Public Version of the Research Plan for Distributed Decentralized Blockchain-based Storage Platform

JSC "NEO Saint Petersburg Competence Center"

February 17, 2019

Abstract

Distributed Decentralized Storage Platform (DDSP) is a distributed decentralized object storage integrated with the Neo Blockchain. It is intended to be primarily used by Decentralized Applications (DApps) as a data storage and a Content Delivery Network (CDN). We propose using smart contracts to control the distribution of rewards from data owners and publishers between participants hosting data.

The first novelty of this research is approach based on homomorphic hash signatures to prove data integrity and availability to minimize load on the network and avoid transferring real data over the network in order to conduct data audit and validation.

The second novelty is a scalable data placement method for *DDSP*. Fine control over object location and minimal data movement in case of storage nodes failures are achieved by using a subset of a network map and storage policy rules for object placement and Rendezvous hashing for node selection.

Contents

1 Motivation

The development of blockchain technology has recently moved not so much towards global public permissionless blockchains as towards the implementation of corporate- and state use to solve specific internal tasks. It is logical to expect the development of Dapp projects in this direction. However, existing decentralized data storage solutions have not yet been fully prepared to meet an emerging need for reliable, trusted data storages with a secure and controlled integration of corporate- and public parts.

Currently, most projects in this field are aimed at implementing simple exclusive storages of data of some users on capacities of other users or at creating add-ons via IPFS for implementing public content distribution based on traditional hosting and CDN. In this case, the niche of storages that have a convenient API for DApp and the ability to organize isolated areas with control over data exchange in both public- and other private data storage areas is practically empty.

It is proposed to solve this problem relying on the Neo Blockchain mechanisms as well as implementing novel approaches and technologies.

DDSP is based on a peer-to-peer network that use information from a smart contract and the Neo Blockchain to coordinate data operations. X.509 PKI Neo Identity is planned to be applied to form isolated storage networks and provide controlled access to data.

A storage network keeps information about its own topology with a history of changes up to date and uses it when placing and searching for data. The applied set of algorithms allows to minimize data movement in the case of changing a network map, while ensuring the specified storage policies.

DDSP is aimed at ensuring data availability and integrity. The integrity guarantee is achieved by using the proposed novel data audit method, which is based on Homomorphic Hash functions, and allows to verify the integrity of data on a storage node without sending real data to a verifying party over a network. A computational complexity of verifications depends linearly on the size of objects and is well suited for parallelization. This approach not only reduces load on a network and storage nodes but also allows to avoid data disclosure to a third party during a verification phase.

At a lower level, *DDSP* provides backward compatibility of a storage format with future versions of the system. This is important not only for corporate use but also for DApp and smart contracts that do not have a technical ability to modify codes or reallocate data in a network.

Thus, the proposed implementation of *DDSP* should ensure that the need of trusted data storages is met for both public use and various private networks, while providing necessary guarantees of information confidentiality, integrity and availability.

2 System overview

2.1 Glossary

- *Network map* is a hierarchical structure in the form of a graph, describing available storage nodes
- *Epoch* is a time period during which a permanent network map exists
- *Homomorphic hash* is a hash resulting from applying homomorphic hashing to an object; it is used for zero-knowledge proof
- *Data audit* is data accessibility and storage validation process

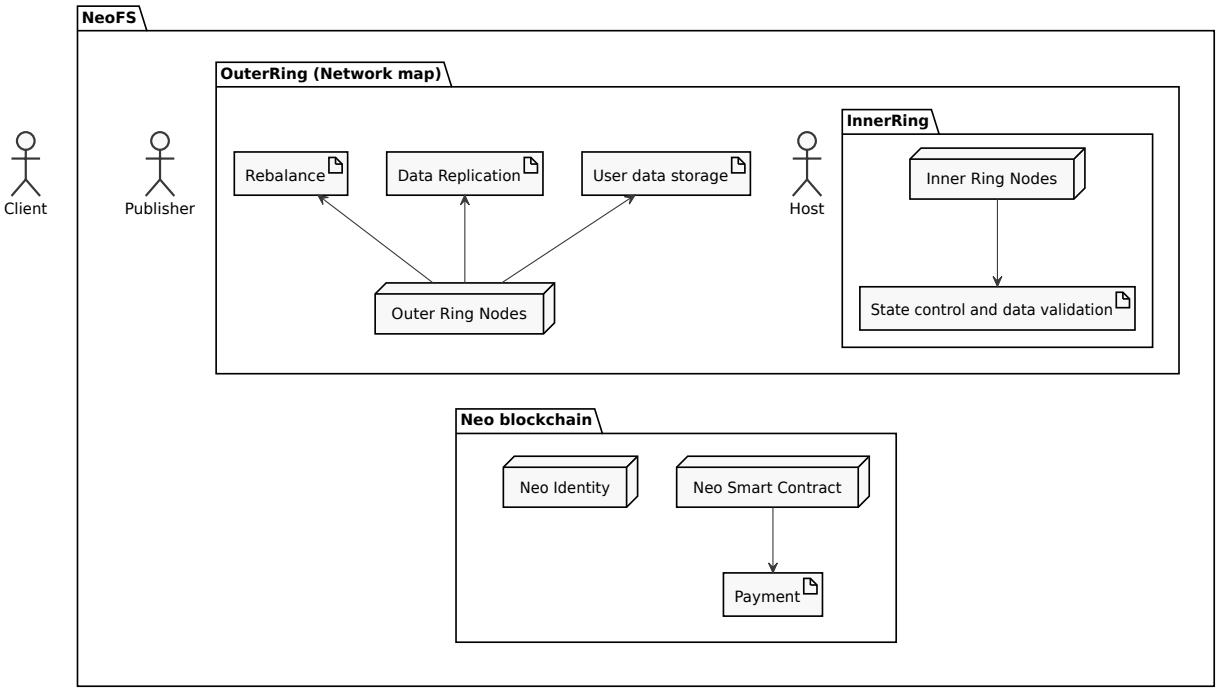

Figure 1: System overview

2.2 Node Roles and Terms

- *Inner Ring* – network nodes executing a network state control and data validation
- *Outer Ring* – all other network nodes

Regardless of whether a node is in *Inner* or *Outer Ring*, it can perform one or more of the following roles:

- *Host* is a computing system being a part of *DDSP* and having at least the following features:
  - recent version of *DDSP* software;
  - ID assigned by agreement with other nodes and unique within *DDSP*;
  - acceptable quality of communication channel with other *DDSP* nodes;
  - storing and processing service/technical data of *DDSP*, using its own capacity;
  - storing data according to parameters set by the system;
  - receiving and handling client queries according to the protocol/API used in *DDSP*.
- *Publisher* is a computing system being a part of *DDSP* only in an incentive model, not being a part of a network map, related to a specific account of *DDSP* (accounting) and interacting with the system via a specific protocol/API. *Publisher* can get data from the system, place it for storing and receive responses to queries according to protocol/API used in *DDSP*.

