
- The rate limiter does not consider the request's origin. So, when a user puts an excessive load on the network, other users may suffer.

Application level

- Only pre-built smart contracts are available. Custom contracts will be available on TestNet 2.0.
- All user wallets are created with a starting balance of 1,000,000 coins.
- Only one contract can be called via the node's API. All other methods (e.g., coin transfer) are called via the `Call` method on a `member` object.

5.2.2 Connecting to TestNet 1.1

To connect to the Insolar test network, do the following:

1. Install Docker and Docker Compose and run the Docker daemon.
2. Download the Insolar's `insolar-node-<version>.tar.gz` archive from the latest release. You can find it under the *Assets* drop-down list.
3. Unpack the archive on your server. A good place is under the `/opt/insolar` directory.
4. Go to the unpacked directory, open the `docker-compose.yml` file in a text editor, and insert your server's public IP address to the `INSOLARD_TRANSPORT_FIXED_ADDRESS` field.
5. Acquire `cert.json` and `keys.json` files from Insolar. You can ask for them in our Telegram developer's chat.

   Put the files to the `configs` directory.
6. Run `docker-compose up -d`.

Enjoy being a part of the Insolar Network!

Note: The Insolar's API is under development and not yet finalized. Please, await its first release.

In addition to the Insolar node, the Docker Compose starts Kibana and Grafana services to take care of *logging and monitoring*.

5.2.3 Ports Used

Insolar uses the following ports:

Port	Protocol	Description
7900, 7901	TCP, UDP	Nodes intercommunication. The node must be publicly available on these ports.
8090	TCP	Node-pulsar communication. The node must be publicly available on this port.
18181, 18182	TCP	Communication between the main node daemon and the smart contract executor daemon.
19191	TCP	Node's JSON-RPC API.
8080	TCP	Prometheus metrics endpoint.

5.3 Setting Up Network Locally

To set up the network locally, do the following:

1. Since Insolar is written in Go, install its programming tools.

   Note: Make sure the `$GOPATH` environment variable is set.

2. Download the Insolar package:

   ```
   go get github.com/insolar/insolar
   ```

3. Go to the package directory:

   ```
   cd $GOPATH/src/github.com/insolar/insolar
   ```

4. Install dependencies and build binaries: simply run `make`.

5. Take a look at the `scripts/insolard/bootstrap_template.yaml` file. Here, you can find a list of nodes to be launched. In local setup, the 'nodes' are simply services listening on different ports.

   To add more nodes to the 'network', uncomment some.

6. Run the launcher:

   ```
   scripts/insolard/launchnet.sh -g
   ```

   The launcher generates bootstrap data, starts the nodes and a pulse watcher, and logs events to `.artifacts/launchnet/logs`.

When the pulse watcher says `INSOLAR STATE: READY`, the network is up and has achieved consensus. You can start running test scripts and benchmarks.

Also, you can manually bring up *logging and monitoring* by running `scripts/monitor.sh`.

5.4 Logging and Monitoring

To see the node's logs, open Kibana in a web browser (`http://<your_server_IP>:5601/`) and click *Discover* in the menu.

To see the monitoring dashboard, open `http://<your_server_IP>:3000/`, log in to Grafana (login: `admin`, password: `pass`), click *Home*, and open the *Insolar Dashboard*.

INDEX