NeoFS Technical Specification

Architecture and Implementation details

Neo Saint Petersburg Competence Center

Date: November 14, 2023
Revision: c57acefc

Contents

Introduction

Overview

NeoFS is a decentralized distributed object storage system integrated with the Neo Blockchain[1].

We store and distribute users' data across a peer-to-peer network of NeoFS Nodes. Whether a business or an individual, any Neo user may join the network and get paid for providing storage resources to others, or pay a competitive price to employ NeoFS as a storage solution.

The decentralized architecture and flexible storage policies allow users to reliably store object data in the NeoFS network. Each NeoFS Node is responsible for executing the specific storage policies selected by the user, including the geographical location, redundancy level, number of nodes, type of disk, capacity, etc. Thus, NeoFS enables a transparent data placement process which gives full control over data to the users.

Deep Neo Blockchain[2] integration allows NeoFS to be used by Decentralized Applications (dApps) directly from NeoVM[3] on the Smart Contract[4] code level. As a result, dApps are not limited to on-chain storage and one can manipulate large amounts of data without paying a prohibitive price.

NeoFS provides native gRPC[5] Application Programming Interface (API) and supports popular protocol gateways such as AWS S3[6], HTTP[7], FUSE[8], and sFTP[9], which allows developers to easily integrate their existing applications without rewriting code.

Together, this set of features makes it possible to utilize a dApp's Smart Contract to manage monetary assets and obtain data access permissions on NeoFS through a regular Web Browser or a mobile application.

[1] https://neo.org
[2] https://neo.org
[3] https://docs.neo.org/docs/en-us/basic/technology/neovm.html
[4] https://docs.neo.org/docs/en-us/basic/technology/neocontract.html
[5] https://grpc.io
[6] https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
[7] https://wikipedia.org/wiki/Hypertext_Transfer_Protocol
[8] https://wikipedia.org/wiki/Filesystem_in_Userspace
[9] https://en.wikipedia.org/wiki/SSH_File_Transfer_Protocol

Background

Technical Requirements

Out of Scope

Future Goals

Architecture overview

Design and components

NeoFS heavily relies on the Neo Blockchain and its features. This allows NeoFS nodes to focus on their primary tasks — data storage and processing, while asset management and distributed system coordination are left to Neo and a set of Smart Contracts. Under this approach, the Blockchain is mainly used as a trusted source of truth and coordination data.

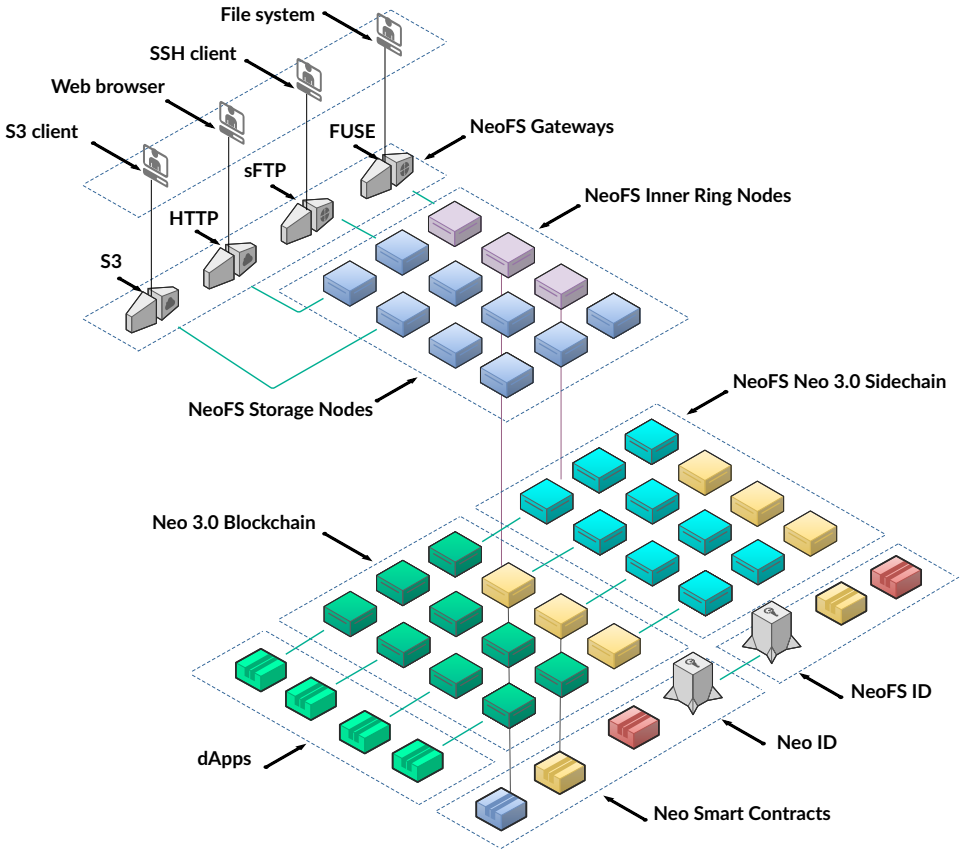

Figure 1: Architecture overview

The N3 Main Net hosts a NeoFS Native Contract[10] concerned with user deposits and withdrawals, network settings, and other maintenance operations such as listing the keys of trusted nodes.

[10] https://medium.com/neo-smart-economy/native-contracts-in-neo-3-0-e786100abf6e

To simplify accounting operations, lessen Main Net burden, and reduce the overall network maintenance costs, NeoFS utilizes an N3-based sidechain[11]. The NeoFS Sidechain runs Smart Contracts which control the NeoFS network structure, user settlements, balances, and other frequently changing data.

There are two types of NeoFS nodes. They are Storage nodes and Inner Ring nodes.

The first type is responsible for receiving data from a user, reliably storing it as required by the storage policy, and providing access to the data according to the applicable Access Control Lists (ACLs). Such storage nodes are coordinated with Smart Contracts from the Sidechain.

The second type does not store user data. Inner Ring nodes monitor the NeoFS network health, aggregate Storage Nodes reputation ratings, and perform data auditing, issuing penalties and bounties depending on the audit results. Inner Ring nodes listen for both Main Net and Sidechain, providing a trusted and reliable way of data synchronization between the two Blockchains.

Each Storage node in the system has a set of key-value attributes describing node properties such as it's geographical location, reputation rating, number of replicas, number of nodes, presence of SSD drives, etc. Inner Ring nodes generate a Network Map — a multi-graph structure which enables Storage nodes to be selected and grouped based on those attributes.

In NeoFS, a user puts files in a container. This container is similar to a folder in a file system or a bucket in AWS S3, but with a storage policy attached. The Storage Policy is defined by the user in an SQL-like language (NetmapQL), specifying how and where objects in the container have to be stored by selecting nodes based on their attributes. Storage nodes keep data in accordance with the policy, otherwise they do not get paid for their service.

All storage nodes service fees are paid in GAS Utility Token. After receiving GAS, a node operator may spend them to pay for their own data backups on other NeoFS nodes, or simply withdraw it for use with other services provided by the Neo Blockchain ecosystem.

An innovative feature of NeoFS is that it can be accessed directly from NeoVM on the smart contract code level. Thanks to the N3 Oracle protocol and the integration between NeoFS and Neo Blockchain, dApps are not limited to on-chain storage and can manipulate large amounts of data without paying a prohibitive price for it.

NeoFS provides native gRPC API and supports the most popular protocol gateways, allowing easy integration with other systems and applications without requiring code rewrites.

Such an architecture makes it possible to implement a dApp's smart contract to manage digital assets and data access permissions on NeoFS and lets users access that data via regular Web Browsers or mobile applications. In the long term, we plan to add more ecosystem components that will facilitate

[11] https://en.wikipedia.org/wiki/Blockchain#Types

the development of truly decentralized applications, solving almost any problems that are nowadays only possible in a centralized manner.

Epoch

For the NeoFS network to work properly, all nodes should have the same view of the network. This snapshot view must be tied to some timestamp, but in the distributed environment for NeoFS there is no reliable common source of time other then a monotonically increasing number of blocks in the Blockchain. It means we can only use discrete time model[12] and define some time period in blocks to snapshot the common view of the network. This period has to be small enough to keep the snapshot information fresh and big enough to let the information be distributed fast enough between network nodes. This regular time period is called Epoch.

During an Epoch all common information snapshots are immutable. New nodes can be registered, misbehaving nodes can be removed, some nodes can go offline, but those changes will be reflected only in the next version of the netmap issued for the next Epoch. All these changes signed by Inner Ring are propagated to the network only when the new Epoch starts.

Network Map

NeoFS Network Map, or just “netmap”, is a structured representation of all active storage nodes available in NeoFS network for the current Epoch.

Storage nodes in the netmap are identified by public key. Netmap also has additional information about each node, like network addresses and a list of attributes.

Attributes are key-value pairs with string values. Depending on the Attribute, the string value can be interpreted as a number or Hex or something else. For detailed information please see the API reference.

Here is how node information in netmap may look like:

```
key: 03e9c4847fb2f4d58161a808ff74363139c4e617cb233a2e96cc6c4c7f219dd9bf
address: /dns4/st1.storage.fs.neo.org/tcp/8080
state: ONLINE
attribute: Capacity=10000
attribute: Continent=Europe
attribute: Country=Germany
attribute: CountryCode=DE
```

[12] https://en.wikipedia.org/wiki/Discrete_time_and_continuous_time

```
attribute: Deployed=NSPCC
attribute: Location=Falkenstein
attribute: Price=0.00000042
attribute: SubDiv=Sachsen
attribute: SubDivCode=SN
attribute: UN-LOCODE=DE FKS
```

Some node attributes can be grouped (e.g. geographical ones), creating a graph representation for the network map. Strictly speaking it is a forest of rooted trees where leaves (single nodes) are shared. Every tree represents a single attribute group. For example, geographical attributes can be naturally ordered: `Continent`, `Country`, `SubDiv`, `Location`, DC (data center) or any other location identifier.

User attributes can have any name and value. They are used primarily for storage policy rules. For example, one may want to designate their own nodes with some specific attribute to be sure (with the help of storage policy) that at least one copy is always stored locally on the nodes one controls.

Storage Policy

In NeoFS, storage policy is a flexible way to specify rules for storing objects. The result of storage policy rules application to the network maps is a set of storage nodes, able to store data according to the requested policy. This result maybe called “placement”, hence sometimes you may find the term “placement policy” used to denote the same thing as the “storage policy”.

Because Storage Policy is attached to the container structure there is a compact definition for system’s internal use and some higher level language definitions for humans to use, that are translated to internal representation. For example, there is an SQL-like language to be used by humans, JSON notation to be used in software and there may be many others, like a graphical language using Blockly[13]. In our examples we will use SQL-like notation.

Storage Policy internal definition consists of four parts:

1. Filters
2. Selectors
3. Replicas
4. Container Backup Factor

The result of applying a storage policy to the netmap is a set of nodes structured by `Replicas` and used to select candidates to put objects on. The selection algorithm is deterministic, hence on dif-

[13] https://developers.google.com/blockly

ferent nodes or clients the same Storage Policy applied to the same version of netmap will give the identical result.

Filters

Filter is a mechanism to specify which nodes are allowed to store an object. This is done by querying node's attributes and checking if they satisfy certain condition. For example, it allows to precisely specify "Store objects in nodes from Europe, but not from Italy, which have SSD and have a good reputation".

Simple filter can compare a single node attribute with some value.

It has 3 fields:

1. `Key` — name of node attribute
2. `Value` — value to compare attribute with
3. `Op` — operation to be used for comparison

For better understanding, simple filters will be specified as `Key Op Value`.

For example:

1. `Country = Argentina` means use nodes for which `Country` attribute is equal to `Argentina`, i.e. nodes located in USA.
2. `Rating > 4.5` means use nodes with rating better then 4.5

Currently only eight operations are supported, two of which are used for creating compound filters from the other ones.

1. `EQ`/`NE` check if attribute is equal/not equal to the filter's value.
2. `GT`/`GE`/`LT`/`LE` check if numerical attribute is greater-than/greater-or-equal/less-than/less-or-equal than the filter's value.
3. `OR` checks if node satisfies at least one of the filters provided as arguments.
4. `AND` checks if node satisfies all filters provided as arguments.

Compound filter can combine simple filters to specify arbitrarily complex conditions. Consider example from the previous section: we may write `Country = Argentina AND Rating > 4.5` to filter nodes which are located in Argentina and have a good rating at the same time.

If filters are used in selectors or other filters, they should have name. Consider filter `Country = Finland OR Country = Iceland AS ColdCountry`. If we need nodes from these countries but want to vary maximum price depending on the storage type they have, we may write this filter:

```
ColdCountry AND StorageType = SSD AND Price < 100
OR
ColdCountry AND StorageType = HDD AND Price < 10
```

Selectors

Selector is a mechanism to specify which of the previously filtered nodes will be included in the container. It has 5 fields:

1. `Name` — name that can be referred to
2. `Attribute` — name of the attribute for grouping nodes. When it is set, nodes are grouped in buckets based on `Attribute` value. It can be omitted to create buckets "randomly".
3. `Count` — number of nodes to be included in a bucket or number of buckets, depending on `Clause`.
4. `Clause` specifies how `Count` is interpreted:
    - `SAME` — choose nodes from the same bucket
    - `DISTINCT` — choose nodes from distinct buckets
5. `Filter` — name of the filter to choose nodes from. If it is omitted or is `*`, all nodes from the netmap are used.

Selector can return different set of nodes for every epoch; however, they are always the same on each storage node having the same netmap. The degree to which this set of nodes is changed also depends on how strict the filter is. For example, if we select a few nodes based on a very specific attribute, this set will always be the same. However, if all these nodes go down, data can be lost.

Replicas

Replica is an independent set of nodes where single object copy is stored. It can refer to selector (by default all nodes are considered) and can specify a number of copies to store.

Container Backup Factor

Container backup factor (CBF) controls maximum number of nodes to be included in a container's node set. It doesn't set strict boundaries, though. Consider placement policy which selects X nodes in 2 different countries with CBF 2. In this case, we can expect container's node set to have from X to X * 2 nodes in every selected country. Having less than X * 2 nodes is not considered as fail.

Objects

NeoFS stores all data in the form of objects, thus providing an object-based storage to the clients. These objects are placed in a flat environment (no hierarchy or directories). To access the required data, the identifying details (ID and metadata) are needed.

ObjectID is a hash that equals Headers hashes plus Payload hashes. Any object includes a system header, extended headers, and a payload. A system header is an obligatory field, while extended headers may be omitted. However, any extended header should follow a particular structure (e.g. IntegrityHeader is a must). A user can add any extended header in the form of a key-value pair, though keeping in mind that it cannot be duplicated with several values. One attribute – one value. Please note that any object initially has FileName, so that you cannot create an extended header with it as a key.

The maximum size for an object is fixed and can be changed only for the whole network in the main contract. It means that if a file is too heavy, it will be automatically divided into smaller objects. This smaller parts are put in a container and placed to a Storage Node. Later, they can be assembled to the initial object. Such assembling is performed in the storage nodes upon a corresponding request for a linking object. Once your file is converted into an object (or several objects), this object cannot be changed.

One can define the format of the object in an API Specification. For more information, see API Specification[14].

Large objects split

NeoFS has a limit on the maximal physically stored single object size. If there is a large object exceeding that `MaxObjectSize`, it will be split into a series of smaller objects that are logically linked together.

For each part of the original object's payload, a separate object with own `ObjectID` will be created. The large object will not be physically present in the system, but it will be reconstructed from the object parts when requested.

[14] https://github.com/nspcc-dev/neofs-api/tree/master/proto-docs

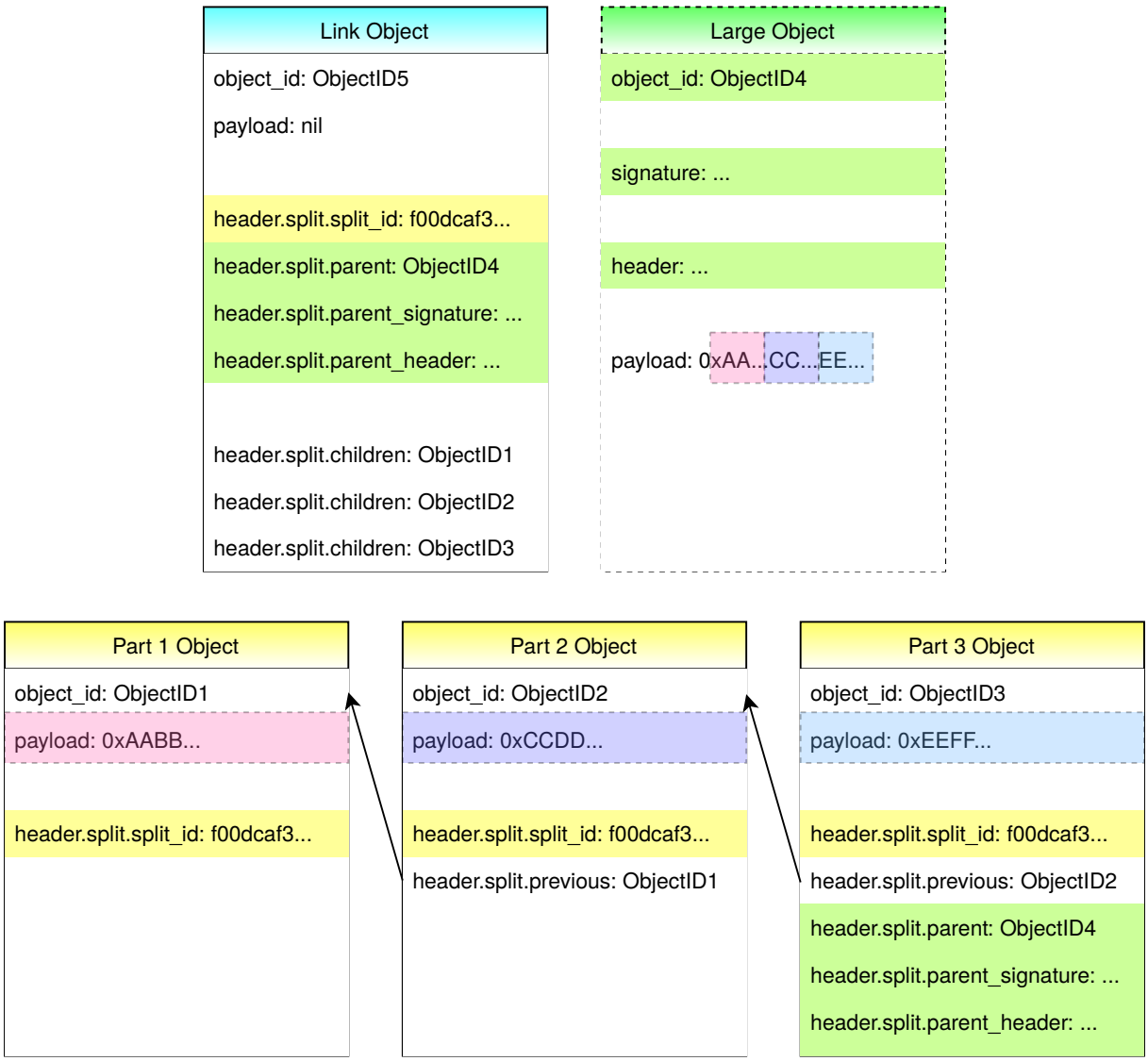

Figure 2: Large object split

All objects participating in the split have the `Split` headers set. Depending on the place in the split hierarchy it has different field combinations. There are four possible cases:

- First part
  First part object only has the `split_id` field set, as there is no more information known at this point
- Middle parts
  Middle parts have information about the previous part in `previous` field in addition to the `split_id`

- Last part
  At this point all the information about the object under split is known. Hence the last part contains not only the `split_id` and `previous` fields, but also the `ObjectID` of the original large object in its `parent` field, signed `ObjectID` in `parent_signature` and original object's `Header` in `parent_header`.
- Link object
  There are special "Link objects" that have the same common `split_id`, do not have any payload, but contain original object's `ObjectID` in `parent` field, it's signature in `parent_signature`, original object's `Header` in `parent_header` and the list of all object parts with payload in repeated `children` field. Link objects help to speed up the large object reconstruction and HEAD requests processing. If Link object is lost, the original large object still will be reconstructed from its parts, but it will require more actions from NeoFS nodes.

All of the split hierarchy objects may be physically stored on different nodes. During reconstruction, at first the link object or the last part object will be found. If it's a HEAD request, the link object or the last part object will have all the information required to return the original large object's HEAD response. For a GET request, the payload will be taken from part objects listed in the `split.children` header. As they are ordered, it will be possible to begin streaming the payload as soon as the first part object becomes available. If Link object is lost, some additional time will be spent on reconstructing the list from `split.previous` header fields.

If the whole payload is available, a large object may be split on the client side using local tools like `neofs-cli`. In this case the resulting object set will be signed with user's key. Such a split type can be called a "Static split".

When the large object's payload is not fully available right away, or it is too big to be split locally, the object upload can be started in a Session with another NeoFS node and be streamed in a PUT operation, part by part. Object parts will be automatically created as soon as the payload hits the MaxObjectSize limit. In this case, the resulting object set will be signed with a session key signed by user's key. This split type can be called a "Dynamic split".

Object Deletion

It's hard to guarantee complete and immediate object removal in a distributed system with eventual consistency. If some nodes are offline at the time of `DELETE` request processing, the object may still be available there and would be replicated to other nodes.

To address this issue, NeoFS Storage nodes don't remove objects immediately, but place a removal mark in a form of a Tombstone object. In order to avoid wasting storage space on the information

about data that has been already put into trash, tombstones are to be removed later by Garbage Collector.

Tombstone Object

Along with a Regular Object and a Storage Group, there is a Tombstone object type. It is like a regular object, but the payload contains a tombstone data structure. This entity is intended to synchronize object removal in a distributed system working in an unreliable environment. When one removes an Object, NeoFS Storage node actually creates a Tombstone alongside it and replicates the Tombstone among the Container nodes.

A Tombstone indicates that the given `ObjectID` does not exist any more and all requests to it must fail. Storage nodes may keep the data for a while until the Garbage Collector reaps it; though, it's up to the node's settings and implementation.

After several Epochs, we assume that deletion event has been properly spread among all Storage Nodes serving the Container, including those which haven't received it "in time" because of outage or lack of network connectivity.

The time to keep the tombstone may be varying. It is set in `__NEOFS__EXPIRATION_EPOCH` object attribute and `expiration_epoch` field in Tombstone structure. It may be set by a user directly or by an intermediate NeoFS Storage node using the default value. While the `__NEOFS__EXPIRATION_EPOCH` attribute is optional for a Regular Objects, it is obligatory for a Tombstone Object type.

Containers

In NeoFS, objects are put into containers and stored therein.

From the user's point of view, there are six verbs applicable for a Container: `PUT`, `GET`, `DELETE`, `LIST`, `SetEACL` and `GetEACL`. Also, there is an `AnnounceUsedSpace` operation which is intended for internal NeoFS synchronization. On an existing Container, any user is allowed to make `GET`, `LIST` and `GetEACL`. `DELETE` and `SetEACL` are allowed only for a Container owner, disregarding of a Container basic or extended ACL.

Any container has attributes, which are actually Key-Value pairs containing metadata. There is a certain number of attributes set authomatically, but users can add attributes themselves. Note that attributes must be unique and have non-empty value. It means that it's not allowed to set - two or more attributes with the same key name (eg. `Size=small`, `Size=big`); - empty-value attributes (eg. `Size=''`). Containers with duplicated attribute names or empty values will be considered invalid.

Access Control Lists

Access control in a decentralized untrusted environment is a complicated problem. It must be verifiable by every network participant and still be open for changes to revoke unwanted access permissions or adjust to infrastructure changes.

NeoFS solves this by using ACL rules from the combination of sources:

- Basic ACL in the container structure,
- BearerToken ACL rules in the request,
- Extended ACL rules in the SideChain smart contract.

ACLs specifies a set of actions that a particular user or a group of users can do with objects in the container. Each request coming through a storage node gets verified against those rules and rejected if the requests's action is not allowed.