- *Client* is a computing system not being a part of *DDSP*, interacting with the system via a specific protocol/API. *Client* can receive data from the system, but can not place it for storing. It can also receive responses to queries according to protocol/API used in *DDSP*.

3 Network structure

DDSP is a p2p network which consists of *Inner* (nodes, specially selected by the network itself) and *Outer Rings* (all other network participants). Only nodes that provide storage space for objects are included in the network map.

DDSP is based on *Inner Ring* nodes that store information about the network state, metadata and state changes. In fact, it is an append-only sharded database in the form of a distributed operations log with signed snapshots at certain points in time.

For a consistent update of a distributed log, a consensus between nodes of *Inner Ring* is achieved using the dBFT protocol. A number of nodes does not change during one epoch of the network map. Nodes are admitted to *Inner Ring* on the basis of X.509 certificates and signatures in the Neo Identity trust network or another X.509 PKI. The decision to remove a node from the ring is made based on its inaccessibility or in case it is compromised. The usage of X.509 PKI allows to avoid re-entering *Inner Ring* by nodes with a poor reputation or unsatisfying technical requirements.

Information from *Inner Ring* is distributed to *Outer Ring* via Gossip protocol.

Inner Ring takes a decision on payments for storing data. A list of *Inner Ring* nodes is given in a smart contract. A candidate node from *Outer Ring* nodes pays a deposit that amounts to the cost of including/excluding a node in/from the list in the smart contract to join queue for being added to *Inner Ring*. When new nodes need to join *Inner Ring* in order to substitute non-available, discredited ones or increase a number of nodes to support the network scalability, a node is selected from the ones being in the queue. *Inner Ring* nodes that have implemented correct data validations and taken decisions on payments for data storing earn a low interest on the paid amount.

3.1 Bootstrapping

Inner Ring nodes perform functions of bootstrap nodes to initiate the work of *Publisher*, *Host* and *Client* as well. The bootstrap node returns a current state of event log (in general, provides the procedure of retrieving a network map), therefore, providing nodes and clients being connected with operations of storing and retrieving data. The bootstrap node registers information on a *Host* node to include it into a network map of a succeeding epoch.

The bootstrap node may refuse a node registration or response. Thus, it is important to know about all available nodes of *Inner Ring*. A current list of *Inner Ring* nodes can be achieved via a smart contract call and is available on open resources.

3.2 Network Map

DDSP keeps a network map up to date and distributes it over nodes. It contains information about groups of nodes, their location in the network and main parameters necessary for correct data search and placement. The network map is a hierarchical structure that provides available storage nodes. The network map is represented as a graph. The graph consists of vertices: buckets and nodes. A bucket is a vertex of the graph. Together with outbound edges, it forms a subgraph of the network map, leaves of which are represented by nodes. Nodes can only be leaves, the bucket can be a parent for other buckets or nodes. The bucket is characterized by type and value.

3.3 Buckets

In a decentralized system, it is impossible to fully control the correctness of metadata provided by a storage node. Therefore, it is proposed to use trustworthy buckets in a network map, i.e. the buckets that *Inner Ring* can form on its own, for example, ranking by GeoIP, AS, a confidence coefficient. It is also assumed that a storage node can independently transmit information about buckets in which it is located – self-defined buckets. A user is responsible for setting placement rules by using self-defined buckets.

3.4 Distributed Log

A distributed event log stores information about changes in the system state, intermediate calculation data and other technical information necessary for the operation of *DDSP*. Each entry in the log is consistent and signed by the nodes of *Inner Ring*.

At epoch changes, a new network map (or difference with the previous version) is committed in the event log.

Each record in the log has a unique, monotonically increasing numeric identifier. Periodically, the tail of the log is discarded and replaced with a snapshot of the state.

Each node of a network records the last log position it knows and can synchronize its state after loosing communication by reading all subsequent blocks from the log. A new node, when it is connected to the network, needs to get the most recent snapshot of the state.

4 Data Placement

4.1 Placement Function and Placement Rule

To get a subgraph of a network map or a subset of nodes where objects are stored, the one needs to use a placement function that has the following arguments:

- storage policy defined with a placement rule;
- a network map (or a network map subgraph);
- Rendezvous hashing salt.

A user can define a placement rule that is applied to a stored object. The placement rule consists of a set of SELECT() or FILTER() operations applied to a network map. The result of these operations is a subgraph of the network map where data can be placed.

The SELECT() operation is applied to a tree. The operation inputs are a replication factor at this level and a bucket type. Multiple operations in the placement rule are put into order and each subsequent SELECT(r, type) operation is applied to the result of the previous one.

The FILTER() operation is applied to a graph. The operation inputs are a bucket type, a bucket value and a comparison operation. For text values, operations *eq*, *ne* are available. For numerical values, *gt*, *ge*, *lt* and *le* are additionally available.

A set of operations on the graph (in the placement rule) can be combined by using AND, OR, NOT operations.

The placement function is executed recursively with the operation of the next step being applied for all the nodes retrieved at the previous step.