Basic ACL

Basic ACL is a part of the container structure, and it is always created simultaneously with the container. Therefore, it is never subject to any changes. It is a 32-bit integer with a bit field in the following format:

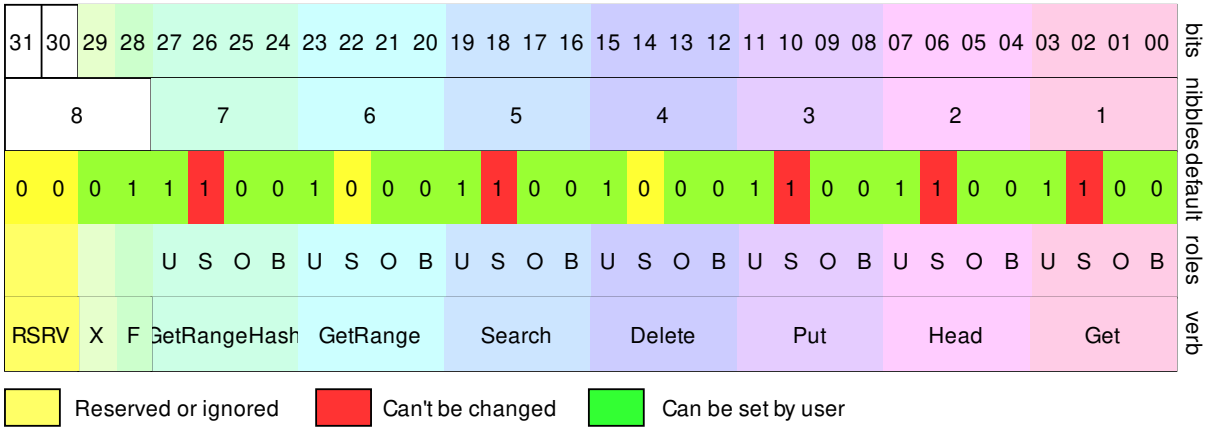

Figure 3: BasicACL bit field

Symbol	Meaning	Description
B	Bearer	Allows using Bear Token ACL rules to replace eACL rules
U	User	The owner of the container identified by the public key linked to the container

Symbol	Meaning	Description
S	System	Inner Ring and/or container nodes in the current version of network map
		IR nodes can only perform `Get`, `GetRangeHash`, `Head`, and `Search` necessary for data audit.
		Container nodes can only do verbs required for the replication. i.e., `Get`, `Put`, `Head`, `Search` and `GetRangeHash`.
O	Others	Clients that do not match any of the categories above
F	Final	Flag denying Extended ACL. If set, Basic ACL check is final, Extended ACL is ignored
X	Sticky	Flag denying different owners of the request and the object
		If set, object in `Put` request must have one `Owner` and be signed with the same signature
		If not set, the object must be correct but can be of any owner.
		The nodes falling for SYSTEM role are exception from this rule. For them the bit is ignored.
0	Deny	Denies operation of the identified category
1	Allow	Allows operation of the identified category

Basic ACL was designed to be processed and verified really fast. It's simple enough, but covers the majority of access restriction use cases, especially when combined with a carefully tailored Storage Policy.

There are well-known Basic ACLs:

Final – with a flag denying Extended ACL:

`private`: 0x1C8C8CCC

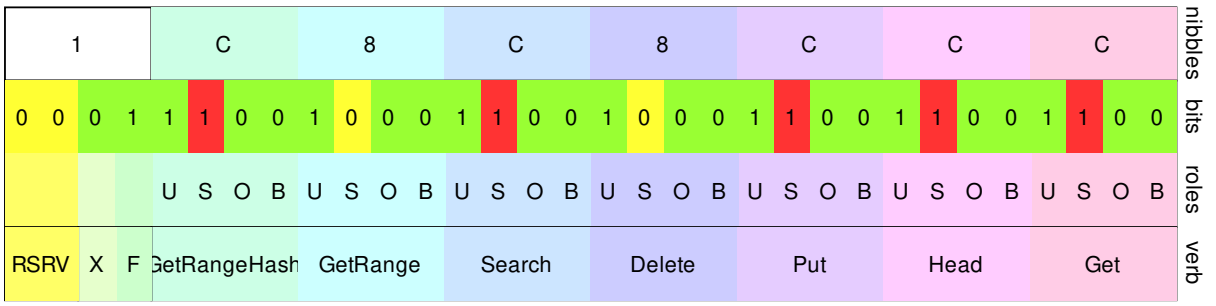

Figure 4: Basic ACL `private`

`public-read`: 0x1FBF8CFF

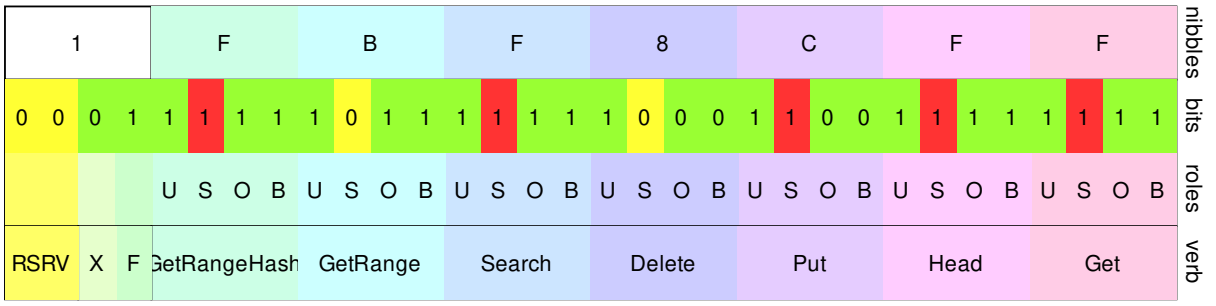

Figure 5: Basic ACL `public-read`

`public-read-write`: 0x1FBFBFFF

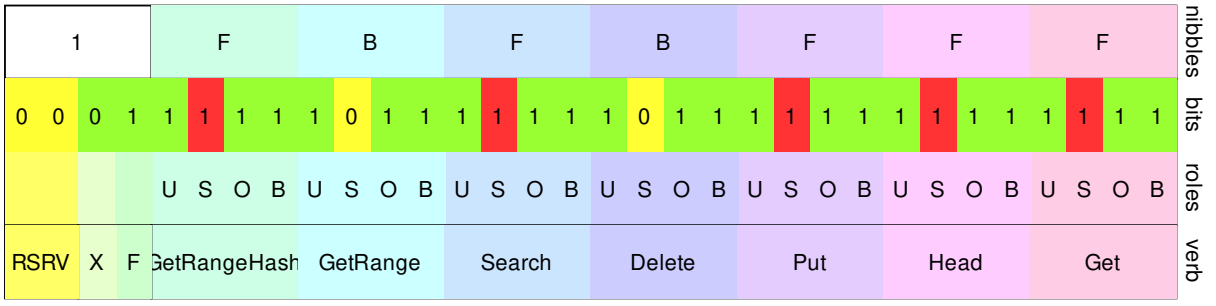

Figure 6: Basic ACL `public-read-write`

`public-append`: 0x1FBF9FFF

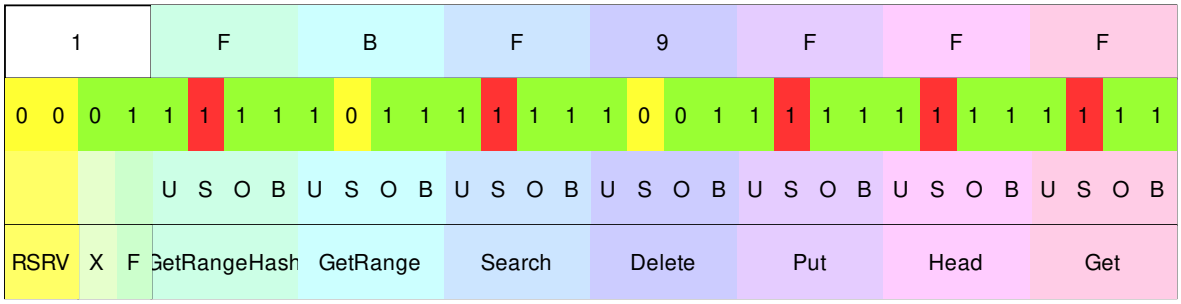

Figure 7: Basic ACL `public-append`

Non-final – Extended ACL can be set:

`eacl-private`: 0x0C8C8CCC

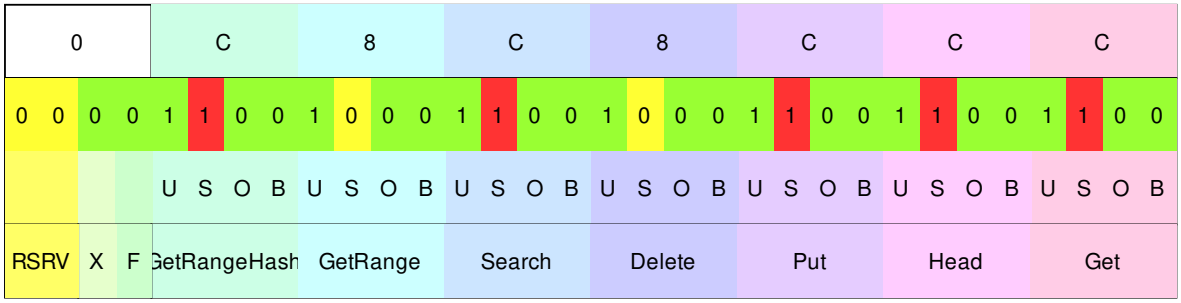

Figure 8: Basic ACL `eacl-private`

`eacl-public-read`: 0x0FBF8CFF

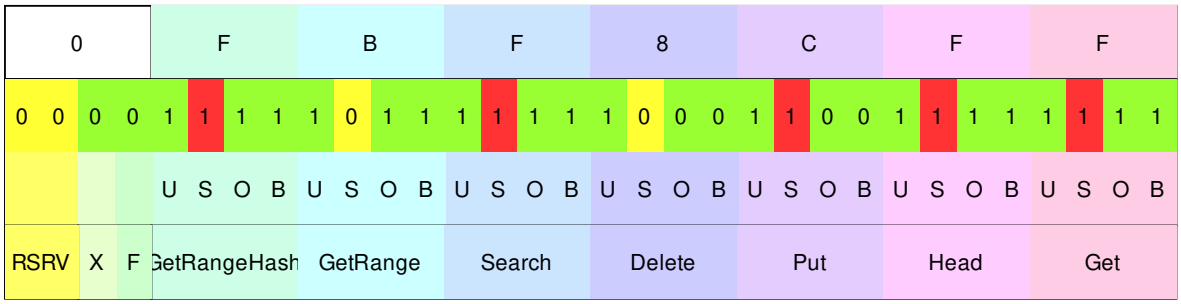

Figure 9: Basic ACL `eacl-public-read`

`eacl-public-read-write`: 0x0FBFBFFF

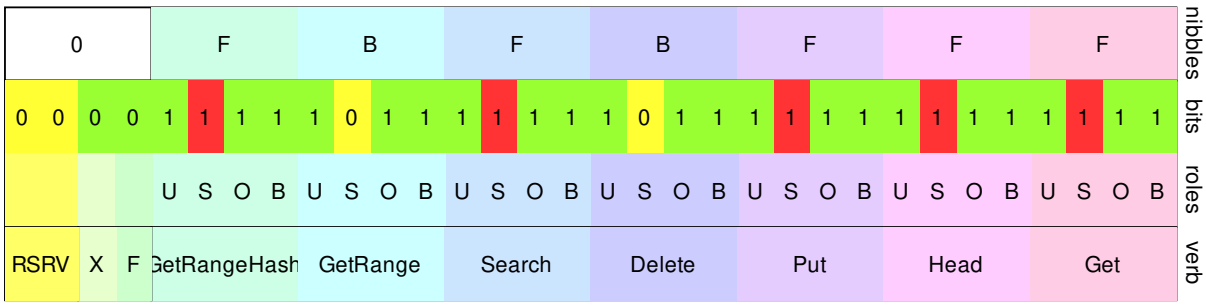

Figure 10: Basic ACL `eacl-public-read-write`

`eacl-public-append`: 0x0FBF9FFF

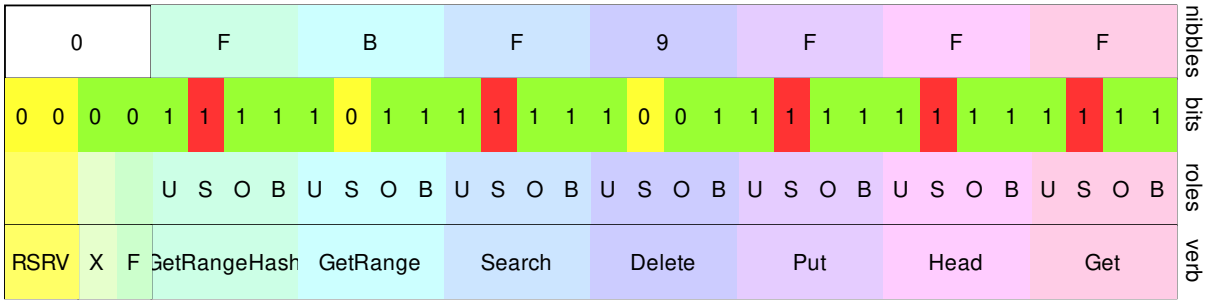

Figure 11: Basic ACL `eacl-public-append`

Extended ACL

Extended ACL is stored in the container smart contract in NeoFS Sidechain. This means it can be changed during container lifetime and there will be only one latest version of it in use. Only the container owner, or the bearer of a SessionToken with a Container context signed by the container owner, can change the Extended ACL rules. Since it is stored in a form of a stable serialized protobuf structure, eACL table can be only replaced with a new version, not altered or changed in-place in any way.

Extended ACL can only specify Basic ACL rules and make them more restitutive, but it can never ease them. Extended ACL rules can never conflict with Basic ACL rules or cancel them. If something is denied at Basic ACL level, it can never be allowed again by eACL. If Basic ACL contains Allow, eACL may specify the rule to a finite list of allowed keys and Deny all others. If Basic ACL already contains Deny, eACL can do nothing. Deny in Basic ACL cannot be changed to Allow in eACL. Therefore, the records with denied GET, GETRANGE, PUT, SEARCH, HEAD for `System` target must be ignored. This reduces to ignoring any `System` target rules.

When a user creates a container with the F-bit of Basic ACL set to 0, they do not need to settle the rules immediately. For a non-existing Extended ACL request, Container contract will return a null byte array.

It will be interpreted as a table with no rules.

To get the latest eACL version, a Storage Node needs to request it via RPC from the SideChain node. If an eACL can't be retrieved, the access permissions check fails.

Extended ACL rules get processed on-by-one, from the beginning of the table, based on the request operation, until matching the rule found. It means that there is no separate rule for setting denying or allowing policy. Final fallback rules must be provided by the user, if needed.

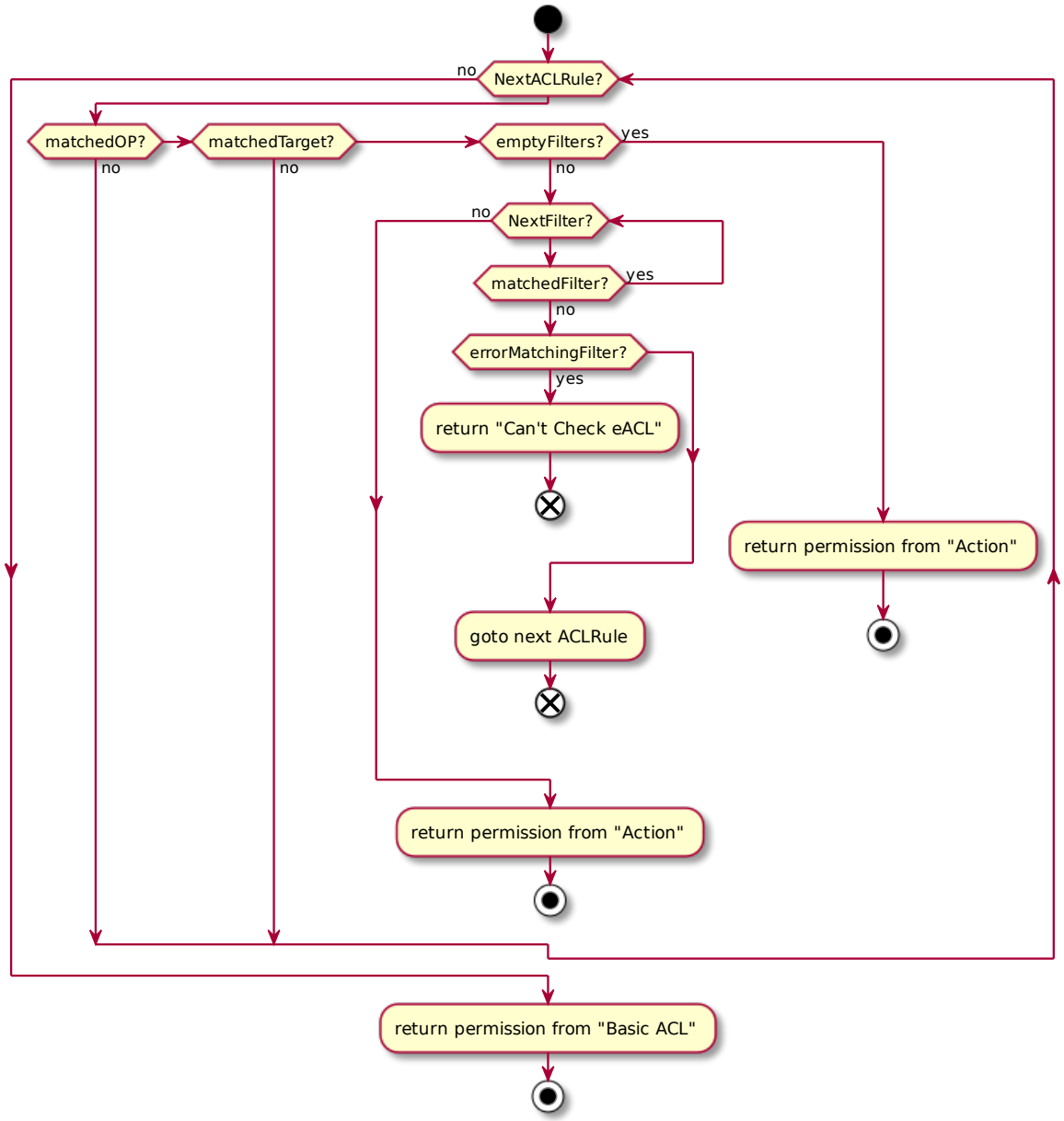

Figure 12: Extended ACL rules check

Extended ACL rules and table format may change depending on the version of NeoFS API used. Please see the corresponding API specification section for details.'

Each eACL rule record has four fields:

Field	Description
Operation	NeoFS request action verb
Action	Rule execution result action. Allows or denies access if the rule's filters match.
Filter	Filter to check particular properties of the request or the object
Target	Subject's role class or a list of public keys to match

and can be presented in different intermediate formats, like JSON, for the users' convenience.

```
{
  "records": [
    {
      "operation": "GET",
      "action": "DENY",
      "filters": [
        {
          "headerType": "OBJECT",
          "matchType": "STRING_NOT_EQUAL",
          "key": "Classification",
          "value": "Public"
        }
      ],
      "targets": [
        {
          "role": "OTHERS"
        }
      ]
    }
  ]
}
```

Note that some filters with `$Object` prefix are not suitable for making denying rules on certain operations. There may be an undefined behavior on some combinations of NeoFS verbs and object attributes when eACL is set. In the table below, + means allowed to be used and − means undefined behavior, hence not allowed.

$Object:	GET	HEAD	PUT	DELETE	SEARCH	RANGE	RANGEHASH
version	+	+	+	-	-	-	-
objectID	+	+	+	+	-	+	+
containerID	+	+	+	+	+	+	+
ownerID	+	+	+	-	-	-	-
creationEpoch	+	+	+	-	-	-	-
payloadLength	+	+	+	-	-	-	-
payloadHash	+	+	+	-	-	-	-
objectType	+	+	+	-	-	-	-
homomorphicHash	+	+	+	-	-	-	-
User headers	+	+	+	-	-	-	-

Let us make an example. `Delete` and `Range` operations are likely to show undefined behavior if `Head` has been denied for objects with particular `payloadLength`. They fail because they need to produce HEAD requests upon execution. If a user cannot `Head`, those operations cannot work properly. The full table of spawning object requests is given below.

Base/Gen	PUT	DELETE	HEAD	RANGE	GET	HASH	SEARCH
PUT	+	-	-	-	-	-	-
DELETE	+	-	+	-	-	-	+
HEAD	-	-	+	-	-	-	-
RANGE	-	-	+	+	-	-	-
GET	-	-	+	-	+	-	-
HASH	-	-	+	+	-	-	-
SEARCH	-	-	-	-	-	-	+

Also, note that user attributes cannot be used as filters in an eACL rule as it provokes an undefined behaviour. By design, when user attributes are set for a Complex Object, they are not inherited in the Part Objects and are only stored in the Link Object header. We cannot control the access for an Object of size more than `maxObjectSize`. To keep the system consistent we do not support eACL filters by

user attributes for small Objects as well.

Bearer Token

`BearerToken` allows to use the Extended ACL rules table from the token attached to the request, instead of the Extended ACL table from the `Container` smart contract.

Just like JWT[15], it has a limited lifetime and scope, hence can be used in the similar use cases, like providing authorization to externally authenticated party.

BearerToken can be issued only by the container owner and must be signed using the key associated with the container's `OwnerID`.

In the gRPC request, `BearerToken` is encoded in a protobuf format, but can be also presented in different intermediate formats, like JSON, for the users' convenience.

```
{
  "body": {
    "eaclTable": {
      "version": {
        "major": 2,
        "minor": 6
      },
      "containerID": {
        "value": "DIFWB4CFTayb9IAqeGwLGJdJfW6i5wWllPsF50EmazQ="
      },
      "records": [
        {
          "operation": "GET",
          "action": "ALLOW",
          "filters": [
            {
              "headerType": "OBJECT",
              "matchType": "STRING_EQUAL",
              "key": "Classification",
              "value": "Public"
            }
          ],
          "targets": [
            {
              "role": "OTHERS",
              "keys": []
```

[15] https://jwt.io

```
            }
          ]
        },
      ]
    },
    "ownerID": null,
    "lifetime": {
      "exp": "100500",
      "nbf": "1",
      "iat": "0"
    }
  },
  "signature": {
    "key": "AiGljnj41qh9o9uVqP9b9CArihHvXfGmljhAZNo4DceG",
    "signature": "BAwfdE1ZVL0LfREGkuXRKT2....GA="
  }
}
```

BearerToken format may change depending on the version of NeoFS API used. Please see the corresponding API specification section for details.

ACL check algorithm

NeoFS tries to start with local Basic ACL checks that are fast and cheap in terms of resource consumption. This should cover the vast majority of cases. Then, if present in the request, the ACL records from `BearerToken`, again locally. For the rest of complex cases, the Storage Node retrieves the Extended ACL table from the Container smart contract. Thereafter, the NeoFS ACL system may slow down the request processing only in complex cases when it's inevitable.

The resulting ACL check algorithm is the following:

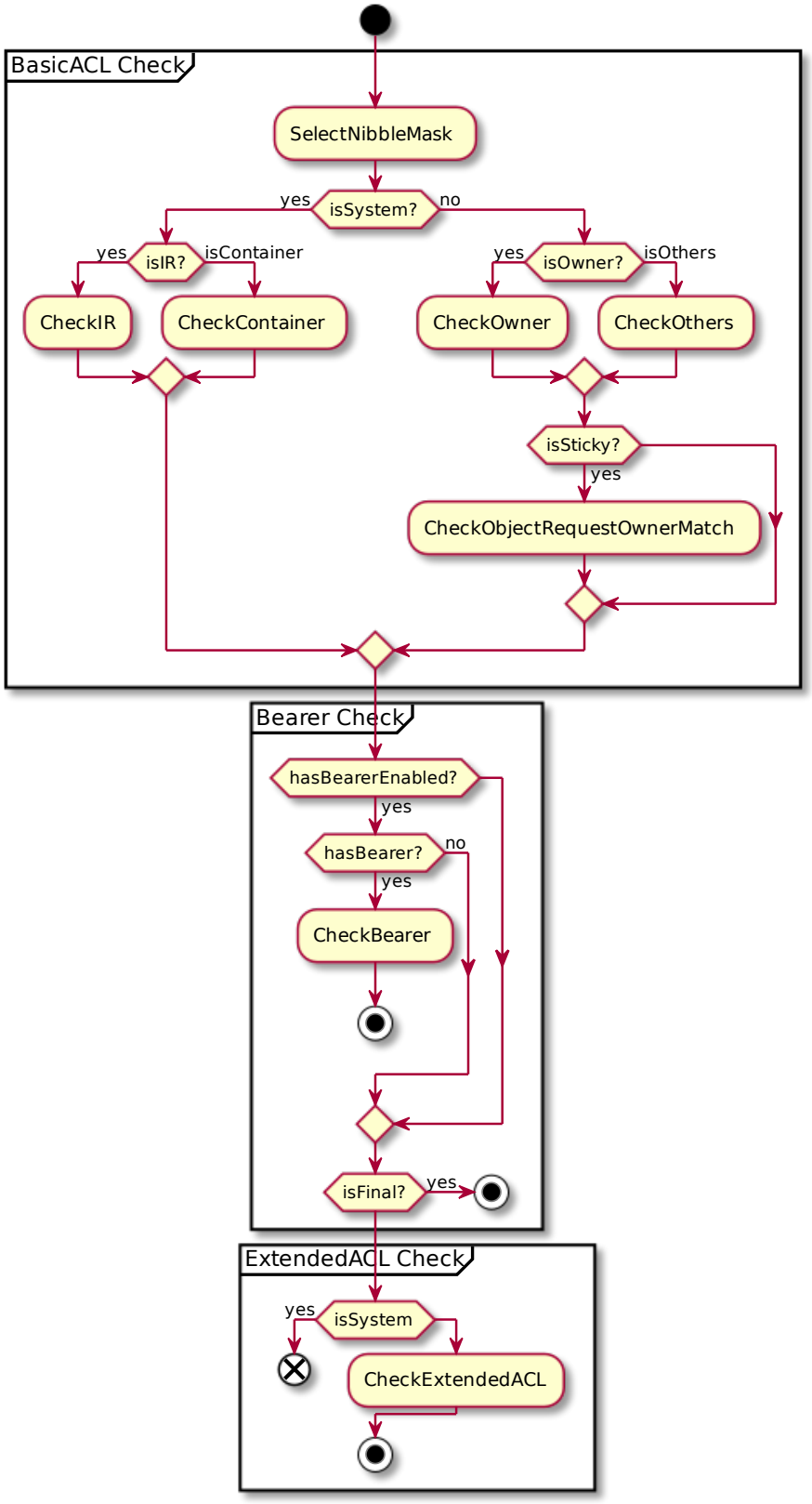

Figure 13: ACL check order

Reputation system

NeoFS reputation system is a subsystem for calculating trust in a node. It is based on a reputation model for assessing trust, which is, in turn, based on the EigenTrust algorithm designed for peer-to-peer reputation management. The algorithm ensures that there is a uniquely defined manager (parent node) for each network participant at each specified time (epoch). Based on the information received from its child node and other managers, it iteratively puts a complex general (Global) Trust of the entire network into the applicable child node.

The reputation system allows introducing nodes performance quality metric (trust). In a situation when most of the nodes make an honest assessment of the actions of other nodes, the system lets you calculate this metric quite accurately. This metric can be used for:

1. Filtering the list of nodes where the user intends to store information. For example, one can use only those nodes whose scores are higher than the minimum acceptable for the user.
2. Making a decision to exclude untrusted hosts from the network. For example, one can suspend a node in case it comes to the minimum level of quality in the network.

Trust

Trust in a NeoFS node is its quantitative (numerical) assessment based on the experience of interacting with that node. The higher the score is, the higher is the trust in the node and vice versa. Since the system uses a reputation-based trust model, the terms “trust” and “reputation” are considered synonymous in this document.

The Subject of trust assessment is the one who calculates trust.
The Object of trust assessment is the one whose trust is being calculated.

Reputation models are based on the nature of social media reputation. Thus, trust in a node is built up both by the estimates of the behavior of the node by another node and by the reputation of the node evaluating its behavior.

Trust in a NeoFS in a node is formed based on its interactions with the Subjects of trust assessment. Therefore, the reputation of a node changes during the NeoFS network working cycle when both the behavior of the node itself, and the behavior of other nodes change.

Algorithm

General problem statement: the Subject of assessment needs to calculate the reputation of the Object of trust assessment at a specific point of time (specific epoch).

EigenTrust is based on the notion of transitive trust: peer `i` will have a high opinion of those peers who have provided it with authentic information. Moreover, peer `i` is likely to trust the opinions of those peers, since peers who are honest about the information they provide are also likely to be honest in reporting their Local Trust values.

Global Trust is calculated in 3 main stages:

1. Each network member collects local statistics of network interactions with other peers, acting as the Subject of reputation assessment.
2. At the end of an epoch, each node announces its local statistics to its manager.
3. Managers exchange received information iteratively and, based on the updated data, make adjustments to the trust obtained in the previous iteration.

The algorithm uses configuration parameters that affect the result of the Global Trust calculation. To synchronize all network participants in terms of the values of these parameters, the nodes "read" these parameters from the `Netmap` contract.

Subjects and Objects of Trust in NeoFS

In NeoFS, the reputation system is used to calculate the trust in Storage Node. Thus, the Object of trust is always a Storage Node (and it is also the Subject in the local case). The Subject of Global Trust is the entire ring of Storage Nodes.

Inner Ring Nodes

Storage Nodes

Address format

NeoFS uses Multiaddress format in a human-readable string version as a Node address in `netmap`. Any new node must provide correct Multiaddress on bootstrap stage. After bootstrap, addresses are verified by `IR` before new Node is a part of `netmap`.

Correct address *composition* and *order*:

1. Network layer(`dns4`, `ip4` or `ip6`);
2. Transport layer(`tcp`);
3. Presentation layer(`tls`) - optional, may be absent.

Examples:

Correct

- `/dns4/somehost/tcp/80/tls`;
- `/ip4/1.2.3.4/tcp/80`.

Incorrect

- `/tcp/80/ip4/1.2.3.4`;
- `/tls/ip4/1.2.3.4/tcp/80`;
- `/ip4/1.2.3.4/dns4/somehost/tcp/80`.

Garbage Collector

Garbage Collector is a part of the Storage Engine. One GC instance runs on one shard.

GC should remove an object itself from Write Cache and Blobstor and the object's metadata from Metabase.

Upon a delete request, a Tombstone object is initialized. It contains all `ObjectID`s which the deleted Object was split into. A Tombstone belongs to the Container where the deleted Object is stored. In spite of the fact that the `__NEOFS__EXPIRATION_EPOCH` attribute is assigned to the Tombstone, its value is taken from the internal field that is specific to Tombstone object type. This action makes it easier for Storage Engine data scrubbers to search through the Tombstone indices and select ones for final physical deletion. By design, expiration epoch is strictly more than the current one and never equals to it. Once created, the Tombstone is put into the Container and thus is spread therein.

Since this moment, users cannot GET the object any more. If they try to GET the object, they will get the error.

When a Tombstone is obtained, the Storage Engine produces some meta information about it, which helps it track object removal. The Graveyard entity is responsible for storing that metadata. It is a Metabase table consisting of key-value pairs. The key is the deleted Object's ID and the value depends on how much time this Tombstone has already spent in the Graveyard: - it is a Tombstone Object ID when the Object has just (i.e., on the current epoch) been inhumed; - it is a special GCMark flag evaluated to "True" if Garbage Collector has marked the inhumed Object for total deletion.

Garbage Collector's routine per epoch includes two jobs: invalid Objects check and marked Objects removal.

Invalid Objects check

We consider an Object invalid if it is expired (i.e., `__NEOFS_EXPIRATION_EPOCH` well-known attribute value is equal to the current epoch) or the Tombstone associated with this object is expired.

If an expired Object is found, GC leads it through the deletion procedure described above.

If an expired Tombstone is found, the associated Graveyard record is updated: the aforementioned GCMark toggles for the inhumed Object.

Marked Objects removal

GC searches through the Graveyard and deletes Objects which have been previously GCMark'ed for deletion. It first removes all metadata associated with this objectID and then removes the Object and the related Tombstone from Blobstor, Write Cache and filesystem.

This procedure is the same for any object type, i.e. a Storage Group removal goes through the mentioned stages as well as a Regular Object.

Object Expiration

In NeoFS, Objects may have an "expiration date". When an Object expires, it is marked for deletion and isn't available anymore. There is a well-known `__NEOFS__EXPIRATION_EPOCH` attribute which specifies the expiration date. Only a Regular Object may expire.

A Tombstone object is created upon Regular Object or Storage Group deletion. Every Tombstone object has the `__NEOFS__EXPIRATION_EPOCH` attribute as well. Thereby Storage Engine is able to filter Tombstones and select ones for total cleanup.

This attribute for a Tombstone Object is set automatically upon its creation. In case of a Regular Object, a user sets it manually.

Notifications

Storage nodes can produce notifications about internal events for external listeners. The specification covers only basic concepts of notifications and their triggers and does not define implementation details such as message binary format or transport protocol. Such details may vary in storage nodes depending on external listeners. Notifications do not affect core protocol and can be disabled. Therefore, one can use it only in a controlled environment with access to storage node configuration.

Notification is the entity that consists of **topic** and **message**.

Object notifications

Stored objects can trigger notifications. Object triggers notification if it contains valid well-known object header `__NEOFS__TICK_EPOCH`. Read more about well-known headers in the NeoFS API v2 section.

When the storage node processes a new epoch event with an epoch number specified in `__NEOFS__TICK_EPOCH`, it should produce a notification related to such objects. If `__NEOFS__TICK_EPOCH` header specifies zero epoch, then the notification should be produced immediately as an object saved in the storage engine.

Notification **message** should contain the address of the object. Notification **topic** is defined by valid well-known object header `__NEOFS__TICK_TOPIC`. If the header is omitted, the storage node should use the default topic. Default topic is defined by storage node implementation.

Protocol gateways

HTTP

S3

NeoFS S3 gateway[16] provides API compatible with Amazon S3 cloud storage service.

Access Box scheme

S3 gateway has to authenticate user requests regarding AWS spec[17]. So we have the following scheme:

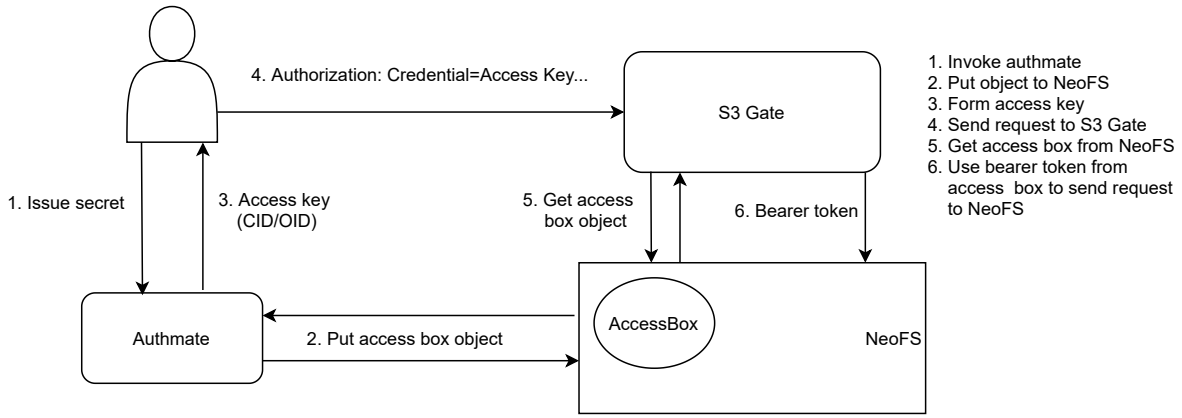

Figure 14: Access box scheme

1. A user uses the neofs-s3-authmate[18] (Authmate) tool to get credentials (`access_key_id` and `secret_access_key`).
2. Authmate forms “AccessBox” object (see the next section) and puts it into NeoFS.
3. Authmate output contains credentials (`access_key_id` and `secret_access_key`) that can be used with AWS CLI, for example.
4. The user sends request to NeoFS S3 Gateway using standard AWS tool.
5. S3 Gateway gets “AccessBox” from NeoFS by `access_key_id` and fetches Bearer Token from it.
6. S3 Gateway uses fetched Bearer token to send a request to NeoFS on behalf of the user.

[16] https://github.com/nspcc-dev/neofs-s3-gw
[17] https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html
[18] https://github.com/nspcc-dev/neofs-s3-gw/blob/master/docs/authmate.md

Form Access Box object Actually, `AccessBox` is a regular object in NeoFS but properly formed. It contains an encrypted Bearer token (BT), Session tokens (STs) and a `secret_access_key`.

The credentials are formed by the following steps:

1. User provides Authmate with BT, STs and `s3gw_public_key_k`(the public keys of the gates that will be able to handle credentials) that will be used when the user sends requests via S3 Gateway.
2. Authmate:
    1. Generates a `secret_access_key` (it's 32 random bytes) and P256 (secp256r1) key pair (`authmate_private_key`, `authmate_public_key`).
    2. Forms a `tokens` protobuf struct that contains BT, STs, `secret_access_key`
    3. For each `s3gw_public_key_k` derives a symmetric key `symmetric_key_k` using ECDH[19] and encrypts `tokens` (`encrypted_tokens_k = encrypt(tokens.to_bytes(), secret_key_k, nonce_k)`).
    4. For each `encypted_tokens_k` forms a struct called `gate_k` that contains `encrypted_tokens_k` and `s3gw_public_key_k`.
    5. Forms the final binary object `AccessBox` that contains `authmate_public_key`, `gate_1`, ..., `gate_k`.
    6. Puts the `AccessBox` object to NeoFS and saves its address `CID/OID` as `access_key_id`.
    7. Returns the pair (`access_key_id`, `secret_access_key`) to the user.

Handle S3 request On getting a request, S3 Gateway:

1. Fetches the `access_key_id` from the `Authorization` header.
2. Gets `AccessBox` from NeoFS by address (recall `access_key_id` is `CID/OID`)
3. Using `s3gw_private_key_k` and `authmate_public_key` derives `symmetric_key_k` and decrypts `encrypted_token_k` that has been found by matching `s3gw_public_key_k` in `gate_k` struct.
4. Checks the signature of the initial request using `secret_access_key` from `tokens` struct.
5. Uses BT and STs to perform requests to NeoFS.

sFTP

[19] https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman

Data Audit

In the case of a large number of objects in a distributed network of untrusted nodes with an ever-changing topology, the classical approach is to compare objects' hashes with some sample in a central meta-data storage. This method is not efficient enought. It causes unacceptable overhead and leads to data disclosure.

To solve this problem, NeoFS uses Homomorphic hashing. It is a special type of hashing algorithms that allows computing the hash of a composite block from the hashes of individual blocks. NeoFS has a focus on a probabilistic approach and homomorphic hashing to minimize network load and avoid single points of failure.

NeoFS implements Data Audit as a unique zero-knowledge multi-stage game based on homomorphic hash calculation without data disclosure. Data Audit is independent of object storage procedures (recovery, replication, and migration) and respects ACL rules set by user.

For integrity checks, NeoFS calculates a composite homomorphic hash of all the objects in a group under control and puts it into a structure called Storage Group. During integrity checks, NeoFS nodes can ensure that hashes of stored objects are correct and are a part of that initially created composite hash. This can be done without moving the object's data over the network; and no matter how many objects are in a Storage Group, the hash size is the same.

Storage Groups

The concept of a storage group has been introduced to reduce the dependence of the check complexity on the number of objects stored in the system.

The consistency and availability of multiple objects on the network are achieved by validating the storage group without saving meta information and performing validation on each object.

StorageGroup keeps verification information for Data Audit sessions. Objects that require paid storage guaranties are gathered in `StorageGroups` with additional information used for proof of storage checks. A `StorageGroup` can be created only for objects from the same container.

A `StorageGroup` is a group of objects of a special type with a payload containing the serialized protobuf structure. For more details on the format, please refer to the API specification in the corresponding section.

StorageGroup structure has information about:

- Total size of the payloads of objects in the storage group
- Homomorphic hash from the concatenation of the payloads of the storage group members. The order of concatenation is the same as the order of the members in the `members` field.

- Last NeoFS epoch number of the storage group lifetime
- Alpha-numerically sorted list of member objects

Data Audit cycle

Data Audit cycle is triggered by Epoch change. InnerRing nodes share the audit work between themselves and do as much audit sessions as they can. On the next round, if InnerRing nodes can't process everything, new nodes can be promoted from the candidate list. On the opposite, if the load is low enough, some InnerRing nodes can be demoted to maintain the balance.

Data Audit Game

Each Epoch, Inner Ring nodes perform a data audit cycle. It is a two-stage game in terms of the game theory. At the first stage, nodes serving the selected container are asked to collectively reconstruct a list of homomorphic hashes that form a composite hash stored in the Storage Group. By doing that, nodes demonstrate that they have all necessary objects and are able to provide hashes of those objects. The provided list of hashes can be validated, but at the current stage it's not known whether some nodes are lying.

At the second stage, it is necessary to make sure nodes are honest and do not fake check results. The Inner Ring nodes calculate a set of node pairs that store the same object and ask each node to provide the homomorphic hashes of that object. Ranges are chosen in a way that the hash of a range asked from one node is the composite hash of ranges asked from another node in that pair. Nodes cannot predict objects or ranges that are chosen for the data audit session. They cannot even predict a pair node for the game. This stage discovers malicious nodes fast because each node is serving multiple containers and Storage Groups and participates in many data audit sessions in parallel during same Epoch. When a node is caught in a lie, it gets a reputation penalty and loses any rewards for the Epoch. So the price of faking checks and risks are too high and it is easier and cheaper for a node to be honest and behave correctly.

Combining the fact of nodes being able to reconstruct the Storage Group's composite hash and the fact of nodes honest behavior, the system can consider that the data is safely stored, not corrupted, and available with a high probability.

In the case of a successful data audit result, the Inner Ring nodes initiate microtransactions between the accounts of the data owner and the owner of the storage node invoking the smart contract in the NeoFS N3 Sidechain.

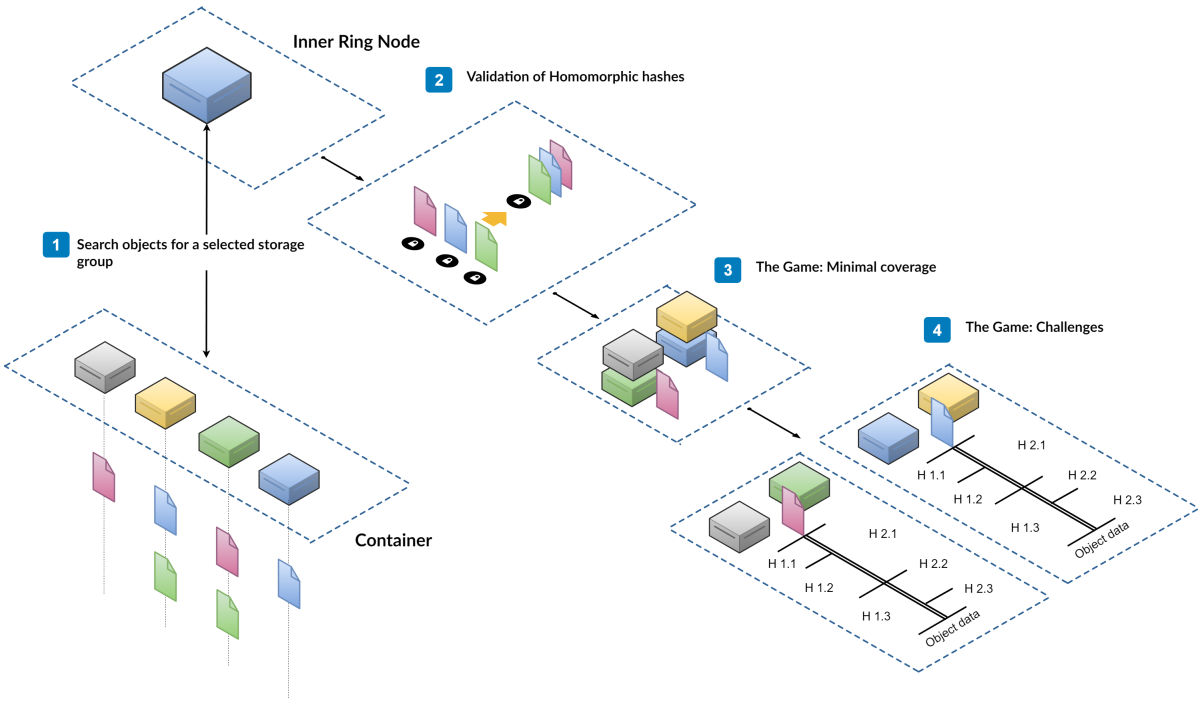

Figure 15: Data Audit

Audit tasks distribution

InnerRing nodes select containers to audit from a list of all containers in the network, forming a ring of containers and taking an offset shackle with its number among InnerRing nodes and by audit number.

Each epoch, Inner Ring node performs data audit. One audit task is a one storage group to check. Storage groups from one container get merged into a single audit result structure that will be saved in the Audit smart contract in NeoFS Sidechain.

Upon each new Epoch notification, Inner Ring node must:

1. Check amount of unfinished audit tasks from queue, log it and flush it
   NeoFS uses these values to initiate Inner Ring list growth or shrink.
2. Publish all unfinished audit results asynchronously
   Audit results should be published on step 5 when all tasks for a single container are done. If a new epoch happens, Inner Ring node should wait for all active tasks to finish and then publish incomplete audit results.
3. Choose new tasks to process
   Inner Ring lists all available containers from the container contract. Additionally it can make

extra invoke to find out container complexity, but for now consider all containers are the same. Then, based on index and epoch number, Inner Ring node chooses a slice of the containers to check. For each container it searches storage groups and put them in a task queue.

4. Run these tasks in a separate fixed size routine pool
   Audit checks run simultaneously in a separate pool of routines. This pool has fixed size, e.g. 3 tasks at a time. When a new epoch happens, inner ring uses a new pool of routines. Previous routine pool is alive until running tasks from the previous epoch are finished (we don't discard already started tasks).
5. Merge task results in audit result structures and publish them if there are no tasks left in the container
6. Optionally dump task results to a file

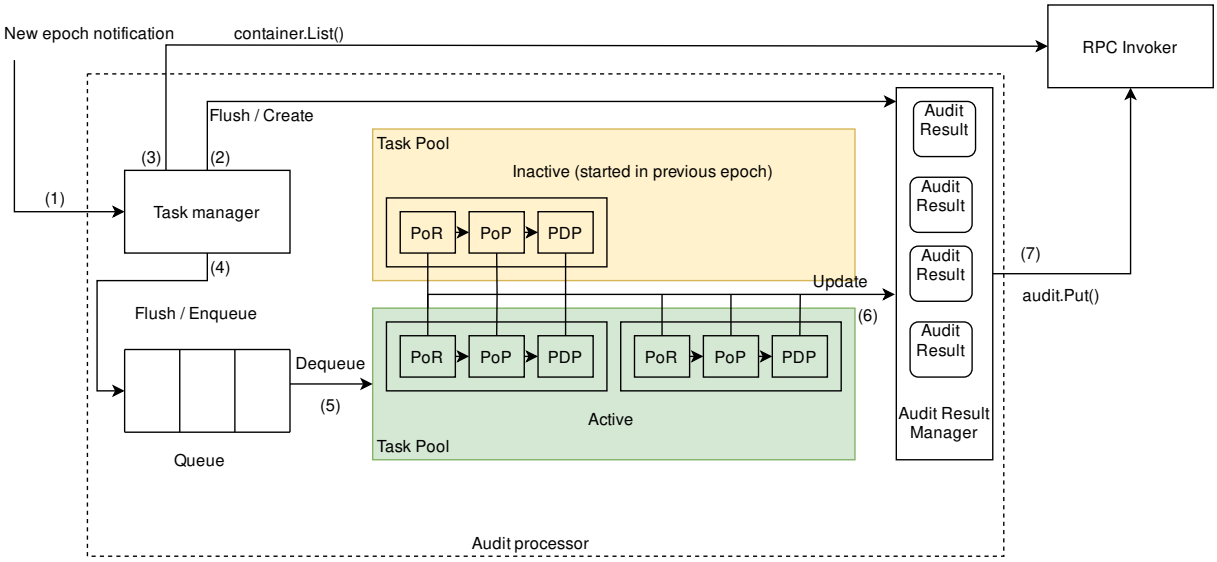

Figure 16: Audit processor

Data Audit session

For each selected container Inner Ring node will:

- Generate a list of storage groups (object `SEARCH + GET`);
- Check each group;
- Record the results of all groups

The check of each group has three stages:

- Prove-of-Retrievability (PoR);
- Prove-of-Placement (PoP);

- Prove-of-Data-Possession (PDP).