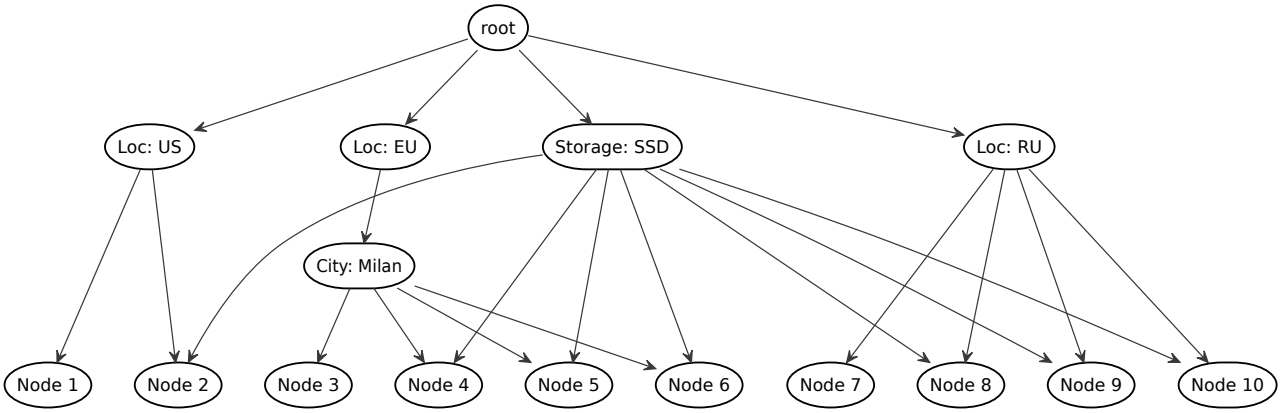

Figure 2: Network map example

This network map (or a network map subgarph) consists of the bucket with types Loc (country), City (city), Storage (storage disk type) and the nodes of corresponding buckets. The storage policy can be defined as follows:

- it is stored in 2 different countries;
- it has 3 copies per each country;
- it should be stored on SSD drives only.

This informal description should be represented as sets of SELECT(r, type) and FILTER(type:value, op) operations. A total replication factor is obtained by multiplying all the factors of each SELECT(r, type) replication. A final statement should always be SELECT(X, Node).

Thus, the subgraph of the network map and six nodes have been obtained, where the object should be placed on to meet the placement policy requirements. The result of the placement function operation is a subgraph of the network map – *Placement group* – leaves of which are a deterministic and consistent list of the storage nodes. If a storage policy, salt and a network

Placement rule	Bucket result	Placement group result
SELECT(2, Loc)	[Eu] ∪ [Ru]	[[Node 3, Node 4, Node 5, Node 6], [Node 7, Node 8, Node 9, Node 10]]
FILTER(Storage:SSD, equal)	([Eu] ∩ [Storage:SSD]) ∪ ([Ru] ∩ [Storage:SSD])	[[Node 4, Node 5, Node 6], [Node 7, Node 8, Node 9, Node 10]]
SELECT(3, Node)	($[N_1, N_2, N_3]$ ∈ ([EU] ∩ [Storage:SSD])) ∪ ($[N_4, N_5, N_6]$ ∈ ([Ru] ∩ [Storage:SSD]))	[[Node 4, Node 5, Node 6], [Node 8, Node 9, Node 10]]

Table 1: The placement rule application to the network map graph

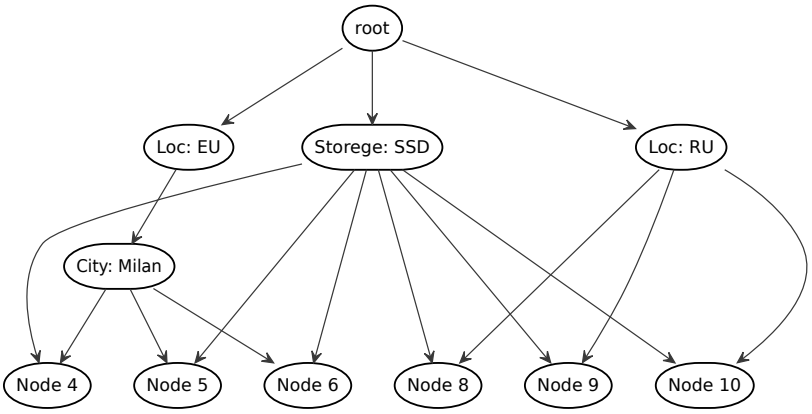

Figure 3: The obtained subgraph of the network

map are known *Placement group* can be retrieved without referring to a third party or storing "object and storage node" pairs.

4.1.1 Rendezvous Hashing

Buckets and placement nodes in a bucket are selected using the Rendezvous hashing algorithm. With it, each node or a bucket has an individual hash number for an individual item, and the bucket or node, having the largest hash number, is chosen. Data in this algorithm is distributed uniformly and a small number of data movement is required when nodes are added or removed. However, it can achieve minimum data movement when nodes are added or removed.

The basic idea is to give each node N_j a score (weight) for each object O_i, and assign the object to the highest scoring node. Firstly, all the clients agree on a hash function h(). For object O_i, the node N_j is defined to have weight $w_{i,j} = \mathrm{h}(O_i, N_j)$. HRW assigns O_i to the node N_m whose weight $w_{i,m}$ is the largest. Since h() is agreed upon, each client can independently compute the weights $w_{i,1}$, $w_{i,2}$, ..., $w_{i,n}$ and pick the largest. If the goal is a distributed k-agreement, the clients can independently pick the nodes with the k largest hash values.

If a node N is added or removed, only the objects mapping to N are remapped to different nodes, satisfying the minimal disruption constraint above. The highest random weight assignment can be computed independently by any client, since it depends only on the identifiers for the set of nodes N_1, N_2, ..., N_n and the object being assigned.

4.2 Container

In order to place objects in the system a user needs to define a container.

The container has at least the following fields:

- an item uuid (for object addressing) being also salt required for a placement function to select buckets deterministically and pseudorandomly;
- a container's owner;
- a storage policy:
  - a placement rule;
  - a redundancy factor;
- a maximum capacity.

The container defines the subgraph of a network map.