Prove-of-Retrievability

During PoR check inner ring node should:

1. Get storage group object
2. For each member of a storage group, Inner Ring node makes HEAD request with `main_only` flag
3. Compare cumulative object size and homomorphic hash with the values from step 1
4. Depending on step 3, save storage group ID in a list of succeeded or failed checks in audit result

Prove-of-Placement

At this stage Inner Ring tries to create pair-coverage for all nodes in container. Later these pairs will play a game based on homomorphic hash properties (PDP check).

To do so Inner Ring:

1. Picks random member X from the storage group
2. Builds placement vector for X
3. Makes HEAD request with `TTL=1 RAW=true main_only=true` to these nodes in placement order until there are enough responses or no more container nodes
4. Increments `HIT` counter in audit result if responses from step 3 are without single failure
5. Increments `MISS` counter in audit result if there are enough responses from step 3 but with intermediate failures
6. Increments `FAIL` counter in audit result if there are not enough responses from step 3
7. Gets pair of nodes that returned result in step 3 and mark them covered,
8. Repeats everything from step 1, but ignore objects with placement in step 2 that does not increase coverage in step 7.

Here, enough responses means a number of copies according to container policy.

Prove-of-Data-Possession

For all pairs after PoP, a Prove-of-Data-Possession is performed:

- If a node from a pair loses the game, it gets into the “lucky” list,
- If a node from a pair wins the game, it gets into the “unsuccessful” list.

Hash check

For the hash check phase, the Inner Ring node gets the object information using HEAD request with the `short` flag toggled. With the size of the object known, the entire payload range is divided into four parts of a random length.

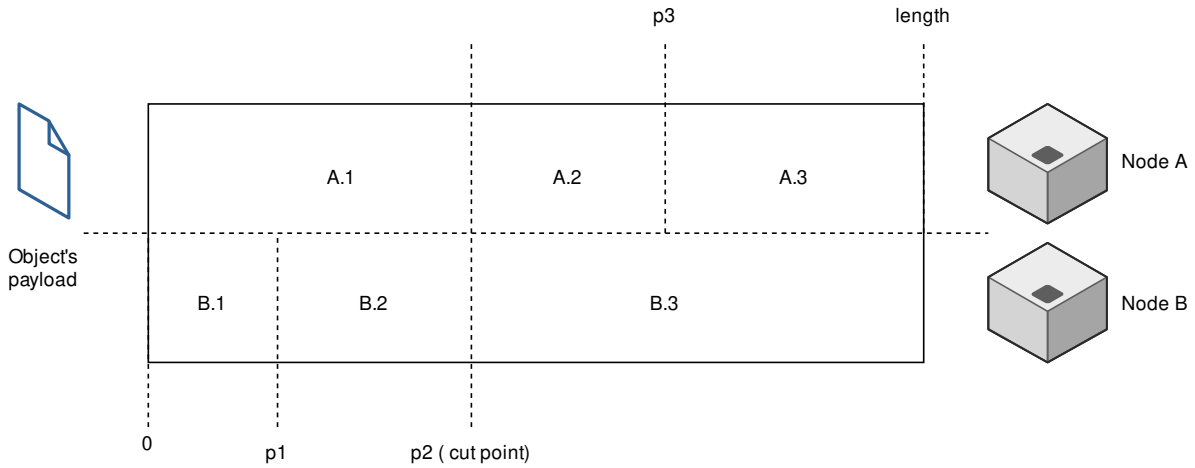

Figure 17: Hash checking challenge

Next, hashes of ranges are requested. One request per range, with random delay, from both nodes.

Node A:

$$hash_{A.1} = TZHash(Range(0, p2)),$$

$$hash_{A.2} = TZHash(Range(p2, p3 - p2)),$$

$$hash_{A.3} = TZHash(Range(p3, length - p3))$$

Node B:

$$hash_{B.1} = TZHash(Range(0, p1)),$$

$$hash_{B.2} = TZHash(Range(p1, p2 - p1)),$$

$$hash_{B.3} = TZHash(Range(p2, length - p2))$$

Once the hashes obtained successfully, the check considered passed if:

$$hashA.1 = hashB.1 + hashB.2,$$

$$hashB.3 = hashA.2 + hashA.3,$$

$$hashA.1 + hashA.2 + hashA.3 = hashB.1 + hashB.2 + hashB.3 = objecthash$$

Blockchain components

Role of blockcahin in the storage system

NeoFS stores data off-chain, on Storage Nodes. Clients access it directly in a peer-to-peer fashion. This allows to maintain the quality of service (read\write speed, big data volumes) at the level of traditional storage systems. That said, NeoFS is a global-scale decentralized network and it needs a proven mechanism to serve as a source of truth and global state and remain at that scale. That's what we use the Neo blockchain for.

The Inner Ring nodes take care of the entire network health. They control the network map of Storage Nodes, manage user containers and access control lists, regulate financial operations and data audit. Therefore, Inner Ring should do the following:

- make decisions according to consensus;
- be fault-tolerant;
- have synchronized global state available to Storage Nodes;
- minimize the amount of p2p connections to protect from direct attacks.

All these requirements can be fulfilled by using the blockchain as a distributed database with business logic implemented in smart contracts. Inner Ring operations must be confirmed and audited, which makes it reasonable to use smart contract memory to store the global state of the storage system. It's what makes Inner Ring applications stateless, lightweight, and scalable.

Meanwhile, NeoFS contracts:

- store the state of the current epoch, network map, audit results, reputation estimations, and container-related data;
- manage the economy;
- control the governance of the NeoFS network.

Mainchain and sidechain

Business logic of smart contracts is executed at contract method invocations from Inner Ring nodes. They are either multisigned transactions or regular transactions. However, these invocations are paid and require GAS Utility Token for computation. If data owners are charged for these computations, it will be economically impractical for them to use NeoFS.

To solve this problem, we divide smart contract operations into two types:

- financial transactions and governance,

- storage system associated operations.

The first type of operations should be executed in the mainchain. These operations are quite rare and they require GAS as a payment asset. The mainchain for NeoFS is N3 Main Net. Thus, `NeoFS` contract is deployed in the mainchain. It allows to make a deposit, update network config, and reregister the Inner Ring candidate node. The mainchain also manages Alphabet nodes of the Inner Ring using the RoleManagement contract.

The second type of operations can be executed on an additional chain that we call the sidechain. Sidechain is the Neo blockchain with a different network configuration. Validator nodes of the sidechain are Alphabet nodes of the Inner Ring. Sidechain GAS is managed by Alphabet contracts and should be used only for network maintenance and NeoFS contract execution. Sidechain has `Alphabet`, `Audit`, `Balance`, `Container`, `NeoFSID`, `Netmap`, `Reputation` contracts. No other contracts unrelated to NeoFS can be deployed in the sidechain.

Inner Ring nodes synchronize states of mainchain and sidechain contracts by listening to the notification events and reacting to them.

Notary service

To make decisions according to the consensus, transactions must be multisigned by Alphabet nodes of the Inner Ring. Inner Ring nodes, however, do not have a p2p connection to each other. Therefore, multi-signature check is replaced with on-chain invocation accumulation. NeoFS contracts await for 5 out of 7 method invocations from Alphabet nodes of the Inner Ring. It leads to:

- increased number of transactions in the network;
- inability to calculate the exact price of a transaction, which leads to an increased cost of execution;
- the contract logic complication.

While in the sidechain6 GAS is used only for utility purposes and invocation prices can be mostly ignored, Alphabet nodes of the Inner Ring in the mainchain use a real GAS asset to do withdrawal operation. High execution cost of such operation leads to high withdrawal commissions.

To solve this issue, NeoFS supports notary service[20]. The notary service allows to build multisigned transactions natively in the blockchain. Thus, the number of transactions and their costs are reduced, contract source code is simplified. Special `proxy` (in the sidechain) and `processing` (in the mainchain) contracts can pay for invocation instead of Alphabet nodes of the Inner Ring. With these contracts, it becomes easier to monitor and control the NeoFS economy.

[20] https://github.com/neo-project/neo/issues/1573#issuecomment-704874472

NeoFS Sidechain Governance

NeoFS uses the sidechain as a database to store meta-information about the network: the network map, audit results, containers, key mappings, network settings, and several supplementary things.

The sidechain operates on the same principles as the mainnet – there are no free transactions, the committee chooses the consensus nodes, etc. This structure provides a number of advantages. For example, one can use the same N3 tool stack to work with NeoFS sidechain information.

To effectively work with the sidechain, we need to solve the following problems:

- How can the mainnet committee control Inner Ring nodes and the consensus nodes of the sidechain?
- How do Storage Nodes and Inner Ring nodes get sidechain GAS Utility Token to send transactions?

NeoFS Governance model solves these problems with seven “Alphabet” sidechain contracts and the first seven Inner Ring nodes bound to those contracts, acting as the sidechain committee.

Alphabet contracts

Alphabet contracts are seven smart contracts deployed in the sidechain. They are named after the first seven Glagolitic[21] script letters: Az(), Buky(), Vedi(), Glagoli(), Dobro(), Yest(), Zhivete(). These contracts hold 100,000,000 sidechain NEO Token on their accounts (approximately 14,285,000 for each). By storing NEO Token on the contract accounts, we protect it from unauthorized use by malicious sidechain nodes. Contracts do not transfer NEO and use it to vote for sidechain Validator nodes and to emit GAS Utility Token.

Alphabet Inner Ring nodes

Alphabet Inner Ring nodes are the first seven nodes in the Inner Ring list that are logically bound with one-to-one relation to the Alphabet contracts. They are the voting nodes, tasked with making the decisions in the NeoFS network. All other Inner Ring nodes take care of Data Audit, Storage Node attribute verification, and other technical tasks.

Being an Alphabet node implies running the sidechain Consensus Node using the same key pair as the NeoFS Inner Ring node instance. Hence, an Alphabet node candidate must:

- Setup a NeoFS Inner Ring node instance

[21] https://en.wikipedia.org/wiki/Glagolitic_script

- Setup a NeoFS sidechain full node using same key pair
- Register the same key in mainnet NeoFS Inner Ring candidates list
- Register the same key in sidechain committee candidates list

Alphabet contracts invocation

Contracts cannot distribute the utility token or vote by themselves. To perform these operations, Inner Ring nodes invoke alphabet contract methods. An Alphabet Inner Ring node can invoke its corresponding contract only. One node invokes one contract.

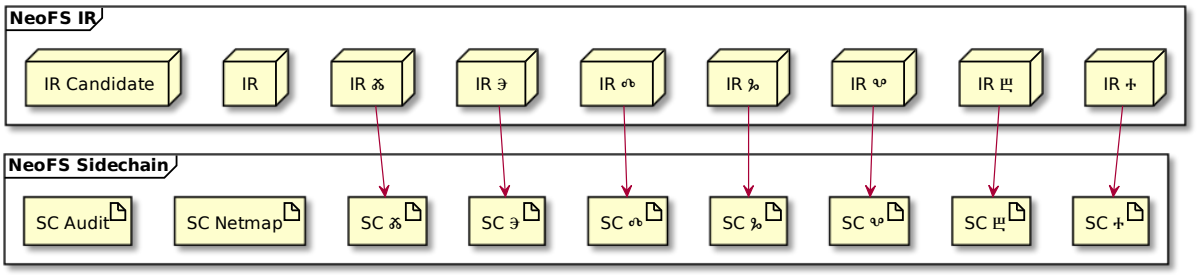

Figure 18: Inner Ring to Alphabet SC relation

Alphabetic contracts have hardcoded indexes. Contracts authenticate the method invoker by using the list of Inner Ring node keys from the Netmap Smart Contract. This scheme helps to limit malicious Alphabet Inner Ring node actions and makes network more resiliant to Inner Ring nodes losses.

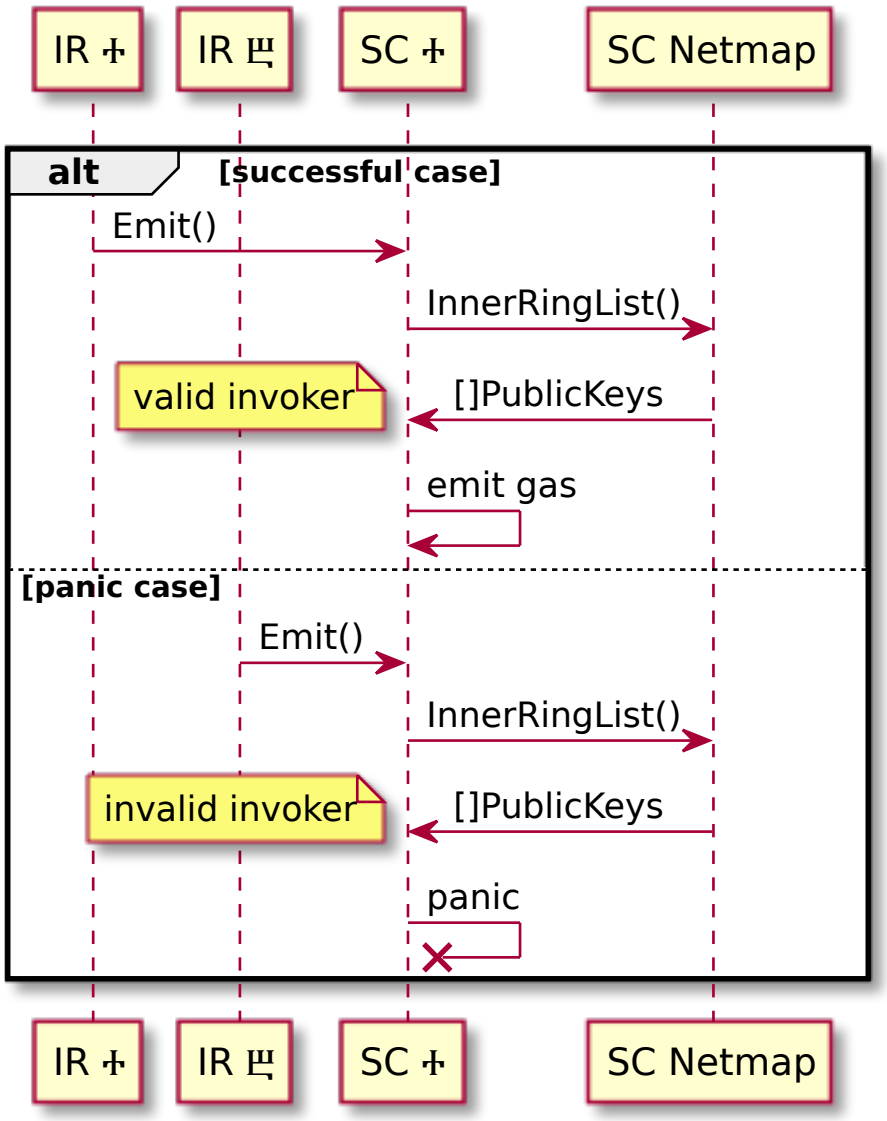

Figure 19: Alphabet SC invocation by Inner Ring nodes

Utility token distribution

Inner Ring nodes invoke `Emit()` method in corresponding alphabetical contracts. This method transfers all it's NEO Token to it's account, thereby producing utility token emission. Within the same invocation context, the contract transfers a share of the available GAS Utility Token to all Inner Ring wallets. Each contract will keep the $\frac{1}{8}$'th part on its balance as an emergency reserve.

$$InnerRingNodeEmission = G \cdot \frac{7}{8} \cdot \frac{1}{N}$$

G - contract's GAS Utility Token amount
N - length of Inner Ring list

After receiving GAS Utility Token, the nodes of the Inner Ring can periodically transfer a share to all registered Storage nodes and use the received utility token for sidechain operations: change epochs, register new containers, save data audit results, etc.

Storage nodes have a limited supply of GAS Utility Token to prevent malicious actions and DoS attacks. Depending on Storage Node activity and reputation records, it will receive a different utility token amount, normally enough to perform all required operations. Sidechain GAS and mainnet GAS are different tokens, hence Storage Nodes don't spend rewards for internal operations and can't convert the sidechain utility token into mainnet GAS Utility Token or vice versa.

Changing sidechain validators

Beforehand, Alphabet Inner Ring node candidates register validator keys in the list of candidates for the sidechain committee. When the sidechain Netmap smart contract sends a notification regarding Inner Ring node list updates, Alphabet Inner Ring nodes invoke the `Vote([]keys)` method of all Alphabet smart contracts in order to gather signatures and then make them vote for the sidechain Committee. Each Alphabet contract votes for the keys proposed by sending `VotesPerKey` votes for each key. Normally, there is just one key per node, hence `N` equals `1`.

$$VotesPerKey = \frac{A}{N}$$

A - contract's NEO amount
N - length of proposed keys list

Changing the Inner Ring list

Inner Ring nodes follow a self-regulation process, allowing them to vote to substitute dead or malfunctioning nodes with new ones from the candidate list. Only Alphabet nodes prepare new Inner Ring node lists and vote for it, but all nodes listed there must confirm their participation via the same voting mechanism.

The voting procedure uses the sidechain `Voting` smart contract, but the list of candidates is taken from mainnet NeoFS contract. When Inner Ring nodes agree on the updated list, it's submitted to the mainnet NeoFS smart contract and then mirrored back to Netmap smart contract on sidechain.

By using the `Emit()` and `Vote()` methods of Alphabetic smart contracts, Inner Ring nodes take full control of the sidechain. They control validator keys and utility token distribution. Thus, if the

mainnet committee will control list of Inner Ring nodes, then it will control sidechain as well.

The mainnet Committee can set the list of priority candidates for Alphabet Inner Ring nodes in the mainnet `DesignationContract`. Nodes from that list will be voted for becoming Alphabet Inner Ring nodes and substitute current Alphabet nodes, if they confirm the following requirements:

- Node's key is registered as a candidate in mainnet NeoFS smart contract
- Node's key is registered as a sidechain committee candidate
- Node is not listed as inactive in sidechain `Netmap` contract

The `DesignationContract` list may contain any number of valid candidates, and the voting process will make sure as many of them as possible are in the first seven active inner Ring nodes. If there is not enough appropriate candidates, the rest will be taken from the regular candidates list. If there are too many, only the first seven suitable nodes will be used.

The voting algorithm is the same for each Inner Ring node and starts in the following cases:

- New Epoch
- Notification from mainnet DesignationContract on Inner Ring nodes list change
- Notification from sidechain Netmap contract on inactive Inner Ring nodes list change

All Inner Ring nodes listen for notifications from the Voting contract. If they see themselves in the new Inner Ring nodes list, they confirm their participation by sending the same list in the `Prepare()` method. Only newly added nodes need to confirm their participation with a transaction. If the node is already in the active Inner Ring list, it doesn't need to send a confirmation.

When there are enough Alphabet signatures and all required candidate signatures have been sent with the `Prepare()` method, the last invocation will update the list and finish voting round.

Active Alphabet Inner Ring nodes will be waiting for the round to end and locally test invoke the `EndRound()` method. When the voting round timeout occurs and the round has not finished successfully through agreement on a new list, one of the Alphabet nodes will invoke `EndRound()` and settle the round's results.

If by the end of the voting round some newly added nodes haven't confirmed their participation, they are added to the Netmap smart contract's inactive list. This will trigger a new voting round without those inactive nodes.

If there are not enough candidates, Inner Ring nodes will accept the best list they can gather.

When the new list is agreed, the `Voting` smart contract sends a notification. All Alphabet nodes react with invocation of `UpdateInnerRing()` on the mainnet NeoFS smart contract. When the majority of Alphabet nodes send the update and mainnet list is updated, it will be mirrored by Alphabet Inner Ring nodes in the sidechain `Netmap` smart contract.

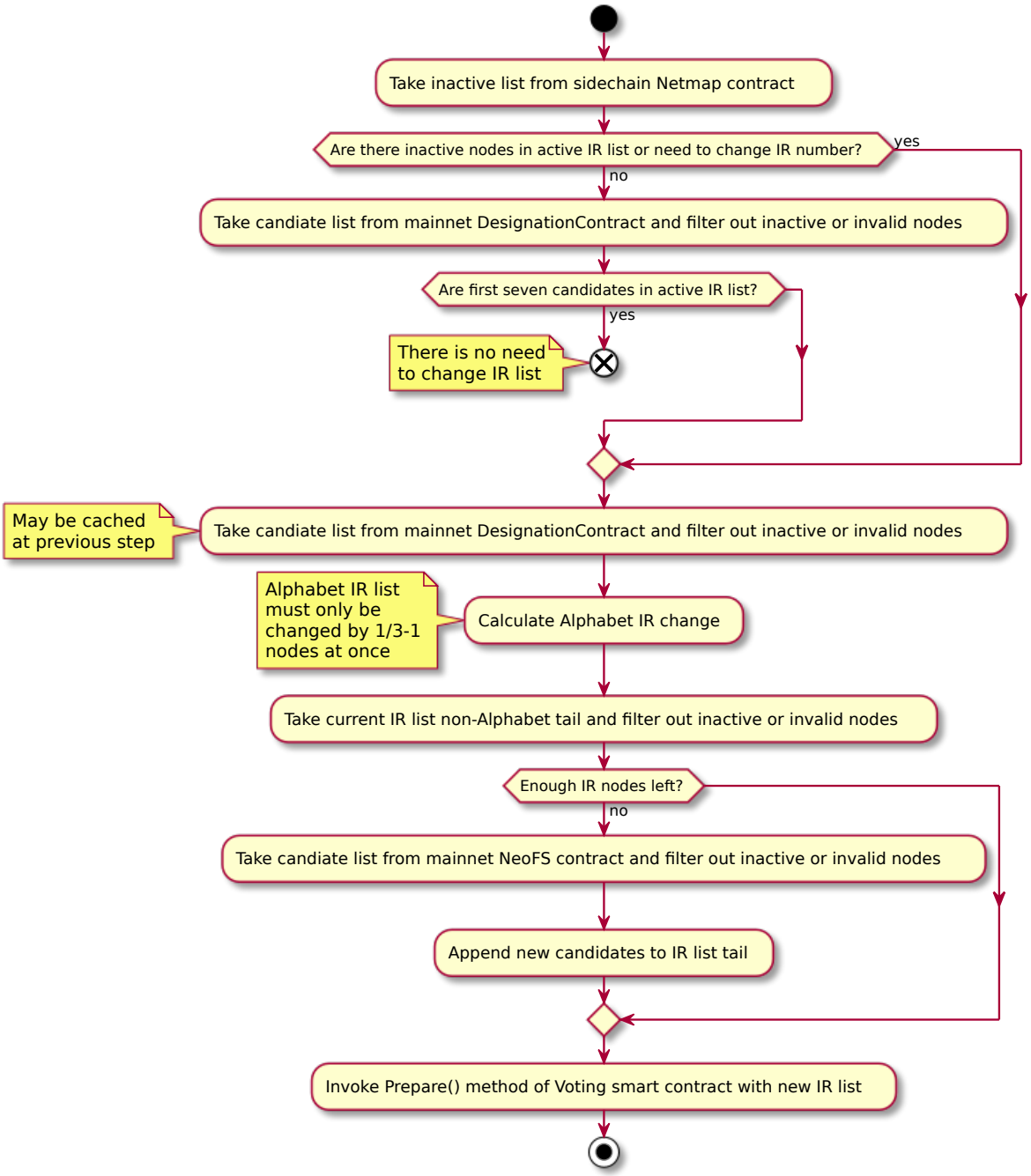

Figure 20: Inner Ring Alphabet node voting algorithm

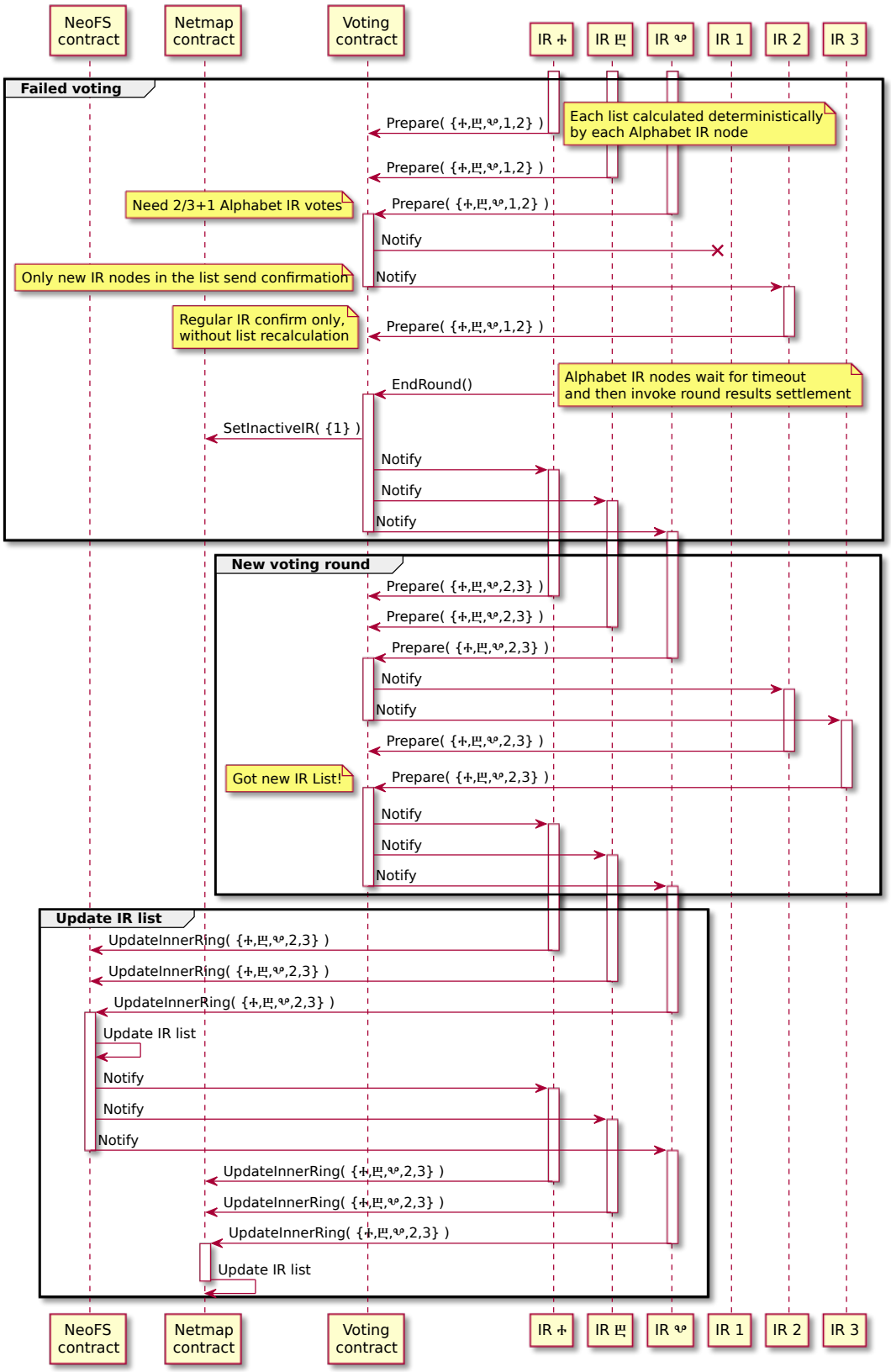

Figure 21: Inner Ring list update in mainnet and sidechain

NeoFS Smart Contracts

alphabet contract

Alphabet contract is a contract deployed in NeoFS sidechain.

Alphabet contract is designed to support GAS production and vote for new validators in the sidechain. NEO token is required to produce GAS and vote for a new committee. It can be distributed among alphabet nodes of the Inner Ring. However, some of them may be malicious, and some NEO can be lost. It will destabilize the economic of the sidechain. To avoid it, all 100,000,000 NEO are distributed among all alphabet contracts.

To identify alphabet contracts, they are named with letters of the Glagolitic alphabet. Names are set at contract deploy. Alphabet nodes of the Inner Ring communicate with one of the alphabetical contracts to emit GAS. To vote for a new list of side chain committee, alphabet nodes of the Inner Ring create multisignature transactions for each alphabet contract.

Contract notifications Alphabet contract does not produce notifications to process.

Contract methods

Emit

```
func Emit()
```

Emit method produces sidechain GAS and distributes it among Inner Ring nodes and proxy contract. It can be invoked only by an Alphabet node of the Inner Ring.

To produce GAS, an alphabet contract transfers all available NEO from the contract account to itself. If notary is enabled, 50% of the GAS in the contract account are transferred to proxy contract. 43.75% of the GAS are equally distributed among all Inner Ring nodes. Remaining 6.25% of the GAS stay in the contract.

If notary is disabled, 87.5% of the GAS are equally distributed among all Inner Ring nodes. Remaining 12.5% of the GAS stay in the contract.

Gas

```
func Gas() int
```

GAS returns the amount of the sidechain GAS stored in the contract account.

Name

```
func Name() string
```

Name returns the Glagolitic name of the contract.

Neo

```
func Neo() int
```

NEO returns the amount of sidechain NEO stored in the contract account.

OnNEP17Payment

```
func OnNEP17Payment(from interop.Hash160, amount int, data interface{})
```

OnNEP17Payment is a callback for NEP-17 compatible native GAS and NEO contracts.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

Version

```
func Version() int
```

Version returns the version of the contract.

Vote

```
func Vote(epoch int, candidates []interop.PublicKey)
```

Vote method votes for the sidechain committee. It requires multisignature from Alphabet nodes of the Inner Ring.

This method is used when governance changes the list of Alphabet nodes of the Inner Ring. Alphabet nodes share keys with sidechain validators, therefore it is required to change them as well. To do that, NEO holders (which are alphabet contracts) should vote for a new committee.

audit contract

Audit contract is a contract deployed in NeoFS sidechain.

Inner Ring nodes perform audit of the registered containers during every epoch. If a container contains StorageGroup objects, an Inner Ring node initializes a series of audit checks. Based on the results of these checks, the Inner Ring node creates a DataAuditResult structure for the container. The content of this structure makes it possible to determine which storage nodes have been examined and see the status of these checks. Regarding this information, the container owner is charged for data storage.

Audit contract is used as a reliable and verifiable storage for all DataAuditResult structures. At the end of data audit routine, Inner Ring nodes send a stable marshaled version of the DataAuditResult structure to the contract. When Alphabet nodes of the Inner Ring perform settlement operations, they make a list and get these AuditResultStructures from the audit contract.

Contract notifications Audit contract does not produce notifications to process.

Contract methods

Get

```
func Get(id []byte) []byte
```

Get method returns a stable marshaled DataAuditResult structure.

The ID of the DataAuditResult can be obtained from listing methods.

List

```
func List() [][]byte
```

List method returns a list of all available DataAuditResult IDs from the contract storage.

ListByCID

```
func ListByCID(epoch int, cid []byte) [][]byte
```

ListByCID method returns a list of DataAuditResult IDs generated during the specified epoch for the specified container.

ListByEpoch

```
func ListByEpoch(epoch int) [][]byte
```

ListByEpoch method returns a list of DataAuditResult IDs generated during the specified epoch.

ListByNode

```
func ListByNode(epoch int, cid []byte, key interop.PublicKey) [][]byte
```

ListByNode method returns a list of DataAuditResult IDs generated in the specified epoch for the specified container by the specified Inner Ring node.

Put

```
func Put(rawAuditResult []byte)
```

Put method stores a stable marshalled ‘DataAuditResult‘ structure. It can be invoked only by Inner Ring nodes.

Inner Ring nodes perform audit of containers and produce ‘DataAuditResult‘ structures. They are stored in audit contract and used for settlements in later epochs.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

Version

```
func Version() int
```

Version returns the version of the contract.

balance contract

Balance contract is a contract deployed in NeoFS sidechain.

Balance contract stores all NeoFS account balances. It is a NEP-17 compatible contract, so it can be tracked and controlled by N3 compatible network monitors and wallet software.

This contract is used to store all micro transactions in the sidechain, such as data audit settlements or container fee payments. It is inefficient to make such small payment transactions in the mainchain.

To process small transfers, balance contract has higher (12) decimal precision than native GAS contract.

NeoFS balances are synchronized with mainchain operations. Deposit produces minting of NEOFS tokens in Balance contract. Withdraw locks some NEOFS tokens in a special lock account. When NeoFS contract transfers GAS assets back to the user, the lock account is destroyed with burn operation.

Contract notifications Transfer notification. This is a NEP-17 standard notification.

```
Transfer:
  - name: from
    type: Hash160
  - name: to
    type: Hash160
  - name: amount
    type: Integer
```

TransferX notification. This is an enhanced transfer notification with details.

```
TransferX:
  - name: from
    type: Hash160
  - name: to
    type: Hash160
  - name: amount
    type: Integer
  - name: details
    type: ByteArray
```

Lock notification. This notification is produced when a lock account is created. It contains information about the mainchain transaction that has produced the asset lock, the address of the lock account and the NeoFS epoch number until which the lock account is valid. Alphabet nodes of the Inner Ring catch notification and initialize Cheque method invocation of NeoFS contract.

```
Lock:
  - name: txID
    type: ByteArray
  - name: from
```

```
    type: Hash160
  - name: to
    type: Hash160
  - name: amount
    type: Integer
  - name: until
    type: Integer
```

Mint notification. This notification is produced when user balance is replenished from deposit in the mainchain.

```
Mint:
 - name: to
   type: Hash160
 - name: amount
   type: Integer
```

Burn notification. This notification is produced after user balance is reduced when NeoFS contract has transferred GAS assets back to the user.

```
Burn:
  - name: from
    type: Hash160
  - name: amount
    type: Integer
```

Contract methods

BalanceOf

```
func BalanceOf(account interop.Hash160) int
```

BalanceOf is a NEP-17 standard method that returns NeoFS balance of the specified account.

Burn

```
func Burn(from interop.Hash160, amount int, txDetails []byte)
```

Burn is a method that transfers assets from a user account to an empty account. It can be invoked only by Alphabet nodes of the Inner Ring.

It produces Burn, Transfer and TransferX notifications.

Burn method is invoked by Alphabet nodes of the Inner Ring when they process Cheque notification from NeoFS contract. It means that locked assets have been transferred to the user in the mainchain, therefore the lock account should be destroyed. Before that, Alphabet nodes should synchronize precision of mainchain GAS contract and Balance contract. Burn decreases total supply of NEP-17 compatible NeoFS token.

Decimals

```
func Decimals() int
```

Decimals is a NEP-17 standard method that returns precision of NeoFS balances.

Lock

```
func Lock(txDetails []byte, from, to interop.Hash160, amount, until int)
```

Lock is a method that transfers assets from a user account to the lock account related to the user. It can be invoked only by Alphabet nodes of the Inner Ring.

It produces Lock, Transfer and TransferX notifications.

Lock method is invoked by Alphabet nodes of the Inner Ring when they process Withdraw notification from NeoFS contract. This should transfer assets to a new lock account that won't be used for anything beside Unlock and Burn.

Mint

```
func Mint(to interop.Hash160, amount int, txDetails []byte)
```

Mint is a method that transfers assets to a user account from an empty account. It can be invoked only by Alphabet nodes of the Inner Ring.

It produces Mint, Transfer and TransferX notifications.

Mint method is invoked by Alphabet nodes of the Inner Ring when they process Deposit notification from NeoFS contract. Before that, Alphabet nodes should synchronize precision of mainchain GAS contract and Balance contract. Mint increases total supply of NEP-17 compatible NeoFS token.

NewEpoch

```
func NewEpoch(epochNum int)
```

NewEpoch is a method that checks timeout on lock accounts and returns assets if lock is not available anymore. It can be invoked only by NewEpoch method of Netmap contract.

It produces Transfer and TransferX notifications.

Symbol

```
func Symbol() string
```

Symbol is a NEP-17 standard method that returns NEOFS token symbol.

TotalSupply

```
func TotalSupply() int
```

TotalSupply is a NEP-17 standard method that returns total amount of main chain GAS in NeoFS network.

Transfer

```
func Transfer(from, to interop.Hash160, amount int, data interface{}) bool
```

Transfer is a NEP-17 standard method that transfers NeoFS balance from one account to another. It can be invoked only by the account owner.

It produces Transfer and TransferX notifications. TransferX notification will have empty details field.

TransferX

```
func TransferX(from, to interop.Hash160, amount int, details []byte)
```

TransferX is a method for NeoFS balance to be transferred from one account to another. It can be invoked by the account owner or by Alphabet nodes.

It produces Transfer and TransferX notifications.

TransferX method expands Transfer method by having extra details argument. TransferX method also allows to transfer assets by Alphabet nodes of the Inner Ring with multisignature.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

Version

```
func Version() int
```

Version returns the version of the contract.

container contract

Container contract is a contract deployed in NeoFS sidechain.

Container contract stores and manages containers, extended ACLs and container size estimations. Contract does not perform sanity or signature checks of containers or extended ACLs, it is done by Alphabet nodes of the Inner Ring. Alphabet nodes approve it by invoking the same Put or SetEACL methods with the same arguments.

Contract notifications containerPut notification. This notification is produced when a user wants to create a new container. Alphabet nodes of the Inner Ring catch the notification and validate container data, signature and token if present.

```
containerPut:
  - name: container
    type: ByteArray
  - name: signature
    type: Signature
  - name: publicKey
    type: PublicKey
  - name: token
    type: ByteArray
```

containerDelete notification. This notification is produced when a container owner wants to delete a container. Alphabet nodes of the Inner Ring catch the notification and validate container ownership, signature and token if present.

```
containerDelete:
  - name: containerID
    type: ByteArray
  - name: signature
    type: Signature
  - name: token
    type: ByteArray
```

setEACL notification. This notification is produced when a container owner wants to update an extended ACL of a container. Alphabet nodes of the Inner Ring catch the notification and validate container ownership, signature and token if present.

```
setEACL:
  - name: eACL
    type: ByteArray
  - name: signature
    type: Signature
  - name: publicKey
    type: PublicKey
  - name: token
    type: ByteArray
```

StartEstimation notification. This notification is produced when Storage nodes should exchange estimation values of container sizes among other Storage nodes.

```
StartEstimation:
  - name: epoch
    type: Integer
```

StopEstimation notification. This notification is produced when Storage nodes should calculate average container size based on received estimations and store it in Container contract.

```
StopEstimation:
  - name: epoch
    type: Integer
```

Contract methods

Count

```
func Count() int
```

Count method returns the number of registered containers.

Delete

```
func Delete(containerID []byte, signature interop.Signature, token []byte)
```

Delete method removes a container from the contract storage if it has been invoked by Alphabet nodes of the Inner Ring. Otherwise, it produces containerDelete notification.

Signature is a RFC6979 signature of the container ID. Token is optional and should be a stable marshaled SessionToken structure from API.

If the container doesn't exist, it panics with NotFoundError.

List

```
func List(owner []byte) [][]byte
```

List method returns a list of all container IDs owned by the specified owner.

ListContainerSizes

```
func ListContainerSizes(epoch int) [][]byte
```

ListContainerSizes method returns the IDs of container size estimations that has been registered for the specified epoch.

NewEpoch

```
func NewEpoch(epochNum int)
```

NewEpoch method removes all container size estimations from epoch older than epochNum + 3. It can be invoked only by NewEpoch method of the Netmap contract.

OnNEP11Payment

```
func OnNEP11Payment(a interop.Hash160, b int, c []byte, d interface{})
```

OnNEP11Payment is needed for registration with contract as the owner to work.

Owner

```
func Owner(containerID []byte) []byte
```

Owner method returns a 25 byte Owner ID of the container.

If the container doesn't exist, it panics with NotFoundError.

Put

```
func Put(container []byte, signature interop.Signature, publicKey
↪  interop.PublicKey, token []byte)
```

Put method creates a new container if it has been invoked by Alphabet nodes of the Inner Ring. Otherwise, it produces containerPut notification.

Container should be a stable marshaled Container structure from API. Signature is a RFC6979 signature of the Container. PublicKey contains the public key of the signer. Token is optional and should be a stable marshaled SessionToken structure from API.

PutContainerSize

```
func PutContainerSize(epoch int, cid []byte, usedSize int, pubKey
↪  interop.PublicKey)
```

PutContainerSize method saves container size estimation in contract memory. It can be invoked only by Storage nodes from the network map. This method checks witness based on the provided public key of the Storage node.

If the container doesn't exist, it panics with NotFoundError.

PutNamed

```
func PutNamed(container []byte, signature interop.Signature, publicKey
↪  interop.PublicKey, token []byte, name, zone string)
```

PutNamed is similar to put but also sets a TXT record in nns contract. Note that zone must exist.

SetEACL

```
func SetEACL(eACL []byte, signature interop.Signature, publicKey
↪  interop.PublicKey, token []byte)
```

SetEACL method sets a new extended ACL table related to the contract if it was invoked by Alphabet nodes of the Inner Ring. Otherwise, it produces setEACL notification.

EACL should be a stable marshaled EACLTable structure from API. Signature is a RFC6979 signature of the Container. PublicKey contains the public key of the signer. Token is optional and should be a stable marshaled SessionToken structure from API.

If the container doesn't exist, it panics with NotFoundError.

StartContainerEstimation

```
func StartContainerEstimation(epoch int)
```

StartContainerEstimation method produces StartEstimation notification. It can be invoked only by Alphabet nodes of the Inner Ring.

StopContainerEstimation

```
func StopContainerEstimation(epoch int)
```

StopContainerEstimation method produces StopEstimation notification. It can be invoked only by Alphabet nodes of the Inner Ring.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked by committee only.

Version

```
func Version() int
```

Version returns the version of the contract.

neofs contract

NeoFS contract is a contract deployed in NeoFS mainchain.

NeoFS contract is an entry point to NeoFS users. This contract stores all NeoFS related GAS, registers new Inner Ring candidates and produces notifications to control the sidechain.

While mainchain committee controls the list of Alphabet nodes in native RoleManagement contract, NeoFS can't change more than 1\3 keys at a time. NeoFS contract contains the actual list of Alphabet nodes in the sidechain.

Network configuration is also stored in NeoFS contract. All changes in configuration are mirrored in the sidechain with notifications.

Contract notifications Deposit notification. This notification is produced when user transfers native GAS to the NeoFS contract address. The same amount of NEOFS token will be minted in Balance contract in the sidechain.

```
Deposit:
  - name: from
    type: Hash160
  - name: amount
    type: Integer
  - name: receiver
    type: Hash160
  - name: txHash
    type: Hash256
```

Withdraw notification. This notification is produced when a user wants to withdraw GAS from the internal NeoFS balance and has paid fee for that.

```
Withdraw:
  - name: user
    type: Hash160
  - name: amount
    type: Integer
  - name: txHash
    type: Hash256
```

Cheque notification. This notification is produced when NeoFS contract has successfully transferred assets back to the user after withdraw.

```
Cheque:
  - name: id
    type: ByteArray
  - name: user
    type: Hash160
  - name: amount
    type: Integer
  - name: lockAccount
    type: ByteArray
```

Bind notification. This notification is produced when a user wants to bind public keys with the user account (OwnerID). Keys argument is an array of ByteArray.

```
Bind:
```

```
  - name: user
    type: ByteArray
  - name: keys
    type: Array
```

Unbind notification. This notification is produced when a user wants to unbind public keys with the user account (OwnerID). Keys argument is an array of ByteArray.

```
Unbind:
  - name: user
    type: ByteArray
  - name: keys
    type: Array
```

AlphabetUpdate notification. This notification is produced when Alphabet nodes have updated their lists in the contract. Alphabet argument is an array of ByteArray. It contains public keys of new alphabet nodes.

```
AlphabetUpdate:
  - name: id
    type: ByteArray
  - name: alphabet
    type: Array
```

SetConfig notification. This notification is produced when Alphabet nodes update NeoFS network configuration value.

```
SetConfig
  - name: id
    type: ByteArray
  - name: key
    type: ByteArray
  - name: value
    type: ByteArray
```

Contract methods

AlphabetAddress

```
func AlphabetAddress() interop.Hash160
```

AlphabetAddress returns 2\3n+1 multisignature address of alphabet nodes. It is used in sidechain notary disabled environment.

AlphabetList

```
func AlphabetList() []common.IRNode
```

AlphabetList returns an array of alphabet node keys. It is used in sidechain notary disabled environment.

AlphabetUpdate

```
func AlphabetUpdate(id []byte, args []interop.PublicKey)
```

AlphabetUpdate updates a list of alphabet nodes with the provided list of public keys. It can be invoked only by alphabet nodes.

This method is used in notary disabled sidechain environment. In this case, the actual alphabet list should be stored in the NeoFS contract.

Bind

```
func Bind(user []byte, keys []interop.PublicKey)
```

Bind method produces notification to bind the specified public keys in NeoFSID contract in the sidechain. It can be invoked only by specified user.

This method produces Bind notification. This method panics if keys are not 33 byte long. User argument must be a valid 20 byte script hash.

Cheque

```
func Cheque(id []byte, user interop.Hash160, amount int, lockAcc []byte)
```

Cheque transfers GAS back to the user from the contract account, if assets were successfully locked in NeoFS balance contract. It can be invoked only by Alphabet nodes.

This method produces Cheque notification to burn assets in sidechain.

Config

```
func Config(key []byte) interface{}
```

Config returns configuration value of NeoFS configuration. If the key does not exists, returns nil.

InnerRingCandidateAdd

```
func InnerRingCandidateAdd(key interop.PublicKey)
```

InnerRingCandidateAdd adds a key to a list of Inner Ring candidates. It can be invoked only by the candidate itself.

This method transfers fee from a candidate to the contract account. Fee value is specified in NeoFS network config with the key InnerRingCandidateFee.

InnerRingCandidateRemove

```
func InnerRingCandidateRemove(key interop.PublicKey)
```

InnerRingCandidateRemove removes a key from a list of Inner Ring candidates. It can be invoked by Alphabet nodes or the candidate itself.

This method does not return fee back to the candidate.

InnerRingCandidates

```
func InnerRingCandidates() []common.IRNode
```

InnerRingCandidates returns an array of structures that contain an Inner Ring candidate node key.

OnNEP17Payment

```
func OnNEP17Payment(from interop.Hash160, amount int, data interface{})
```

OnNEP17Payment is a callback for NEP-17 compatible native GAS contract. It takes no more than 9000.0 GAS. Native GAS has precision 8, and NeoFS balance contract has precision 12. Values bigger than 9000.0 can break JSON limits for integers when precision is converted.

SetConfig

```
func SetConfig(id, key, val []byte)
```

SetConfig key-value pair as a NeoFS runtime configuration value. It can be invoked only by Alphabet nodes.

Unbind

```
func Unbind(user []byte, keys []interop.PublicKey)
```

Unbind method produces notification to unbind the specified public keys in NeoFSID contract in the sidechain. It can be invoked only by the specified user.

This method produces Unbind notification. This method panics if keys are not 33 byte long. User argument must be a valid 20 byte script hash.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by sidechain committee.

Version

```
func Version() int
```

Version returns version of the contract.

Withdraw

```
func Withdraw(user interop.Hash160, amount int)
```

Withdraw initializes gas asset withdraw from NeoFS. It can be invoked only by the specified user.

This method produces Withdraw notification to lock assets in the sidechain and transfers withdraw fee from a user account to each Alphabet node. If notary is enabled in the mainchain, fee is transferred to Processing contract. Fee value is specified in NeoFS network config with the key WithdrawFee.

neofsid contract

NeoFSID contract is a contract deployed in NeoFS sidechain.

NeoFSID contract is used to store connection between an OwnerID and its public keys. OwnerID is a 25-byte N3 wallet address that can be produced from a public key. It is one-way conversion. In simple cases, NeoFS verifies ownership by checking signature and relation between a public key and an OwnerID.

In more complex cases, a user can use public keys unrelated to the OwnerID to maintain secure access to the data. NeoFSID contract stores relation between an OwnerID and arbitrary public keys. Data

owner can bind a public key with its account or unbind it by invoking Bind or Unbind methods of NeoFS contract in the mainchain. After that, Alphabet nodes produce multisigned AddKey and RemoveKey invocations of NeoFSID contract.

Contract notifications NeoFSID contract does not produce notifications to process.

Contract methods

AddKey

```
func AddKey(owner []byte, keys []interop.PublicKey)
```

AddKey binds a list of the provided public keys to the OwnerID. It can be invoked only by Alphabet nodes.

This method panics if the OwnerID is not an ownerSize byte or the public key is not 33 byte long. If the key is already bound, the method ignores it.

Key

```
func Key(owner []byte) [][]byte
```

Key method returns a list of 33-byte public keys bound with the OwnerID.

This method panics if the owner is not ownerSize byte long.