4.2.1 Select Limitation and Container Registration

The approach, where a complete network map, which includes all connected nodes and plays the role of an initial graph, can impose additional costs when a network expands. The allocation of limited space within the network map (subgraph), called a container, allows to do the following:

- to isolate a subset of nodes. Handling the subset is faster and more predictable. After the container has been provided with a graph, container's network map traversal operations are faster;
- to protect against overprovisioning – the overall container‘s capacity and its remaining capacity can be roughly estimated;
- to control access to the container;
- to simplify payment operations by linking them to the container;
- to facilitate the integration of s3 and swift API for *DDSP*.

To form a container‘s subgraph, a replication factor in each SELECT(r, type) call is increased by a redundancy factor K_r. A container's uuid is used as salt for selection in a placement function. An example, where $K_r = 2$, is considered:

As a result, a set of nodes that can be represented as a subgraph of the network map is obtained. When forming the container, it is possible to approximately estimate it's capacity.

To do so, the weights of nodes are assigned depending on nodes' capacity. A total weight of the selected nodes has to correspond to the container's declared capacity. The object is placed into the container by using the placement function, where the hash of the object being placed is used as salt.

Highlighted buckets are *Placement group* subgraph for the object.

Source	Placement rule	Placement function result	Salt
Network Map	SELECT($2 \cdot K_r$, Loc)	[US , EU, RU, Ja]	Container salt
	SELECT($2 \cdot K_r$, node)	[[N_1, N_2, N_3, N_4], [N_5, N_6, N_7, N_8], [N_9, N_{10}, N_{11}, N_{12}], [N_{13}, N_{14}, N_{15}, N_{16}]]	

Table 2: Container

Source	Placement Rule	Placement function Result	Salt
Container	SELECT(2, Loc)	[US, Ja]	Hash of the object being placed
	SELECT(2, node)	[[N_2, N_4], [N_{15}, N_{16}]]	

Table 3: Placement group

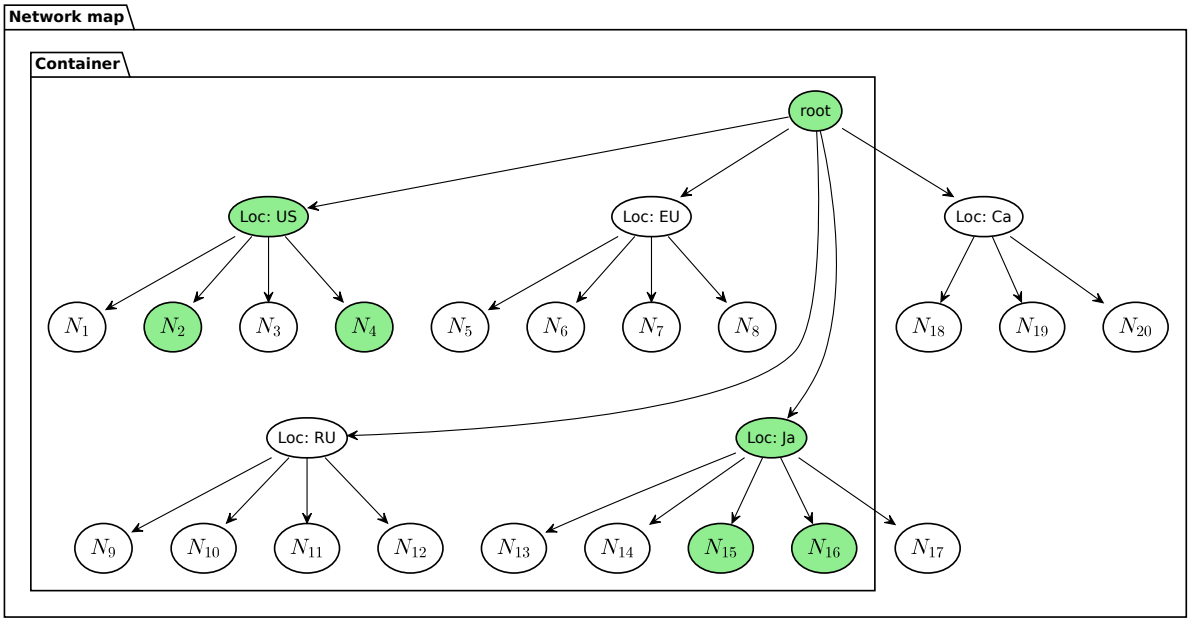

Figure 4: An example of the placement function's result

4.2.2 Matching Container Uuid and Storage Policy

To encode a placement rule in a container uuid seems to be impossible since the placement rule can be of an arbitrary complexity and length, with a set of SELECT() and FILTER() operations for random buckets. A pair container's uuid - placement rule needs to be stored and distributed in *DDSP*. To match the container uuid and the placement rule, DHT is used.

Once the first query has been completed, the result is cached.

5 Data Storage

5.1 Object Format

To implement data storage and processing, the system operates with objects. An object is a structure intended to be placed on a data storage device and transmitted over a network. This structure consists of a user's data block of a finite length and a set of headers containing information about the data and the object itself. The size of the data in the object can be zero.

Firstly, there is a fixed header containing the version of the object format, its total length, unique identifier, optional reference to a parent object, electronic signature of a data publisher, type identifier of the next extended header and identifier of the storage group.

The extended type header has a similar structure, and its last field indicates the type of the next extended header. In this way, backward compatibility is maintained at the data format level and the ability to extend supported functionality is provided. In fact, the version of the data format is determined by the list of known and supported formats of extended headers.

User data attached to the object is placed after headers.

The object in the system is immutable. It cannot be changed under any circumstances.

DDSP core works only with a fixed object header and treats data as an immutable sequence of bytes without interacting with the content.

Extended headers store information about user data properties, encryption algorithms, checksum values, cryptographic signatures, identifiers, links to other objects, etc. Extended headers are processed in the order of appearance by separate data processing modules.

Such an object format allows, by delegating the processing and conversion of user data to higher-level modules of the system, to organize complex schemes of working with information.

At initial stages, to simplify the development, an object can be placed in a file system of a storage node as a simple file. In the future, to improve the performance and efficiency of using a disk subsystem, it is possible to directly use raw block devices to store objects.

5.2 Object Meta-Database

Each node locally supports a meta-database of objects placed on it. The database supports indexes for quick access to an object on a disk, graphs of links between objects in a convenient representation, statistical information, and others necessary for the node to work.

The meta-database is an auxiliary database. It is neither replicated to other nodes, nor stored in a blockchain. It can be completely restored based on information from a network map, *Inner Ring* and objects on a storage node itself.

5.3 N-factor Replication

In the simplest storage policy case, a replication factor is set for an object. During a control period, the object needs to be available in at least a specified number of copies on different storage nodes.

5.4 Erasure Coding

When using erasure coding, data is divided into several parts so that the original data can be restored even having an incomplete set of source parts. In this case, a hierarchy of objects is created.

Information about the algorithm and parameters of erasure coding is placed in the extended header of the root object.

5.5 Availability and QoS/SLA

A required level of object availability and parameters of service delivery are a mater of finance and affect payments of rewards or assignment of penalties to nodes.

The system can take into account the statistics of nodes when selecting locations of a new object so that SLA requirements are highly likely to be met in the future. However, this is a topic of further research.

6 Data Audit

In the decentralized storage system, where users no longer physically own the storage of their data, traditional cryptographic primitives for the purpose of data validation cannot be directly adopted. A group of works have been done focusing on attempt to solve a remote data validation task in a cloud storage. These methods can be classified as Proof of Data Possession (PDP) and Proof of Retrievability (PoR).

The PDP scheme initially has been presented by Ateniese et al. ("Provable Data Possession at Unstructured Stores", 2007). Related protocols detect a large amount of corruption in out-sourced data. However, the schemes do not support the possibility to allow an external party to verify the correctness of remotely stored data without knowledge of data content, what is needed for decentralized storage systems.

An efficient PDP method with privacy protection of users data from external auditors on a client and server sides is needed for a decentralized storage. From the perspective of data privacy protection, this drawback greatly affects the security of protocols.

The design of *DDSP* requires to develop an efficient data validation method taking into account a network scalability issue, possibility to check data without knowledge of content and privacy protection of users data. It is proposed a zero-knowledge data validation method for a *DDSP* for minimizing data transferring to maintain a network scalability and minimizing a computational cost on the side of a storage node to maintain a large number of parallel interactions, and on the side of a validating node. The integrity guarantee is achieved due to the proposed data validation method based on the homomorphic hash function which allows to verify the integrity of data on a storage node without transferring real data to a validating party over the network.

6.1 Data Validation Method

Requirements for the data validation method for *DDSP*:

- the ability to publicly verify stored data without an actual object ownership and the knowledge of an object content (zero-knowledge proof);
- validation needs to ensure that a storage node's response cannot be saved and repeated to counteract nodes-malefactors (method has to be developed in accordance with an arbitrary task of validation);
- minimization of a computational cost on the side of a storage node to ensure a large number of parallel interactions and on the side of a verifying node;
- minimization of a network load to maintain a network scalability.

6.1.1 Homomorphic Hashing

In order to fulfill the requirements, the data validation method based on homomorphic hashing is proposed.

A hash function has to have the following properties:

- it should be easily computable;
- it should be computationally difficult to find collisions.

Homomorphic hash is a hash function that can compute a hash of a composite block from hashes of individual blocks.

A computational complexity of validations depends linearly on the size of validated data and is well suited for parallelization.

In the proposed method, a challenge-response model is applied. In the simplest case, a request for an arbitrary set of hashes from the data being verified is considered. The obtained hashes have to produce an expected hash according to the rule of homomorphism.

Validation of each individual object is a computationally expensive task for a decentralized system with an unlimited number of users and the amount of stored data. A storage group is introduced to reduce a computational cost and the amount of stored meta-information for validation.

6.1.2 Data Validation Complexity: Storage Group

The concept of a storage group is introduced to reduce a validation complexity dependence on the number of stored objects in the system. The storage group encapsulates a group of objects' identifiers and a set of homomorphic hashes required for data validation. The safety and accessibility of multiple objects in the network are achieved by the storage group validation without storing meta-information and conducting validation of each object. One container can have any number of storage groups. The storage group is located on the *Inner Ring* nodes. The

storage group is an immutable structure, however, new storage groups can still be formed based on the existing ones (the merge operation) or the storage group can be removed (the objects will be deleted). A group of homomorphic hashes is generated for data validation. A fixed number of hashes is created from zones (first-level hashes). A fixed and equal number of homomorphic hashes is stored for all the groups in *DDSP*.

In the simplest case, a validating node selects a random first-level homomorphic hash during data validation and requests some number of second-level homomorphic hashes from *Placement group* nodes. Merging the second-level homomorphic hashes, a validating node confirms the validation if the obtained hash corresponds to the first-level hash being stored.

6.1.3 Challenge and Response

Challenge is formed to confirm objects storage for the selected storage group. The division of nodes into pairs, storing copies of one object, is formed to cover with a minimum combination of pairs of nodes all nodes that store data from the selected verification area.

For each pair, a task is formed so that responses from one node can be checked by responses from another one. As a challenge, the task is given to provide a certain number of homomorphic hashes from a specific data range of the scanned object. The ranges in each case are determined pseudo-randomly so that a set of hashes of one node from the selected area of the object can be additionally checked by a set of hashes from another node. At the same time, the object itself can be independently verified for each node by hashing the responses received from it.

6.1.4 Advantages

The proposed method allows to minimize load on the network and validating nodes without transferring a real data over the network. The storage group allows to keep a fixed amount of meta-information needed for validation process regardless of the size and the number of objects from the storage group.

7 Data Replication

Storage nodes have to control data replication since they are motivated by incentive model to maintain an object storage policy in order to get paid.

7.1 Maintaining Storage Policy

Storage nodes have to check the availability of a required number of an object‘s replicas that they store.