RemoveKey

```
func RemoveKey(owner []byte, keys []interop.PublicKey)
```

RemoveKey unbinds the provided public keys from the OwnerID. It can be invoked only by Alphabet nodes.

This method panics if the OwnerID is not an ownerSize byte or the public key is not 33 byte long. If the key is already unbound, the method ignores it.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

Version

```
func Version() int
```

Version returns the version of the contract.

netmap contract

Netmap contract is a contract deployed in NeoFS sidechain.

Netmap contract stores and manages NeoFS network map, Storage node candidates and epoch number counter. In notary disabled environment, contract also stores a list of Inner Ring node keys.

Contract notifications AddPeer notification. This notification is produced when a Storage node sends a bootstrap request by invoking AddPeer method.

```
AddPeer
  - name: nodeInfo
    type: ByteArray
```

UpdateState notification. This notification is produced when a Storage node wants to change its state (go offline) by invoking UpdateState method. Supported states: (2) -- offline.

```
UpdateState
  - name: state
    type: Integer
  - name: publicKey
    type: PublicKey
```

NewEpoch notification. This notification is produced when a new epoch is applied in the network by invoking NewEpoch method.

```
NewEpoch
  - name: epoch
    type: Integer
```

Contract methods

AddPeer

```
func AddPeer(nodeInfo []byte)
```

AddPeer method adds a new candidate to the next network map if it was invoked by Alphabet node. If it was invoked by a node candidate, it produces AddPeer notification. Otherwise, the method throws panic.

If the candidate already exists, its info is updated. NodeInfo argument contains a stable marshaled version of netmap.NodeInfo structure.

AddPeerIR

```
func AddPeerIR(nodeInfo []byte)
```

AddPeerIR method tries to add a new candidate to the network map. It should only be invoked in notary-enabled environment by the alphabet.

Config

```
func Config(key []byte) interface{}
```

Config returns configuration value of NeoFS configuration. If key does not exists, returns nil.

Epoch

```
func Epoch() int
```

Epoch method returns the current epoch number.

InnerRingList

```
func InnerRingList() []common.IRNode
```

InnerRingList method returns a slice of structures that contains the public key of an Inner Ring node. It should be used in notary disabled environment only.

If notary is enabled, look to NeoFSAlphabet role in native RoleManagement contract of the sidechain.

LastEpochBlock

```
func LastEpochBlock() int
```

LastEpochBlock method returns the block number when the current epoch was applied.

NewEpoch

```
func NewEpoch(epochNum int)
```

NewEpoch method changes the epoch number up to the provided epochNum argument. It can be invoked only by Alphabet nodes. If provided epoch number is less than the current epoch number or equals it, the method throws panic.

When epoch number is updated, the contract sets storage node candidates as the current network map. The contract also invokes NewEpoch method on Balance and Container contracts.

It produces NewEpoch notification.

SetConfig

```
func SetConfig(id, key, val []byte)
```

SetConfig key-value pair as a NeoFS runtime configuration value. It can be invoked only by Alphabet nodes.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

UpdateInnerRing

```
func UpdateInnerRing(keys []interop.PublicKey)
```

UpdateInnerRing method updates a list of Inner Ring node keys. It should be used only in notary disabled environment. It can be invoked only by Alphabet nodes.

If notary is enabled, update NeoFSAlphabet role in native RoleManagement contract of the sidechain. Use notary service to collect multisignature.

UpdateSnapshotCount

```
func UpdateSnapshotCount(count int)
```

UpdateSnapshotCount updates the number of the stored snapshots. If a new number is less than the old one, old snapshots are removed. Otherwise, history is extended with empty snapshots, so ‘Snapshot‘ method can return invalid results for ‘diff = new-old‘ epochs until ‘diff‘ epochs have passed.

UpdateState

```
func UpdateState(state int, publicKey interop.PublicKey)
```

UpdateState method updates the state of a node from the network map candidate list. For notary-ENABLED environment, tx must be signed by both storage node and alphabet. To force update without storage node signature, see 'UpdateStateIR'.

For notary-DISABLED environment, the behaviour depends on who signed the transaction: 1. If it was signed by alphabet, go into voting. 2. If it was signed by a storage node, emit 'UpdateState' notification. 2. Fail in any other case.

The behaviour can be summarized in the following table: | notary \ Signer | Storage node | Alphabet | Both | | ENABLED | FAIL | FAIL | OK | | DISABLED | NOTIFICATION | OK | OK (same as alphabet) | State argument defines node state. The only supported state now is (2) -- offline state. Node is removed from the network map candidate list.

Method panics when invoked with unsupported states.

UpdateStateIR

```
func UpdateStateIR(state nodeState, publicKey interop.PublicKey)
```

UpdateStateIR method tries to change the node state in the network map. Should only be invoked in notary-enabled environment by alphabet.

Version

```
func Version() int
```

Version returns the version of the contract.

processing contract

Processing contract is a contract deployed in NeoFS mainchain.

Processing contract pays for all multisignature transaction executions when notary service is enabled in the mainchain. Notary service prepares multisigned transactions, however they should contain sidechain GAS to be executed. It is inconvenient to ask Alphabet nodes to pay for these transactions: nodes can change over time, some nodes will spend sidechain GAS faster. It leads to economic instability.

Processing contract exists to solve this issue. At the Withdraw invocation of NeoFS contract, a user pays fee directly to this contract. This fee is used to pay for Cheque invocation of NeoFS contract that

returns mainchain GAS back to the user. The address of the Processing contract is used as the first signer in the multisignature transaction. Therefore, NeoVM executes Verify method of the contract and if invocation is verified, Processing contract pays for the execution.

Contract notifications Processing contract does not produce notifications to process.

Contract methods

OnNEP17Payment

```
func OnNEP17Payment(from interop.Hash160, amount int, data interface{})
```

OnNEP17Payment is a callback for NEP-17 compatible native GAS contract.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by the sidechain committee.

Verify

```
func Verify() bool
```

Verify method returns true if transaction contains valid multisignature of Alphabet nodes of the Inner Ring.

Version

```
func Version() int
```

Version returns the version of the contract.

proxy contract

Proxy contract is a contract deployed in NeoFS sidechain.

Proxy contract pays for all multisignature transaction executions when notary service is enabled in the sidechain. Notary service prepares multisigned transactions, however they should contain sidechain GAS to be executed. It is inconvenient to ask Alphabet nodes to pay for these transactions: nodes can change over time, some nodes will spend sidechain GAS faster. It leads to economic instability.

Proxy contract exists to solve this issue. While Alphabet contracts hold all sidechain NEO, proxy contract holds most of the sidechain GAS. Alphabet contracts emit half of the available GAS to the proxy contract. The address of the Proxy contract is used as the first signer in a multisignature transaction. Therefore, NeoVM executes Verify method of the contract; and if invocation is verified, Proxy contract pays for the execution.

Contract notifications Proxy contract does not produce notifications to process.

Contract methods

OnNEP17Payment

```
func OnNEP17Payment(from interop.Hash160, amount int, data interface{})
```

OnNEP17Payment is a callback for NEP-17 compatible native GAS contract.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

Verify

```
func Verify() bool
```

Verify method returns true if transaction contains valid multisignature of Alphabet nodes of the Inner Ring.

Version

```
func Version() int
```

Version returns the version of the contract.

reputation contract

Reputation contract is a contract deployed in NeoFS sidechain.

Inner Ring nodes produce data audit for each container during each epoch. In the end, nodes produce DataAuditResult structure that contains information about audit progress. Reputation contract provides storage for such structures and simple interface to iterate over available DataAuditResults on specified epoch.

During settlement process, Alphabet nodes fetch all DataAuditResult structures from the epoch and execute balance transfers from data owners to Storage and Inner Ring nodes if data audit succeeds.

Contract notifications Reputation contract does not produce notifications to process.

Contract methods

Get

```
func Get(epoch int, peerID []byte) [][]byte
```

Get method returns a list of all stable marshaled DataAuditResult structures produced by the specified Inner Ring node during the specified epoch.

GetByID

```
func GetByID(id []byte) [][]byte
```

GetByID method returns a list of all stable marshaled DataAuditResult with the specified id. Use ListByEpoch method to obtain the id.

ListByEpoch

```
func ListByEpoch(epoch int) [][]byte
```

ListByEpoch returns a list of IDs that may be used to get reputation data with GetByID method.

Put

```
func Put(epoch int, peerID []byte, value []byte)
```

Put method saves DataAuditResult in contract storage. It can be invoked only by Inner Ring nodes. It does not require multisignature invocations.

Epoch is the epoch number when DataAuditResult structure was generated. PeerID contains public keys of the Inner Ring node that has produced DataAuditResult. Value contains a stable marshaled structure of DataAuditResult.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

Version

```
func Version() int
```

Version returns the version of the contract.

subnet contract

Subnet contract is a contract deployed in NeoFS sidechain.

Subnet contract stores and manages NeoFS subnetwork states. It allows registering and deleting subnetworks, limiting access to them, and defining a list of the Storage Nodes that can be included in them.

Contract notifications Put notification. This notification is produced when a new subnetwork is registered by invoking Put method.

```
Put
  - name: id
    type: ByteArray
  - name: ownerKey
    type: PublicKey
  - name: info
    type: ByteArray
```

Delete notification. This notification is produced when some subnetwork is deleted by invoking Delete method.

```
Delete
  - name: id
    type: ByteArray
```

RemoveNode notification. This notification is produced when some node is deleted by invoking RemoveNode method.

```
RemoveNode
  - name: subnetID
    type: ByteArray
  - name: node
    type: PublicKey
```

Contract methods

AddClientAdmin

```
func AddClientAdmin(subnetID []byte, groupID []byte, adminPublicKey
↪  interop.PublicKey)
```

AddClientAdmin adds a new client administrator of the specified group in the specified subnetwork. Must be called by the owner only.

AddNode

```
func AddNode(subnetID []byte, node interop.PublicKey)
```

AddNode adds a node to the specified subnetwork. Must be called by the subnet's owner or the node administrator only.

AddNodeAdmin

```
func AddNodeAdmin(subnetID []byte, adminKey interop.PublicKey)
```

AddNodeAdmin adds a new node administrator to the specified subnetwork.

AddUser

```
func AddUser(subnetID []byte, groupID []byte, userID []byte)
```

AddUser adds user to the specified subnetwork and group. Must be called by the owner or the group's admin only.

Delete

```
func Delete(id []byte)
```

Delete deletes the subnet with the specified id.

Get

```
func Get(id []byte) []byte
```

Get returns info about the subnet with the specified id.

NodeAllowed

```
func NodeAllowed(subnetID []byte, node interop.PublicKey) bool
```

NodeAllowed checks if a node is included in the specified subnet.

Put

```
func Put(id []byte, ownerKey interop.PublicKey, info []byte)
```

Put creates a new subnet with the specified owner and info.

RemoveClientAdmin

```
func RemoveClientAdmin(subnetID []byte, groupID []byte, adminPublicKey
↪  interop.PublicKey)
```

RemoveClientAdmin removes client administrator from the specified group in the specified subnetwork. Must be called by the owner only.

RemoveNode

```
func RemoveNode(subnetID []byte, node interop.PublicKey)
```

RemoveNode removes a node from the specified subnetwork. Must be called by the subnet's owner or the node administrator only.

RemoveNodeAdmin

```
func RemoveNodeAdmin(subnetID []byte, adminKey interop.PublicKey)
```

RemoveNodeAdmin removes node administrator from the specified subnetwork. Must be called by the subnet owner only.

RemoveUser

```
func RemoveUser(subnetID []byte, groupID []byte, userID []byte)
```

RemoveUser removes a user from the specified subnetwork and group. Must be called by the owner or the group's admin only.

Update

```
func Update(script []byte, manifest []byte, data interface{})
```

Update method updates contract source code and manifest. It can be invoked only by committee.

UserAllowed

```
func UserAllowed(subnetID []byte, user []byte) bool
```

UserAllowed returns bool that indicates if a node is included in the specified subnet.

Version

```
func Version() int
```

Version returns the version of the contract.

Balance transfer details encoding

Alphabet nodes of the Inner Ring use `balance.TransferX` method to manage balances of the NeoFS users at deposit (mint), withdraw (burn), audit settlements, etc. `TransferX` method has `details` argument and `transferX` notification contains `details` field. This field contains bytes that encode the reason of data transfer. First byte of the `details` field defines the transfer type and all other bytes provide extra details.

First Byte	Description	Extra data
0x01	**Mint**: deposit processed by NeoFS contract.	32-byte hash of mainchain transaction which invoked `neofs.Deposit` method.
0x02	**Burn**: cheque processed by NeoFS contract.	32-byte hash of mainchain transaction which invoked `neofs.Cheque` method.
0x03	**Lock**: withdraw processed by NeoFS contract.	32-byte hash of mainchain transaction which invoked `neofs.Withdraw` method.
0x04	**Unlock**: withdraw processed by NeoFS contract but cheque didn't process before timeout, so balance returned to the account.	Up to 8 bytes of epoch number when asset lock was removed.
0x10	**ContainerFee**: put processed by Container contract.	32-byte Container ID

First Byte	Description	Extra data
0x40	**AuditSettlement**: payment to Inner Ring node for processed audit.	8 bytes of epoch (LittleEndian) number when settlement happened.
0x41	**BasicIncomeCollection**: transfer assets from data owner accounts to the banking account.	8 bytes of epoch (LittleEndian) number when settlement happened.
0x42	**BasicIncomeDistribution**: transfer assets from banking account to storage node owner accounts.	8 bytes of epoch (LittleEndian) number when settlement happened.

Reputation model

NeoFS reputation system is a subsystem for calculating trust in a node. It is based on a reputation model for assessing trust, which is, in turn, based on the EigenTrust algorithm designed for peer-to-peer reputation management.

The EigenTrust algorithm is built on the concept of transitive trust: if peer `i` trusts any peer `j`, it will also trust the peers that `j` trusts.

The Subject of trust assessment is the one who calculates trust.
The Object of trust assessment is the one whose trust is being calculated.

Configuration

To calculate reputation values using the EigenTrust algorithm, nodes need the values of the variable parameters of the algorithm. These parameters are constant but can be changed if one needs to adapt the algorithm. For synchronization and correct operation of all nodes, the values are moved to the `Netmap` contract and read from it at the beginning of each epoch.

Parameters and their keys in the global configuration of the `Netmap` contract are as follows:

Key	Type	Description
EigenTrustIterations	uint64	A number of iterations required to calculate Global Trust.
EigenTrustAlpha	string	A parameter responsible for the level of influence of the current node reputation used when calculating its trust in other nodes.

Managers

Exactly one manager is selected for each node in each epoch. Every manager is also a child node of another manager. Thus, each network participant plays two roles: a child node for one of the managers, and a manager for one of the child nodes.

Managers are required to assess the trust of a child node considering its reputation among other nodes. Also, managers are direct (and only) participants in the iterative part of the EigenTrust algorithm; in the late stage of the algorithm, they send Global Trust in their child node to the Reputation contract.

Defining a manager for a node

To unambiguously determine the manager for the current node, the following rule is used:

1. For the current epoch, a network map (a list of active participants) is obtained from the `Netmap` of the contract.
2. The resulting array is sorted using HRW hashing, which takes into account the current epoch number.
3. The index of the current node is determined in a uniquely sorted array - `i`.
4. The `i+1`-th node is taken as a manager of the `i`-th node.

Thus, for the (`Netmap`, `Epoch`) pair (one for all) each network member can uniquely define a manager for any network node. In this case, forecasting a manager for node `i` at an epoch `n`, generally speaking, is a nontrivial task because of the variability of the network map. Also, the rule allows to select a manager for a node pseudo-randomly every epoch.

Taken together, all the above makes it difficult for rogue nodes to cluster for profit.

Local Trust

Local Trust trust of one node to another, calculated using *only* statistical information of their peer-to-peer network interactions. The Subject and Object of such a trust are peer-to-peer nodes.

Subject and Object of a trust

Any node is the Subject of evaluation. The Object of trust is any node other than the Subject. In NeoFS, nodes do not evaluate their own performance because the default is “every node trusts itself”.

The evaluation criterion is successful (non error in sense of network interaction) execution of one of the RPC calls:

1. `Put`
2. `Get`
3. `Search`
4. `GetRange`
5. `GetRangeHash`
6. `Head`
7. `Delete`

A Subject of the Local Trust assessment stores an assessment of the experience of communication with the Objects in `LocalStorage` while interacting with them. Local reputation only makes sense

in context with some epoch value, which should be taken into account when storing and transferring values.

Calculating trust

Using all these formulations, terms and statements, we recognize that Local Trust is only applicable in the context of a certain epoch. It means that peer-to-peer interactions of nodes for an epoch n affect *only* Local Trusts calculated in the epoch n. That fact is not emphasized explicitly further.

Assessment of interactions with a node is a binary value, i.e. equals either `0 (false)` or `1 (true)`.

Let $sat(i, j)$ be a number of positive assessments of interactions with node `j` by node `i`.

Let $unsat(i, j)$ be a number of negative assessments of interactions with node `j` by node `i`.

Therefore, the total number of interactions between node `i` and node `j`:

$$all(i, j) := sat(i, j) + unsat(i, j).$$

Then:

$$S_{ij} = \frac{sat(i, j)}{all(i, j)} \in [0, 1]$$

is the averaged assessment of interactions with node `j` by node `i`.

Let us define $C_{i,j}$ as following:

$$C_{i,j} := \begin{cases} \frac{S_{i,j}}{\sum_j S_{i,j}}, \sum_j S_{i,j} \neq 0 \\ \frac{1}{N-1}, otherwise \end{cases}, \in [0, 1],$$

where N is the number of network participants. $C_{i,j}$ according to the chapter 4.5 of the EigenTrust article[22] is normalized trust of node `i` to node `j`. This value is taken as the final Local reputation of node `j` for node `i`.

Transport

At the end of each epoch, all child nodes must announce the accumulated statistics — the set of all Local Trusts for the epoch that has just finished — to their managers.

[22] http://ilpubs.stanford.edu:8090/562/1/2002-56.pdf

The transfer is made by calling manager's `AnnounceLocalTrust` gRPC method[23].

Each node should not only collect and transmit its local Trusts, but also be ready to accept (act as a server) and correctly process similar data from another node, i.e. act as a manager.

Global Trust

Global Trust the result of the EigenTrust algorithm is the trust in the network participant, which has been obtained regarding *all* Local Trusts of *all* nodes.

Subject and Object of a trust

The Subject of the assessment is the entire system. Any node in the network is an Object of trust.

The evaluation criterion is the aggregate of Local Trust of *all* network participants to the current node, but adjusted for similar Local Trust to each participant from the rest of the network.

Calculating trust

Each manager retrieves the Local Trust value of its child at the end of each epoch and stores it in the `DaughterStorage`. The further task of each manager is to calculate the Global Trust of the system in its child node, and also to transfer all the information it has to another managers so that they, in turn, can perform a similar task.

Global Trust is calculated iteratively. While calculating `GlobalTrust`, according to the EigenTrust algorithm, managers should exchange "intermediate" trust values, the so-called `IntermediateTrust`. With the help of these values, managers at each iteration approximate (recalculate the results of the previous iteration) the value of the Global Trust to the common Global Value Limit: $T = [t_0, ..., t_n]$, where t_i is the limiting Global Trust value in node `i`.

Using all these formulations, terms and statements, we recognize that Local Trust is only applicable in the context of a certain epoch. It means that peer-to-peer interactions of nodes for an epoch n affect *only* Global Trusts calculated in the epoch `n`. `IntermediateTrust` values, likewise, only make sense when used in the context of some epoch **and iteration number**. That fact is not emphasized explicitly further.

The iterative formula:

[23] https://github.com/nspcc-dev/neofs-api/blob/master/reputation/service.proto#L18

$$t_j^{k+1} = (1-\alpha)\sum_{j=0}^{n} C_{ij}t_i^k + \alpha t_j^0$$

,

where C_{ij} is Local Trust of node `i` to node `j`; α is a value that determines the influence of the t_i^0 ("blind" trust in node `i`) - on the final result; $C_{i,j}t_i^k$, in the introduced terminology, is `IntermediateTrust` which has been transferred by the manager of node `i` to the manager of node `j`. In order for the algorithm to work synchronously, the number of iterations and α is read by all nodes from the Global Configuration of the `Netmap` contract at the beginning of each epoch.

Actually, blind trust t_i^0 can be different for different nodes. It can take into account how long ago a node joined the network, whether the node belongs to the developers of the network, etc. In the current implementation $t_i^0 = \frac{1}{N}, \forall i$, where N is the number of network participants in the current epoch.

Transport

In order for the manager of node `j` to be able to calculate t_j^{k+1}, which is the intermediate Global Trust (this is *not* `IntermediateTrust`) to its child node, the following is essential:

1. The manager must have all the existing `IntermediateTrusts` - $C_{ij}t_i^k, \forall i \neq j$.
2. The manager must calculate and transfer all `IntermediateTrusts`, according to the information received from its child node, that is, calculate and transfer all existing $C_{ji}t_j^k, \forall i \neq j$.

To do this, the manager must act both as a client and a server. The transfer itself is run by calling the `AnnounceIntermediateResult` gRPC method[24].

The manager should store intermediate `IntermediateTrust` values in `ConsumerStorage` considering both the epoch and the iteration numbers. Each manager must transmit them based on the number of iterations and the number of blocks in the current epoch.

[24] https://github.com/nspcc-dev/neofs-api/blob/master/reputation/service.proto#L22

Incentive model

Economic of NeoFS uses GAS stored in NeoFS contract in the mainchain. It increases with user deposits and decreases after withdrawals. Therefore, assets don't appear out of nowhere and don't go nowhere. All internal accounts and settlements are managed by Balance contract in the sidechain. Every epoch many settlements take place, which we can divide into two groups:

- data storage payments,
- service fees.

Data storage payments

Data storage payments are made once an epoch. Epochs are measured in sidechain blocks and can change their duration against real time. Thus, estimated profit and expense for an hour, week or month may vary depending on epoch duration. Payment operations are not specified in the protocol. Storage node owners should check payment details themselves and delete unpaid data if they want to free some storage space.

Data storage payments are also made in two steps:

- basic income payments,
- data audit payments.

Basic income

Basic income provides asset flow from data owners to storage node owners when data owners do not create storage groups to trigger audit and audit payments. Basic income settlements are calculated per container. Exact payment price is calculated from an average data size estimated for a single node of a container, basic income rate in NeoFS network configuration, and the number of nodes in the container.

Basic income rate is a NeoFS network configuration value managed by Alphabet nodes of the Inner Ring. It is stored as GAS per GiB value. Once an epoch, Storage Nodes calculate the average data size of each container node store. This data is then accumulated inside the container nodes; once done, the aggregated value is stored in the container contract.

When epoch N starts, Inner Ring nodes estimate the data size for every registered container from epoch `N-1`. To do so, they use the formula given below

$$\frac{Size \cdot Rate}{2^{30}}$$

GAS, where $Size$ is the estimated container size, $Rate$ is the basic income rate. The owner of the container will be charged

$$\frac{Size \cdot Rate}{2^{30}} \cdot N$$

GAS, where N is the number of nodes in the container.