If the number of the object‘s replicas is insufficient data is replicated to maintain a storage policy. Data replication uses a placement function and ignores failed nodes. This enables to deterministically define the nodes where all network participants have replicated data. A container remains consistent within one epoch (a single network map). An example for the policy:

SELECT(2, Loc)

SELECT(2, node)

$K_r = 2$

Placement rule	Placement function result
SELECT($2 \cdot K_r$, Loc)	[US , EU, RU, Ja]
SELECT($2 \cdot K_r$, node)	[[N_1, N_2, N_3, N_4], [N_5, N_6, N_7, N_8], [N_9, N_{10}, N_{11}, N_{12}], [N_{13}, N_{14}, N_{15}, N_{16}]]

Table 4: Container

The object placement function have the following weight ratio:

$w_{Loc:US} > w_{Loc:Ja} > w_{Loc:EU} > w_{Loc:RU}$

Loc:US: $w_{N_2} > w_{N_4} > w_{N_1} > w_{N_2}$

Loc:Ja: $w_{N_{16}} > w_{N_{15}} > w_{N_{14}} > w_{N_{13}}$

Loc:EU: $w_{N_5} > w_{N_6} > w_{N_8} > w_{N_7}$

Placement rule	Placement function result
SELECT(2, Loc)	[US, Ja]
SELECT(2, node)	[[N_2, N_4], [N_{15}, N_{16}]]

Table 5: Placement group

In the first case, let's consider the failure of the node N_{16} in the subgraph "Loc: Ja" where the subgraph can still follow the storage policy. In this case, the verification node N_{15} may mark the node N_{16} as the failed one and apply the placement function for the stored object, not allowing for the node N_{16} (Fig.5).

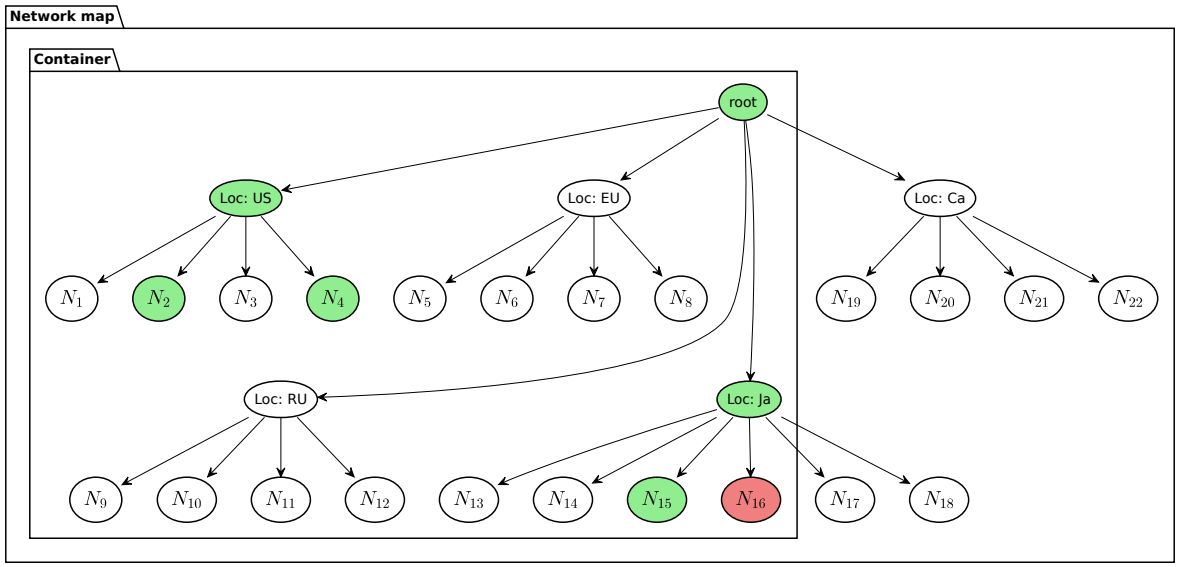

Figure 5: Example of the failed node

Placement rule	Placement function result
SELECT(2, Loc)	[US, Ja]
SELECT(2, node)	[[N_2, N_4], [N_{14}, N_{15}]]

Table 6: New Placement group

The node N_{14}, having checked the node N_{16} for unavailability, is ready to receive a copy of the object from N_{15} (Fig. 6).

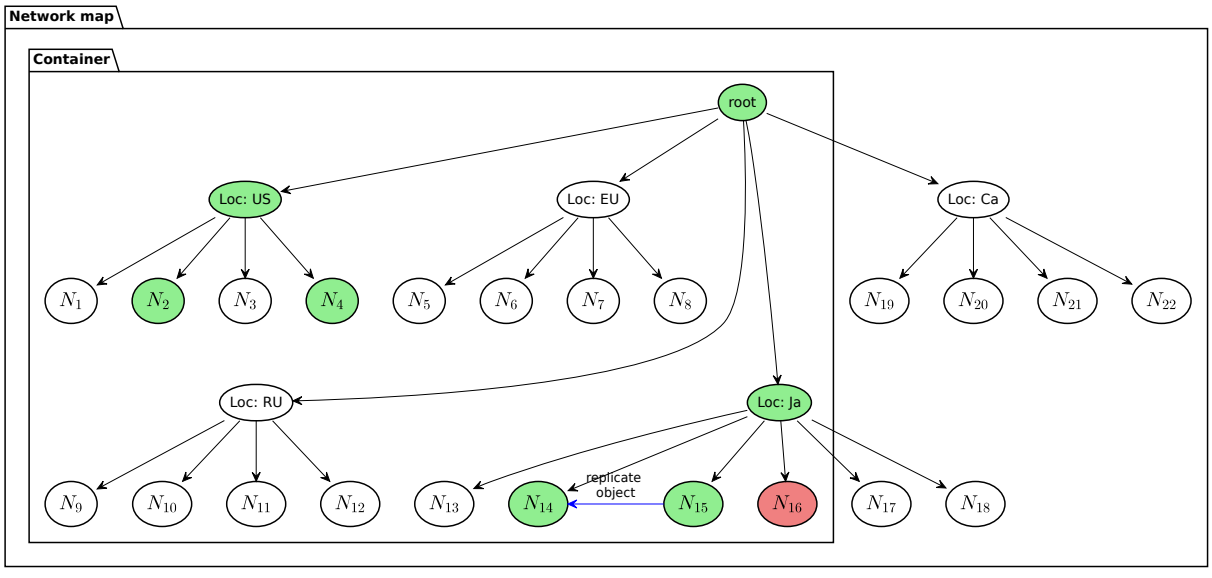

Figure 6: Object replication

The case, where the subgraph can not follow the storage policy due to the failure of several nodes (N_{13}, N_{15}, N_{16}), is considered (Fig. 7).

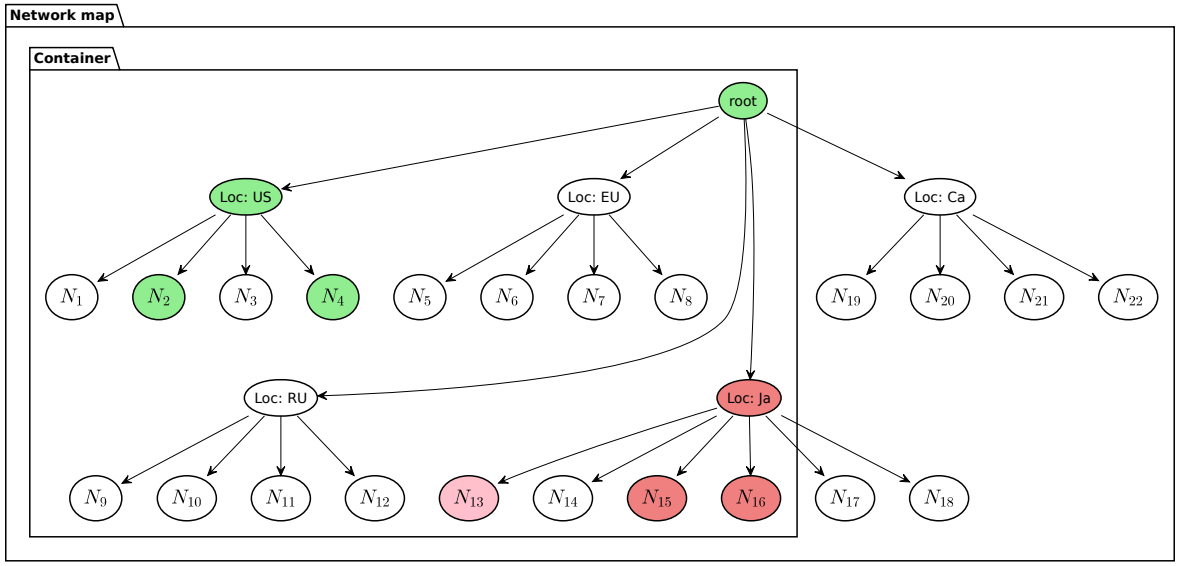

Figure 7: Example of the failed nodes

In this case, the nodes are marked as failed and the object placement function is applied. Instead of the subgraph "Loc: Ja" that does not meet the storage policy now, the object placement function chooses the next by weight subgraph that does meet the requirements of the storage policy – "Loc: EU". According to Rendezvous hashing by maximum weights, the nodes N_5 and N_6 are selected for placement (Fig. 8).

Placement rule	Placement function result
SELECT(2, Loc)	[US, EU]
SELECT(2, node)	[[N_2, N_4], [N_5, N_6]]

Table 7: Placement group

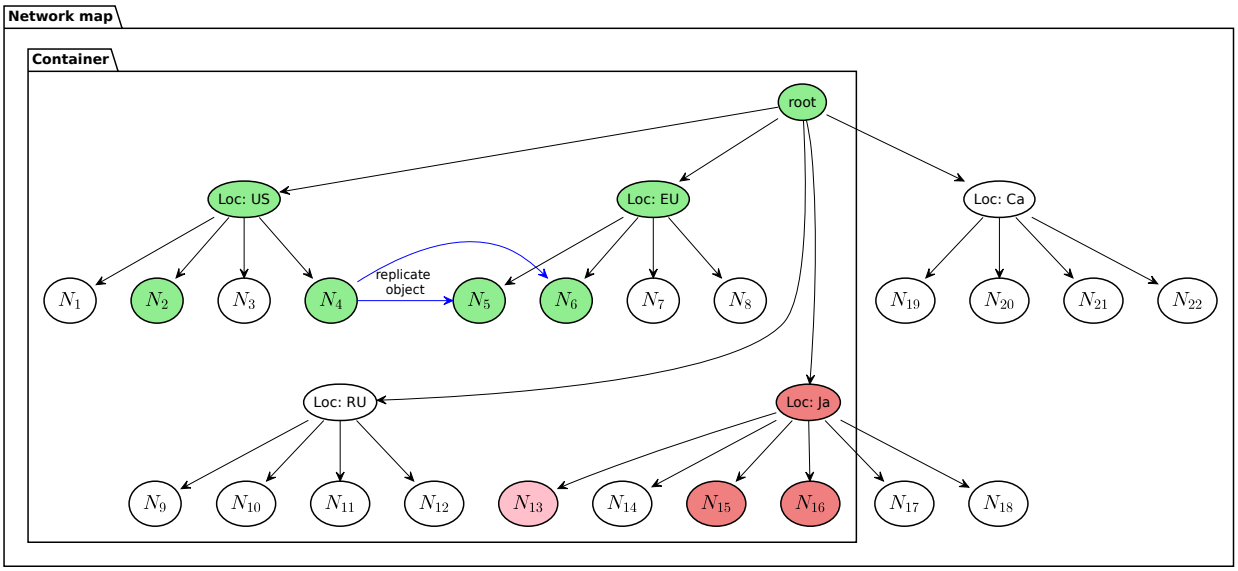

Figure 8: Object replication

7.2 Reorganization and Data Movement

When changing an epoch, a situation may arise where data needs to be rebalanced, if a node storing an object no longer falls into a placement group. In this case, it checks that the object is available in a new storage group; afterwards the object can be deleted.

8 Data Services

8.1 End-to-end Encryption

The core of the system does not deal directly with user data, so a data publisher is responsible for data encryption. To save information about encryption keys, algorithms and signatures, the corresponding extended object headers are used. Control of encryption keys is assigned to the data publisher.

8.2 Data Deduplication

Through the mechanism of representing an object, which consists of references to data from other objects, data deduplication can be implemented at a block level using the algorithm for reconciling the hashes of data with a sliding window when uploading data into the system.