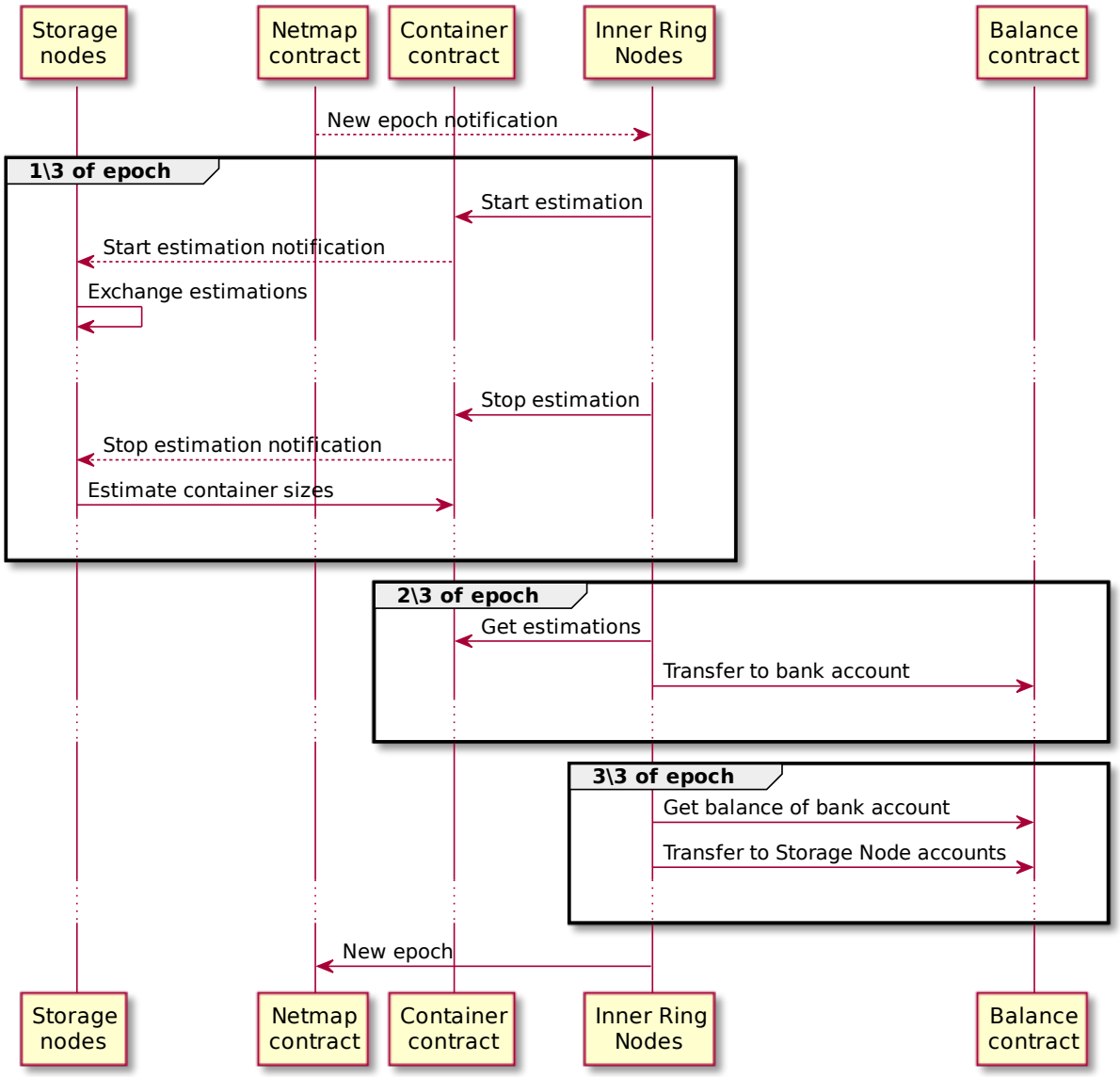

Figure 22: Basic income collection

Data audit

Data audit is triggered if a container contains Storage Group objects. Data audit settlements are also calculated per container. Exact payment price is calculated from Storage Node cost attributes, total size of successfully audited storage groups, and the number of nodes in the container.

Storage groups define the subset of objects inside a container and provides extra meta information for data audit. Objects that are not covered by a storage group are not tested for integrity or safety by Inner Ring nodes. Inner Ring nodes perform data audit permanently. At the start of epoch `N`, Alphabet nodes of the Inner Ring create settlements of all successfully checked Storage Groups from epoch `N-1`. To do so, they use the formula given below

$$\frac{\sum_{i=1}^{k} SGSize_i}{2^{30}} \cdot Price$$

GAS, where $SGSize$ is the total size of objects covered by the storage group (in bytes), $Price$ is a storage node attribute (n GAS per GiB), k is the number of successfully checked Storage Groups in the container. The owner of the container will be charged

$$\sum_{j=1}^{n} \frac{\sum_{i=1}^{k} SGSize_i}{2^{30}} \cdot Price_j$$

GAS, where n is the number of nodes in the container.

Service fees

Container creation fee

To create a container, a data owner should pay fee. It is calculated as $7 \cdot fee$, where fee is a value from NeoFS network configuration (`ContainerFee`). Each Alphabet node gets fee GAS during this operation.

Audit result fee

Each generated audit result must be paid for by the container owner. Data owner pays fee set in NeoFS network configuration (`AuditFee`) per one audited container.

Inner Ring candidate fee

To become a part of the Inner Ring list, an Inner Ring candidate must register its key in NeoFS contract. This operation transfers to NeoFS contract fee value that is set in NeoFS network configuration (`InnerRingCandidateFee`).

Withdraw fee

To withdraw assets, Alphabet nodes need mainchain GAS to invoke Cheque method of NeoFS contract that transfers assets back to the user. This GAS is paid by the user at Withdraw invocation. In notary enabled environment, the user pays fee value set in NeoFS network configuration (`WithdrawFee`). In notary disabled environment, the user pays $7 \cdot fee$.

NeoFS API v2

NeoFS API v2 is focusing on simplification of previous versions of API used during early stages of NeoFS development. The new structure makes it easier to implement NeoFS API in different languages and for different platforms.

All data structures are defined in protobuf format and grouped together with corresponding services into comparatively independent packages. This allows to significantly simplify development (automatically generate for most parts) of a library for working in NeoFS. One can start with a required minimum instead of implementing the whole package right in the beginning. We tried to make the packages as independent as possible, and thus minimizing horizontal dependency.

For transport layer, by default we assume that gRPC will be used. These are popular, simple, and time-tested tools relevant for most languages and platforms. Although gRPC is used now, we have everything to transfer structures through other protocols, e.g. JSON-RPC.

Nodes and their identification

NeoFS API does not differentiate client and server. All members of the network communicate using the same protocol. The protocol does not distinguish between a small Command Line Interface (CLI) utility and a full-featured storage node. Both are nodes of the same p2p network.

A NeoFS Node is identified by a pair of keys for encryption and decryption.

Public key is encoded in compressed form according to ANSI x9.62[25] (section 4.3.6).

Elliptic curve	secp256r1
Private key size	32 bytes
Public key size	33 bytes
Example keys:	
private key	6af2b8b41ad2e78f19aa0bc4fb5cb746d61ad44ebf9ba2a43b6e5cc3e46715a6
public key	03065e513fdaccc4556e7de010bf3d5445552357fb17928f3bd8cea33e092a64eb
Neo 3.0 address	Na6DELLB6dtnPmsD7y1HFjVZNZp8S5BdCJ

The key format is compatible with the Neo 3.0 Wallets keys. It lets smart contracts verify the sender by public keys. Thus, each member may have an internal NEP-5 balance, where NEO Wallet address

[25] https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.202.2977&rep=rep1&type=pdf

is formed from a public key, as it happens in Neo 3.0.

NeoFS is a peer-to-peer network, which means that the clients are equal and each of them needs a pair of asymmetric keys. To create a container and place an object, one should have GAS Utility Token on NeoFS sidechain internal balance. When a user makes a deposit from his NEO Wallet, operations of container creation and object placement should be carried out with the same key (if no other keys have been associated with that OwnerID).

Requests and Responses

All request and responses in NeoFS API v2 have the same structure. Only their `.body` fields are different. Any request consists of body, meta headers, and verification headers. See the corresponding paragraph in the relevant package description section to learn about the structure of particular parts of a message in detail.

The `.body` field delivers the structure with the data making up the request or a response to it.

The `.meta_header` contains metadata to the request.

The `.verify_header` carries cryptographic signatures for `.body` and `.meta_header`. It allows to check if the message is authentic, see if it has been correctly transferred between two nodes, and provide an assumed route of the message where each intermediate node has left its signature.

Signing RPC messages and data structures

The messages exchanged between the users of the network involves ECDSA signatures. These signatures is defined in `refs.Signature` type structure. The `.sign` field keeps byte representation of a signature, while `.key` field contains public key to verify the signature.

Stable serialization

To sign messages or structures, one should first turn them into a byte series. NeoFS protocol is described in protocol buffers v3 format. Protocol Buffers v3 defines the format of serialization for all messages, but specification[26] allows serialization process to be unstable: the same message can be encoded correctly by various methods. The field order in the encoded message may be changed randomly.

[26] https://developers.google.com/protocol-buffers/docs/encoding#order

> When a message is serialized, there is no guaranteed order for how its known or unknown fields should be written. Serialization order is an implementation detail and the details of any particular implementation may change in the future. Therefore, protocol buffer parsers must be able to parse fields in any order.

To generate unified signatures for messages, all NeoFS nodes **must stably serialize structures and messages described in the protocol**.

Serialization is considered stable when it does no change the order of the fields in an encoded message. All fields are encoded in ascending order regarding their numbers specified in protocol description.

```
message Foo {
  bytes one = 1; // C0 FF EE
  bytes two = 2; // BE EF
}

StableSerialize(Foo) =
[0A 03 C0 FF EE][12 02 BE EF]   // OK for GRPC
                                // OK for NeoFS Signature

UnstableSerialize(Foo) =
[12 02 BE EF][0A 03 C0 FF EE]   // OK for GRPC
                                // FAIL for NeoFS Signature
```

Most auto-generated serializers behave occasionally stable, but there is no guarantee that they will remain the same in future. Clients exploiting auto-generated serializers should be aware of such risk.

Signature generation format

The signature of a message or a structure should be stably serialized. The serialized byte array is hashed with SHA-512. The obtained signature (R,S) is enciphered uncompressed according to ANSI x9.62 (section 4.3.6)

Elliptic curve	secp256r1
Hash function	SHA-512
Signature size	65 bytes

message Foo { bytes one = 1; // C0 FF EE bytes two = 2; // BE EF }

```
private key = 6af2b8b41ad2e78f19aa0bc4fb5cb746d61ad44ebf9ba2a43b6e5cc3e46715a6
StableSerialize(Foo) = 0a03coffee1202beaf
SHA-512(StableSerialize(Foo)) =
5efec2432616ca322824a7140d5ac332c6a3a388d2746f8cff6e48909d36829f9cb8586718d457c9

R,S = Sign(private key, SHA-512(StableSerialize(Foo)), secp256r1)
R = e13f3e71db728b85acc4cea688d3dae6b01453d2bff1b5ebc2695cedfef7fdd5
S = 2ecbc0cc0ae4f70696682b4e358a4b698d74f9b708c13470e5c808fe04f526e5

Signature =
04e13f3e71db728b85acc4cea688d3dae6b01453d2bff1b5ebc2695cedfef7fdd52ecbc0cc0ae4f7
```

Signature function uses random number generator, which grants different signatures to the same message with each signing cycle.

Signature chaining in requests and responses

The structure and the order of signatures for `Request` and `Response` messages are the same. Therefore, everything about `Request` message written below is true for `Response` on corresponding structures.

All signatures are kept in `RequestVerificationHeader` structure which is filled before a message is sent. The NeoFS Nodes involved in the transmission and resigning of the messages fill in their own structure copies and put them in origin field recursively.

If a user sends a request directly to a receiver, the chain consists of one `RequestVerification-Header` only.

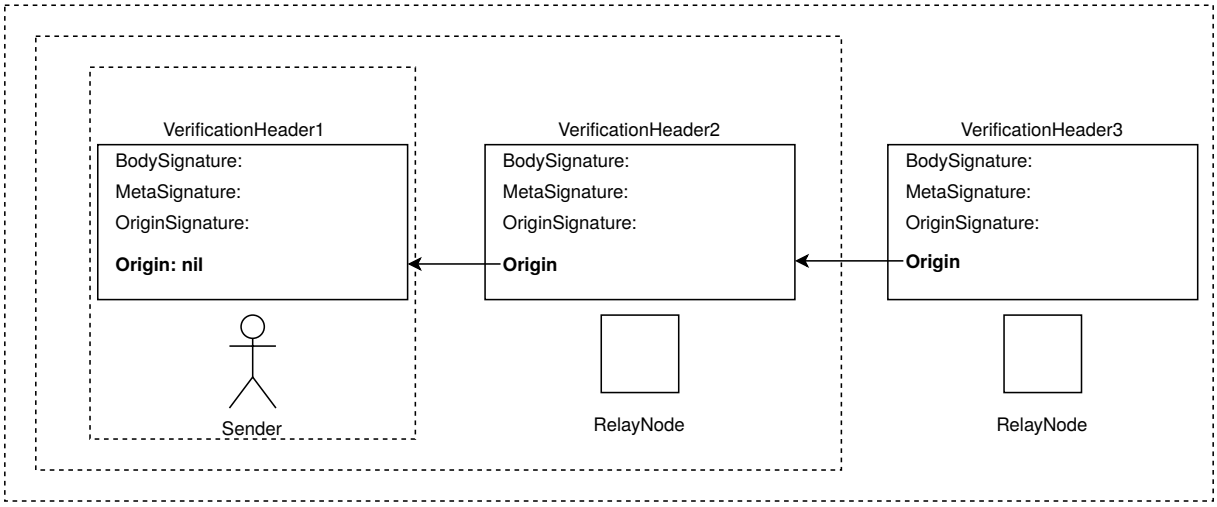

Figure 23: Signature chain verification

The `RequestVerificationHeader` always carries three signatures:

- Message body signature
- Meta header signature
- Verification header signature

Message body signature

Each RPC message has a field called `Body`. This structure is serialized stably; it is signed by the sender only. If a message is retransmitted, this signature is never set in the new copies of `VerificationHeader`.

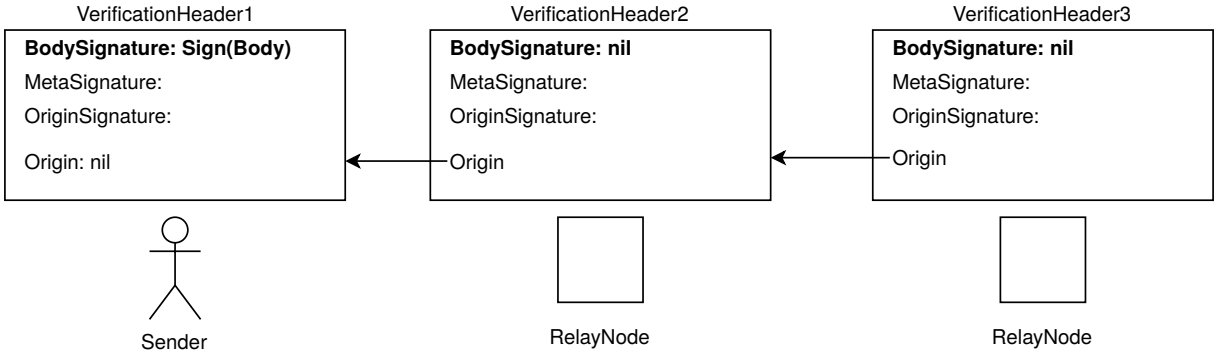

Figure 24: Body signature verification

Meta header signature

Each RPC meassage has a field called meta_header. Meta headers are changed for every retransmission (e.g. TTL is reduced) and form an equivalent chain. meta_signature field contains the signature for an already organized structure of the meta header.

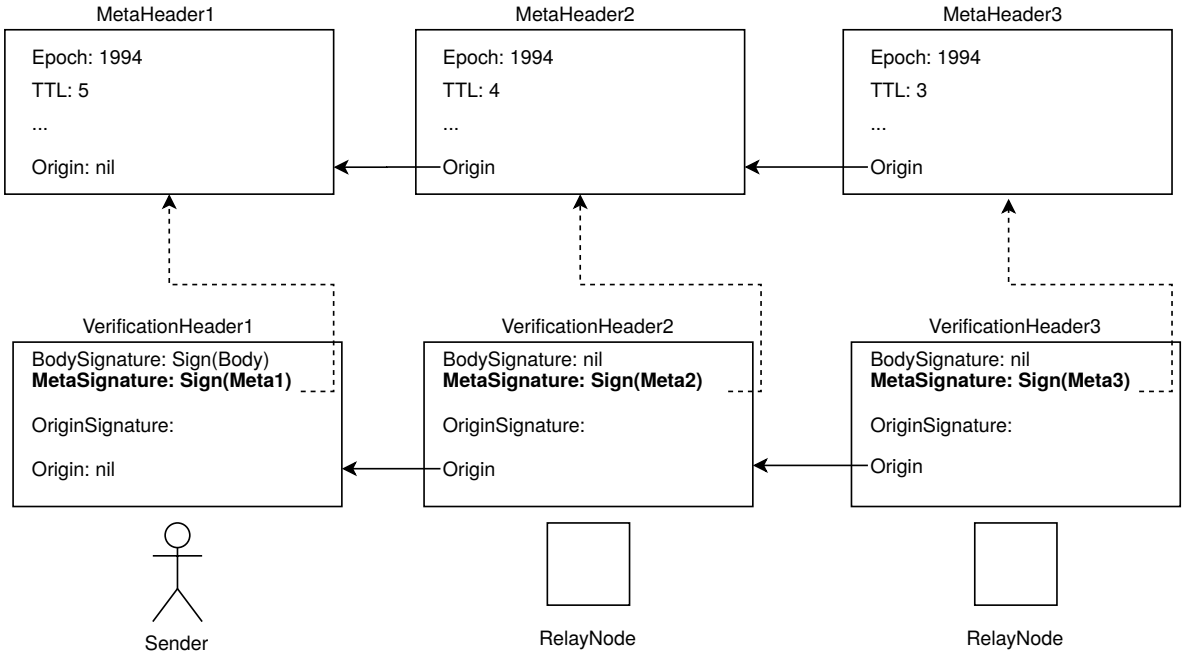

Figure 25: Meta header signature verification

Verification header signature

While putting a new verification header, intermediate nodes should sign the preceding verification header and put the signature in the origin_signature field. The requestor does not set this signature.

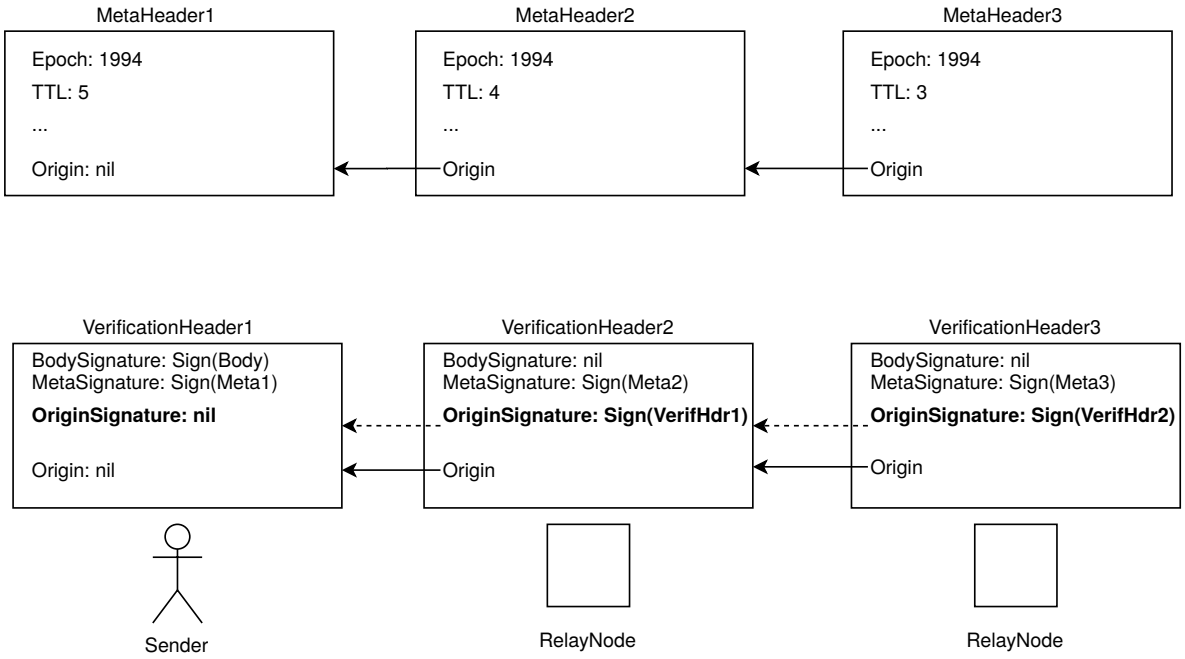

Figure 26: Verification header signature verification

Container service signatures

In addition to the RPC requests themselves there is a need to sign followong structures:

- Container in `container.PutRequest.Body` message
- Container ID in `container.DeleteRequest.Body` message
- Extended ACL table structure in `container.SetExtendedACLRequest.Body` message

Those structures' signature is verified by smart contracts, hence it must be compatible with Neo Virtual Machine. The signature format supported by Neo Virtual Machine is different from the format described in previous sections.

A stably serialized message is hashed using SHA-256 algorithm. The resulting signature (R,S) is encoded uncompressed as a concatenation of the 32-byte sized coordinates of R and S.

Elliptic curve	secp256r1
Hash function	SHA-256
Signature size	64 bytes

```
message ContainerID {
```

```
  bytes value = 1; // 29fe85bb8c36f5cb676e256113193235a2ba0c0abe6a71f84654afa92801c
}

private key = 6af2b8b41ad2e78f19aa0bc4fb5cb746d61ad44ebf9ba2a43b6e5cc3e46715a6
StableSerialize(Foo) = 0a2029fe85bb8c36f5cb676e256113193235a2ba0c0abe6a71f84654af
SHA-256(StableSerialize(Foo)) = a086acbc03862c01bdff3f850b8254f1be9a6d56ec5661c6e

R,S = Sign(private key, SHA-256(StableSerialize(Foo)), secp256r1)
R = 1233d0e5c87a24c5a56c518596da64b1ceb8d667723b0030c4888b524229ff8a
S = d4e42952d516c2959ba1825e2768cbfe3f4336e7a14c635236ae2ea95fa50435

Signature =
1233d0e5c87a24c5a56c518596da64b1ceb8d667723b0030c4888b524229ff8ad4e42952d516c295
```

Object service and Session signatures

Object service and the rest of the services and structures in NeoFS API v2 use the same signature format as in RPC messages signing.

Elliptic curve	secp256r1
Hash function	SHA-512
Signature size	65 bytes

neo.fs.v2.accounting

Service "AccountingService"

Accounting service provides methods for interaction with NeoFS sidechain via other NeoFS nodes to get information about the account balance. Deposit and Withdraw operations can't be implemented here, as they require Mainnet NeoFS smart contract invocation. Transfer operations between internal NeoFS accounts are possible if both use the same token type.

Method Balance

Returns the amount of funds in GAS token for the requested NeoFS account.

Statuses: - **OK** (0, SECTION_SUCCESS): balance has been successfully read; - Common failures (SECTION_FAILURE_COMMON).

Request Body: BalanceRequest.Body

To indicate the account for which the balance is requested, its identifier is used. It can be any existing account in NeoFS sidechain `Balance` smart contract. If omitted, client implementation MUST set it to the request's signer `OwnerID`.

Field	Type	Description
owner_id	OwnerID	Valid user identifier in `OwnerID` format for which the balance is requested. Required field.

Response Body BalanceResponse.Body

The amount of funds in GAS token for the `OwnerID`'s account requested. Balance is given in the `Decimal` format to avoid precision issues with rounding.

Field	Type	Description
balance	Decimal	Amount of funds in GAS token for the requested account.

Message Decimal

Standard floating point data type can't be used in NeoFS due to inexactness of the result when doing lots of small number operations. To solve the lost precision issue, special `Decimal` format is used for monetary computations.

Please see The General Decimal Arithmetic Specification[27] for detailed problem description.

Field	Type	Description
value	int64	Number in the smallest Token fractions.
precision	uint32	Precision value indicating how many smallest fractions can be in one integer.

neo.fs.v2.acl

Message BearerToken

BearerToken allows to attach signed Extended ACL rules to the request in `RequestMetaHeader`. If container's Basic ACL rules allow, the attached rule set will be checked instead of one attached to the container itself. Just like JWT[28], it has a limited lifetime and scope, hence can be used in the similar use cases, like providing authorisation to externally authenticated party.

BearerToken can be issued only by the container's owner and must be signed using the key associated with the container's `OwnerID`.

Field	Type	Description
body	Body	Bearer Token body
signature	Signature	Signature of BearerToken body

Message BearerToken.Body

Bearer Token body structure contains Extended ACL table issued by the container owner with additional information preventing token abuse.

[27] http://speleotrove.com/decimal/
[28] https://jwt.io

Field	Type	Description
eacl_table	EACLTable	Table of Extended ACL rules to use instead of the ones attached to the container. If it contains `container_id` field, bearer token is only valid for this specific container. Otherwise, any container of the same owner is allowed.
owner_id	OwnerID	`OwnerID` defines to whom the token was issued. It must match the request originator's `OwnerID`. If empty, any token bearer will be accepted.
lifetime	TokenLifetime	Token expiration and valid time period parameters

Message BearerToken.Body.TokenLifetime

Lifetime parameters of the token. Field names taken from rfc7519[29].