8.3 POSIX-like FS

Representing *DDSP* as a fully POSIX-compatible file system is a challenge because of differences in data models. At the first stage, it is suggested implementing access in the “read-only mode”, with only regular files and directories supported.

From the previous experience, this should already be enough for integration with most legacy applications.

The implementation of access support with the possibility of writing requires further research.

8.4 HTTP Gateway

It seems favourable to develop a module for Nginx that implements an internal data access protocol, for support in HTTP. At the first stage, it is suggested implementing access in the "read-only mode" only.

8.5 External Storage Integration

To efficiently adopt *DDSP* in existing projects, it is necessary to have a possibility to use existing data storage systems. The most popular protocols that need to be supported are AWS S3 and OpenStack Swift.

For data in an external system, it is suggested creating objects with a null data block and an extended header that contains the address of an object in the external system.

9 Neo Blockchain

9.1 Distributed Decentralized Storage Platform PKI

If X509 PKI is used, a node authentication for connecting to a network has to be done together with verifying the node certificate signature. In the absence of an external Certification Authority (CA) and the mechanism for issuing and distributing certificates, it makes sense to develop CA integrated with NeoID and used Automated Certificate Management Environment (ACME) protocol for issuing certificates. As a *Client* Challenge, the proof of ownership of NeoID can be used.

9.2 Neo Ecosystem

As a common solution, *DDSP* is integrated as an element of Neo Ecosystem with a deposit-based payment by the GAS tokens and is based on the Neo smart contract.

We are considering different options for using *DDSP*: as an ecosystem element for internal and external use in the NEO blockchain, and integration with other projects.

External use:

- user data storage;
- neo blockchain-based DApps can use the storage for keeping data;
- storing data and files for smart contracts.

Possible Neo blockchain internal use:

- Storing smart contract code with keeping only the hash of the smart contract script in the blockchain to reduce a deployment cost;
- Old Block Data can be stored by the storage, not by Full Nodes as it now, to increase the Neo blockchain scalability.

Neo smart contract is used as a deposit storage of the *Publisher*'s payment for subsequent phased distribution between storage nodes after verification of initiated data availability (data audit).

In addition, smart contract saves a list of nodes to enter *Inner Ring* and an updated list of *Inner Ring* nodes as well as information for their identification.

9.3 Neo Smart Contract: Payments and Accounting

Accounting is based on the Neo Blockchain. Primary asset storage in *DDSP* is a smart contract. We deploy the smart contract into the NEO Blockchain and expose it as a wallet address. *DDSP* users might transfer assets to this wallet, what is equal to deposit placement into the system.

Inner Ring nodes run a blockchain explorer in the background and monitor assets movement to the smart contract wallet. Through transaction details analysis, we can define the sender's wallet as a ScriptHash and use it as an identity in the system. Users' deposits are stored in local databases and are synchronized among nodes via consensus.

The smart contract provides a call to the user to withdraw assets from the deposit back to his wallet. This call costs a small amount of GAS and is stored in the blockchain as a transaction. A once written transaction is noted by blockchain monitors, and a selected *Inner Ring* node by the consensus sends the transaction for assets transfer from the smart contract wallet to the user's one.

Besides payments routine, the smart contract is used as a key storage. We store two things: *Inner Ring* nodes public keys and *Inner Ring* entrants queue.

Inner Ring keys storage lets verify the initiator of the smart contract calls related to transfer of rewards through the 'CheckWitness' syscall. Thus, an *Inner Ring* node can only initiate payouts to *Hosts*.

When *Host* is going to become an *Inner Ring* node, it should place enough deposit and put itself into the entrants queue. This requires a certain amount of GAS to call the smart contract. The payment confirms the intention of *Host* to serve administrative tasks of *DDSP*.

The placed deposit is considered as an advance payment for(1) the smart contact call to remove *Host* from the queue, (2) the smart contact call to add a *Host*'s key to *Inner Ring* keys list and (3) the smart contract call to remove it from the list, upon *Host* requests to leave *Inner Ring*.

10 A View to the Future

- The possibility of paying in fiat- and crypto-currency (*nash.io* can be used to convert input payments into GAS) to be considered

- The HTTP access to stored objects (gates) for public data to be implemented
- Integration with other Neo Ecosystem projects to be provided
- The possibility of storing Old Block Data instead of Full Nodes by *DDSP*, to increase the Neo blockchain scalability, to be studied
- The possibility of creating private payment channels

11 Incentive Model

- It is necessary to solve the problem with latency and bandwidth overhead associated with the use of the platform
- It is necessary to ensure the storage cost is lower than existing solutions, such as Amazon S3, Microsoft Azure and others. At the same time, it is necessary to set a sufficient storage cost to enable storage nodes to make profit

Possible ways of commercializing the project:

- get royalty from successful smart contract operations;
- build a group of own storages to share storage capacity;
- adapt the project to private storage solutions, for different business customers.

12 Distributed Decentralized Storage Platform Advantages

- A unique data audit method that has a minimum load on the network and a computing power of verifying- and verified parties
- Building a p2p network by using a unique data placement method reduces the network load
- Truly decentralized without a single point of failure
- *DDSP* versioning support
- Support for private- (encrypted) and public data
- Optional support for erasure codes
- Platform that can be used in various Dapps for the Neo Blockchain
- API S3 and Swift compatibility

- The system works with both a normal disk space and private data storage space that works with various protocols (Swift / s3 and etc)
- Private storage can be implemented on *DDSP* for specialized business tasks

13 Points to Research

- Options for the execution of data audit procedures using homomorphic hashes
- Using of node statistics when selecting a node to store an object to ensure the required probability of object availability
- Algorithm to calculate an optimal number of copies of each object based on a minimum cost of data storage with requested storage policy
- It is necessary to consider the possibility of supporting popular protocols AWS S3 and OpenStack Swift for storing objects in the existing storage node of traditional systems, the space of which can be used for *DDSP*
- Methods to increase data availability (erasure coding optimized for *DDSP* or other algorithms)
- Metadata format and object header fields
- Search for the optimum implementation of meta-databases
- Possibility of using private channels to pay for object storage
- Variants of a smart contract work logic and payment model for object storage
- Incentive model and Economic Research