Field	Type	Description
exp	uint64	Expiration Epoch
nbf	uint64	Not valid before Epoch
iat	uint64	Issued at Epoch

Message EACLRecord

Describes a single eACL rule.

Field	Type	Description
operation	Operation	NeoFS request Verb to match
action	Action	Rule execution result. Either allows or denies access if filters match.
filters	Filter	List of filters to match and see if rule is applicable

[29] https://tools.ietf.org/html/rfc7519

Field	Type	Description
targets	Target	List of target subjects to apply ACL rule to

Message EACLRecord.Filter

Filter to check particular properties of the request or the object.

By default `key` field refers to the corresponding object's `Attribute`. Some Object's header fields can also be accessed by adding `$Object:` prefix to the name. Here is the list of fields available via this prefix:

- $Object:version
  version
- $Object:objectID
  object_id
- $Object:containerID
  container_id
- $Object:ownerID
  owner_id
- $Object:creationEpoch
  creation_epoch
- $Object:payloadLength
  payload_length
- $Object:payloadHash
  payload_hash
- $Object:objectType
  object_type
- $Object:homomorphicHash
  homomorphic_hash

Please note, that if request or response does not have object's headers of full object (Range, RangeHash, Search, Delete), it will not be possible to filter by object header fields or user attributes. From the well-known list only `$Object:objectID` and `$Object:containerID` will be available, as it's possible to take that information from the requested address.

Field	Type	Description
header_type	HeaderType	Define if Object or Request header will be used
match_type	MatchType	Match operation type
key	string	Name of the Header to use
value	string	Expected Header Value or pattern to match

Message EACLRecord.Target

Target to apply ACL rule. Can be a subject's role class or a list of public keys to match.

Field	Type	Description
role	Role	Target subject's role class
keys	bytes	List of public keys to identify target subject

Message EACLTable

Extended ACL rules table. A list of ACL rules defined additionally to Basic ACL. Extended ACL rules can be attached to a container and can be updated or may be defined in `BearerToken` structure. Please see the corresponding NeoFS Technical Specification section for detailed description.

Field	Type	Description
version	Version	eACL format version. Effectively, the version of API library used to create eACL Table.
container_id	ContainerID	Identifier of the container that should use given access control rules
records	EACLRecord	List of Extended ACL rules

Emun Action

Rule execution result action. Either allows or denies access if the rule's filters match.

Number	Name	Description
0	ACTION_UNSPECIFIED	Unspecified action, default value
1	ALLOW	Allow action
2	DENY	Deny action

Emun HeaderType

Enumeration of possible sources of Headers to apply filters.

Number	Name	Description
0	HEADER_UNSPECI	Unspecified header, default value.
1	REQUEST	Filter request headers
2	OBJECT	Filter object headers
3	SERVICE	Filter service headers. These are not processed by NeoFS nodes and exist for service use only.

Emun MatchType

MatchType is an enumeration of match types.

Number	Name	Description
0	MATCH_TYPE_UNSPECIFIED	Unspecified match type, default value.
1	STRING_EQUAL	Return true if strings are equal
2	STRING_NOT_EQUAL	Return true if strings are different

Emun Operation

Request’s operation type to match if the rule is applicable to a particular request.

Number	Name	Description
0	OPERATION_UNSPECIFIED	Unspecified operation, default value
1	GET	Get
2	HEAD	Head
3	PUT	Put
4	DELETE	Delete
5	SEARCH	Search
6	GETRANGE	GetRange
7	GETRANGEHASH	GetRangeHash

Emun Role

Target role of the access control rule in access control list.

Number	Name	Description
0	ROLE_UNSPECIFIE	Unspecified role, default value
1	USER	User target rule is applied if sender is the owner of the container
2	SYSTEM	System target rule is applied if sender is a storage node within the container or an inner ring node
3	OTHERS	Others target rule is applied if sender is neither a user nor a system target

neo.fs.v2.audit

Message DataAuditResult

DataAuditResult keeps record of conducted Data Audits. The detailed report is generated separately.

Field	Type	Description
version	Version	Data Audit Result format version. Effectively, the version of API library used to report DataAuditResult structure.
audit_epoch	fixed64	Epoch number when the Data Audit was conducted
container_id	ContainerID	Container under audit
public_key	bytes	Public key of the auditing InnerRing node in a binary format
complete	bool	Shows if Data Audit process was complete in time or if it was cancelled
requests	uint32	Number of request done at PoR stage
retries	uint32	Number of retries done at PoR stage
pass_sg	ObjectID	List of Storage Groups that passed audit PoR stage
fail_sg	ObjectID	List of Storage Groups that failed audit PoR stage
hit	uint32	Number of sampled objects under the audit placed in an optimal way according to the containers placement policy when checking PoP
miss	uint32	Number of sampled objects under the audit placed in suboptimal way according to the containers placement policy, but still at a satisfactory level when checking PoP
fail	uint32	Number of sampled objects under the audit stored inconsistently with the placement policy or not found at all when checking PoP
pass_nodes	bytes	List of storage node public keys that passed at least one PDP
fail_nodes	bytes	List of storage node public keys that failed at least one PDP

neo.fs.v2.container

Service "ContainerService"

`ContainerService` provides API to interact with `Container` smart contract in NeoFS sidechain via other NeoFS nodes. All of those actions can be done equivalently by directly issuing transactions and RPC calls to sidechain nodes.

Method Put

`Put` invokes `Container` smart contract's `Put` method and returns response immediately. After a new block is issued in sidechain, request is verified by Inner Ring nodes. After one more block in sidechain, the container is added into smart contract storage.

Statuses: - **OK** (0, SECTION_SUCCESS):
request to save the container has been sent to the sidechain; - Common failures (SECTION_FAILURE_COMMON).

Request Body: PutRequest.Body

Container creation request has container structure's signature as a separate field. It's not stored in sidechain, just verified on container creation by `Container` smart contract. `ContainerID` is a SHA256 hash of the stable-marshalled container strucutre, hence there is no need for additional signature checks.

Field	Type	Description
container	Container	Container structure to register in NeoFS
signature	SignatureRFC6979	Signature of a stable-marshalled container according to RFC-6979.

Response Body PutResponse.Body

Container put response body contains information about the newly registered container as seen by `Container` smart contract. `ContainerID` can be calculated beforehand from the container structure and compared to the one returned here to make sure everything has been done as expected.

Field	Type	Description
container_id	ContainerID	Unique identifier of the newly created container

Method Delete

`Delete` invokes `Container` smart contract's `Delete` method and returns response immediately. After a new block is issued in sidechain, request is verified by Inner Ring nodes. After one more block in sidechain, the container is added into smart contract storage.

Statuses: - **OK** (0, SECTION_SUCCESS):
request to remove the container has been sent to the sidechain; - Common failures (SECTION_FAILURE_COMMON).

Request Body: DeleteRequest.Body

Container removal request body has signed `ContainerID` as a proof of the container owner's intent. The signature will be verified by `Container` smart contract, so signing algorithm must be supported by NeoVM.

Field	Type	Description
container_id	ContainerID	Identifier of the container to delete from NeoFS
signature	SignatureRFC6979	`ContainerID` signed with the container owner's key according to RFC-6979.

Response Body DeleteResponse.Body

`DeleteResponse` has an empty body because delete operation is asynchronous and done via consensus in Inner Ring nodes.

Method Get

Returns container structure from `Container` smart contract storage.

Statuses: - **OK** (0, SECTION_SUCCESS):
container has been successfully read; - Common failures (SECTION_FAILURE_COMMON); - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
requested container not found.

Request Body: GetRequest.Body

Get container structure request body.

Field	Type	Description
container_id	ContainerID	Identifier of the container to get

Response Body GetResponse.Body

Get container response body does not have container structure signature. It has been already verified upon container creation.

Field	Type	Description
container	Container	Requested container structure
signature	SignatureRFC6979	Signature of a stable-marshalled container according to RFC-6979.
session_token	SessionToken	Session token if the container has been created within the session

Method List

Returns all owner's containers from 'Container' smart contract' storage.

Statuses: - **OK** (0, SECTION_SUCCESS):
container list has been successfully read; - Common failures (SECTION_FAILURE_COMMON).

Request Body: ListRequest.Body

List containers request body.

Field	Type	Description
owner_id	OwnerID	Identifier of the container owner

Response Body ListResponse.Body

List containers response body.

Field	Type	Description
container_ids	ContainerID	List of `ContainerID`s belonging to the requested `OwnerID`

Method SetExtendedACL

Invokes 'SetEACL' method of 'Container' smart contract and returns response immediately. After one more block in sidechain, changes in an Extended ACL are added into smart contract storage.

Statuses: - **OK** (0, SECTION_SUCCESS):
request to save container eACL has been sent to the sidechain; - Common failures (SECTION_FAILURE_COMMON).

Request Body: SetExtendedACLRequest.Body

Set Extended ACL request body does not have separate `ContainerID` reference. It will be taken from `EACLTable.container_id` field.

Field	Type	Description
eacl	EACLTable	Extended ACL table to set for the container
signature	SignatureRFC6979	Signature of stable-marshalled Extended ACL table according to RFC-6979.

Response Body SetExtendedACLResponse.Body

`SetExtendedACLResponse` has an empty body because the operation is asynchronous and the update should be reflected in `Container` smart contract's storage after next block is issued in sidechain.

Method GetExtendedACL

Returns Extended ACL table and signature from `Container` smart contract storage.

Statuses: - **OK** (0, SECTION_SUCCESS):
container eACL has been successfully read; - Common failures (SECTION_FAILURE_COMMON); - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
container not found; - **EACL_NOT_FOUND** (3073, SECTION_CONTAINER):
eACL table not found.

Request Body: GetExtendedACLRequest.Body

Get Extended ACL request body

Field	Type	Description
container_id	ContainerID	Identifier of the container having Extended ACL

Response Body GetExtendedACLResponse.Body

Get Extended ACL Response body can be empty if the requested container does not have Extended ACL Table attached or Extended ACL has not been allowed at the time of container creation.

Field	Type	Description
eacl	EACLTable	Extended ACL requested, if available
signature	SignatureRFC6979	Signature of stable-marshalled Extended ACL according to RFC-6979.
session_token	SessionToken	Session token if Extended ACL was set within a session

Method AnnounceUsedSpace

Announces the space values used by the container for P2P synchronization.

Statuses: - **OK** (0, SECTION_SUCCESS):
estimation of used space has been successfully announced; - Common failures (SECTION_FAILURE_COMMON).

Request Body: AnnounceUsedSpaceRequest.Body

Container used space announcement body.

Field	Type	Description
announcements	Announcement	List of announcements. If nodes share several containers, announcements are transferred in a batch.

Response Body AnnounceUsedSpaceResponse.Body

`AnnounceUsedSpaceResponse` has an empty body because announcements are one way communication.

Message AnnounceUsedSpaceRequest.Body.Announcement

Announcement contains used space information for a single container.

Field	Type	Description
epoch	uint64	Epoch number for which the container size estimation was produced.
container_id	ContainerID	Identifier of the container.
used_space	uint64	Used space is a sum of object payload sizes of a specified container, stored in the node. It must not include inhumed objects.

Message Container

Container is a structure that defines object placement behaviour. Objects can be stored only within containers. They define placement rule, attributes and access control information. An ID of a container is a 32 byte long SHA256 hash of stable-marshalled container message.

Field	Type	Description
version	Version	Container format version. Effectively, the version of API library used to create the container.
owner_id	OwnerID	Identifier of the container owner
nonce	bytes	Nonce is a 16 byte UUIDv4, used to avoid collisions of `ContainerID`s
basic_acl	uint32	`BasicACL` contains access control rules for the owner, system and others groups, as well as permission bits for `BearerToken` and `Extended ACL`
attributes	Attribute	Attributes represent immutable container's meta data

Field	Type	Description
placement_policy	PlacementPolicy	Placement policy for the object inside the container

Message Container.Attribute

`Attribute` is a user-defined Key-Value metadata pair attached to the container. Container attributes are immutable. They are set at the moment of container creation and can never be added or updated.

Key name must be a container-unique valid UTF-8 string. Value can't be empty. Containers with duplicated attribute names or attributes with empty values will be considered invalid.

There are some "well-known" attributes affecting system behaviour:

- __NEOFS__SUBNET
  String ID of a container's storage subnet. Any container can be attached to one subnet only.
- __NEOFS__NAME
  String of a human-friendly container name registered as a domain in NNS contract.
- __NEOFS__ZONE
  String of a zone for `__NEOFS__NAME`. Used as a TLD of a domain name in NNS contract. If no zone is specified, use default zone: `container`.
- __NEOFS__DISABLE_HOMOMORPHIC_HASHING
  Disables homomorphic hashing for the container if the value equals "true" string. Any other values are interpreted as missing attribute. Container could be accepted in a NeoFS network only if the global network hashing configuration value corresponds with that attribute's value. After container inclusion, network setting is ignored.

And some well-known attributes used by applications only:

- Name
  Human-friendly name
- Timestamp
  User-defined local time of container creation in Unix Timestamp format

Field	Type	Description
key	string	Attribute name key
value	string	Attribute value

neo.fs.v2.lock

Message Lock

Lock objects protects a list of objects from being deleted. The lifetime of a lock object is limited similar to regular objects in `__NEOFS__EXPIRATION_EPOCH` attribute. Lock object MUST have expiration epoch. It is impossible to delete a lock object via ObjectService.Delete RPC call.

Field	Type	Description
members	ObjectID	List of objects to lock. Must not be empty or carry empty IDs. All members must be of the `REGULAR` type.

neo.fs.v2.netmap

Service "NetmapService"

`NetmapService` provides methods to work with `Network Map` and the information required to build it. The resulting `Network Map` is stored in sidechain `Netmap` smart contract, while related information can be obtained from other NeoFS nodes.

Method LocalNodeInfo

Get NodeInfo structure from the particular node directly. Node information can be taken from `Netmap` smart contract. In some cases, though, one may want to get recent information directly or to talk to the node not yet present in the `Network Map` to find out what API version can be used for further communication. This can be also used to check if a node is up and running.

Statuses: - **OK** (0, SECTION_SUCCESS): information about the server has been successfully read; - Common failures (SECTION_FAILURE_COMMON).

Request Body: LocalNodeInfoRequest.Body

LocalNodeInfo request body is empty.

Response Body LocalNodeInfoResponse.Body

Local Node Info, including API Version in use.

Field	Type	Description
version	Version	Latest NeoFS API version in use
node_info	NodeInfo	NodeInfo structure with recent information from node itself

Method NetworkInfo

Read recent information about the NeoFS network.

Statuses: - **OK** (0, SECTION_SUCCESS): information about the current network state has been successfully read; - Common failures (SECTION_FAILURE_COMMON).

Request Body: NetworkInfoRequest.Body

NetworkInfo request body is empty.

Response Body NetworkInfoResponse.Body

Information about the network.

Field	Type	Description
network_info	NetworkInfo	NetworkInfo structure with recent information.

Method NetmapSnapshot

Returns network map snapshot of the current NeoFS epoch.

Statuses: - **OK** (0, SECTION_SUCCESS): information about the current network map has been successfully read; - Common failures (SECTION_FAILURE_COMMON).

Request Body: NetmapSnapshotRequest.Body

Get netmap snapshot request body.

Response Body NetmapSnapshotResponse.Body

Get netmap snapshot response body

Field	Type	Description
netmap	Netmap	Structure of the requested network map.

Message Filter

This filter will return the subset of nodes from `NetworkMap` or another filter's results that will satisfy filter's conditions.

Field	Type	Description
name	string	Name of the filter or a reference to a named filter. '*' means application to the whole unfiltered NetworkMap. At top level it's used as a filter name. At lower levels it's considered to be a reference to another named filter
key	string	Key to filter
op	Operation	Filtering operation
value	string	Value to match
filters	Filter	List of inner filters. Top level operation will be applied to the whole list.

Message Netmap

Network map structure

Field	Type	Description
epoch	uint64	Network map revision number.
nodes	NodeInfo	Nodes presented in network.

Message NetworkConfig

NeoFS network configuration

Field	Type	Description
parameters	Parameter	List of parameter values

Message NetworkConfig.Parameter

Single configuration parameter

Field	Type	Description
key	bytes	Parameter key. UTF-8 encoded string
value	bytes	Parameter value

Message NetworkInfo

Information about NeoFS network

Field	Type	Description
current_epoch	uint64	Number of the current epoch in the NeoFS network
magic_number	uint64	Magic number of the sidechain of the NeoFS network
ms_per_block	int64	MillisecondsPerBlock network parameter of the sidechain of the NeoFS network
network_config	NetworkConfig	NeoFS network configuration

Message NodeInfo

NeoFS node description

Field	Type	Description
public_key	bytes	Public key of the NeoFS node in a binary format
addresses	string	Ways to connect to a node

Field	Type	Description
attributes	Attribute	Carries list of the NeoFS node attributes in a key-value form. Key name must be a node-unique valid UTF-8 string. Value can't be empty. NodeInfo structures with duplicated attribute names or attributes with empty values will be considered invalid.
state	State	Carries state of the NeoFS node

Message NodeInfo.Attribute

Administrator-defined Attributes of the NeoFS Storage Node.

`Attribute` is a Key-Value metadata pair. Key name must be a valid UTF-8 string. Value can't be empty.

Attributes can be constructed into a chain of attributes: any attribute can have a parent attribute and a child attribute (except the first and the last one). A string representation of the chain of attributes in NeoFS Storage Node configuration uses ":" and "/" symbols, e.g.:

```
`NEOFS_NODE_ATTRIBUTE_1=key1:val1/key2:val2`
```

Therefore the string attribute representation in the Node configuration must use ":", "/" and "\" escaped symbols if any of them appears in an attribute's key or value.

Node's attributes are mostly used during Storage Policy evaluation to calculate object's placement and find a set of nodes satisfying policy requirements. There are some "well-known" node attributes common to all the Storage Nodes in the network and used implicitly with default values if not explicitly set:

- Capacity
  Total available disk space in Gigabytes.
- Price
  Price in GAS tokens for storing one GB of data during one Epoch. In node attributes it's a string presenting floating point number with comma or point delimiter for decimal part. In the Network Map it will be saved as 64-bit unsigned integer representing number of minimal token fractions.

- __NEOFS_*SUBNET*%s
  `True` or `False`. Defines if the node is included in the `%s` subnetwork or not. `%s` must be an existing subnetwork's ID (non-negative integer number). A node can be included in more than one subnetwork and, therefore, can contain more than one subnet attribute. A missing attribute is equivalent to the presence of the attribute with `False` value (except default zero subnetwork (with `%s` == 0) for which missing attribute means inclusion in that network).
- UN-LOCODE
  Node's geographic location in UN/LOCODE[30] format approximated to the nearest point defined in the standard.
- CountryCode
  Country code in ISO 3166-1_alpha-2[31] format. Calculated automatically from `UN-LOCODE` attribute.
- Country
  Country short name in English, as defined in ISO-3166[32]. Calculated automatically from `UN-LOCODE` attribute.
- Location
  Place names are given, whenever possible, in their national language versions as expressed in the Roman alphabet using the 26 characters of the character set adopted for international trade data interchange, written without diacritics . Calculated automatically from `UN-LOCODE` attribute.
- SubDivCode
  Country's administrative subdivision where node is located. Calculated automatically from `UN-LOCODE` attribute based on `SubDiv` field. Presented in ISO 3166-2[33] format.
- SubDiv
  Country's administrative subdivision name, as defined in ISO 3166-2[34]. Calculated automatically from `UN-LOCODE` attribute.
- Continent
  Node's continent name according to the [Seven-Continent model] (https://en.wikipedia.org/wiki/Continent#Nu
  Calculated automatically from `UN-LOCODE` attribute.

For detailed description of each well-known attribute please see the corresponding section in NeoFS Technical Specification.

[30] https://www.unece.org/cefact/codesfortrade/codes_index.html
[31] https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2
[32] https://www.iso.org/obp/ui/#search
[33] https://en.wikipedia.org/wiki/ISO_3166-2
[34] https://en.wikipedia.org/wiki/ISO_3166-2

Field	Type	Description
key	string	Key of the node attribute
value	string	Value of the node attribute
parents	string	Parent keys, if any. For example for `City` it could be `Region` and `Country`.

Message PlacementPolicy

Set of rules to select a subset of nodes from `NetworkMap` able to store container's objects. The format is simple enough to transpile from different storage policy definition languages.

Field	Type	Description
replicas	Replica	Rules to set number of object replicas and place each one into a named bucket
container_backup_factor	uint32	Container backup factor controls how deep NeoFS will search for nodes alternatives to include into container's nodes subset
selectors	Selector	Set of Selectors to form the container's nodes subset
filters	Filter	List of named filters to reference in selectors
subnet_id	SubnetID	Subnetwork ID to select nodes from. Zero subnet (default) represents all of the nodes which didn't explicitly opt out of membership.

Message Replica

Number of object replicas in a set of nodes from the defined selector. If no selector set, the root bucket containing all possible nodes will be used by default.

Field	Type	Description
count	uint32	How many object replicas to put

Field	Type	Description
selector	string	Named selector bucket to put replicas

Message Selector

Selector chooses a number of nodes from the bucket taking the nearest nodes to the provided `ContainerID` by hash distance.

Field	Type	Description
name	string	Selector name to reference in object placement section
count	uint32	How many nodes to select from the bucket
clause	Clause	Selector modifier showing how to form a bucket
attribute	string	Bucket attribute to select from
filter	string	Filter reference to select from

Emun Clause

Selector modifier shows how the node set will be formed. By default selector just groups nodes into a bucket by attribute, selecting nodes only by their hash distance.

Number	Name	Description
0	CLAUSE_UNSPECI	No modifier defined. Nodes will be selected from the bucket randomly
1	SAME	SAME will select only nodes having the same value of bucket attribute
2	DISTINCT	DISTINCT will select nodes having different values of bucket attribute

Emun NodeInfo.State

Represents the enumeration of various states of the NeoFS node.

Number	Name	Description
0	UNSPECIFIED	Unknown state
1	ONLINE	Active state in the network
2	OFFLINE	Network unavailable state

Emun Operation

Operations on filters

Number	Name	Description
0	OPERATION_UNSPECIFIED	No Operation defined
1	EQ	Equal
2	NE	Not Equal
3	GT	Greater then
4	GE	Greater or equal
5	LT	Less then
6	LE	Less or equal
7	OR	Logical OR
8	AND	Logical AND

neo.fs.v2.object

Service "ObjectService"

`ObjectService` provides API for manipulating objects. Object operations do not affect the sidechain and are only served by nodes in p2p style.

Method Get

Receive full object structure, including Headers and payload. Response uses gRPC stream. First response message carries the object with the requested address. Chunk messages are parts of the ob-

ject's payload if it is needed. All messages, except the first one, carry payload chunks. The requested object can be restored by concatenation of object message payload and all chunks keeping the receiving order.

Extended headers can change `Get` behaviour: * __NEOFS__NETMAP_EPOCH
Will use the requsted version of Network Map for object placement calculation. * __NEOFS__NETMAP_LOOKUP_DEPTH
Will try older versions (starting from `__NEOFS__NETMAP_EPOCH` if specified or the latest one otherwise) of Network Map to find an object until the depth limit is reached.

Please refer to detailed `XHeader` description.

Statuses: - **OK** (0, SECTION_SUCCESS):
object has been successfully read; - Common failures (SECTION_FAILURE_COMMON); - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
object container not found; - **ACCESS_DENIED** (2048, SECTION_OBJECT):
read access to the object is denied; - **OBJECT_NOT_FOUND** (2049, SECTION_OBJECT):
object not found in container; - **TOKEN_EXPIRED** (4097, SECTION_SESSION):
provided session token has expired; - **OBJECT_ALREADY_REMOVED** (2052, SECTION_OBJECT):
the requested object has been marked as deleted.

Request Body: GetRequest.Body

GET Object request body

Field	Type	Description
address	Address	Address of the requested object
raw	bool	If `raw` flag is set, request will work only with objects that are physically stored on the peer node

Response Body GetResponse.Body

GET Object Response body

Field	Type	Description
init	Init	Initial part of the object stream
chunk	bytes	Chunked object payload
split_info	SplitInfo	Meta information of split hierarchy for object assembly.

Method Put

Put the object into container. Request uses gRPC stream. First message SHOULD be of PutHeader type. `ContainerID` and `OwnerID` of an object SHOULD be set. Session token SHOULD be obtained before PUT operation (see session package). Chunk messages are considered by server as a part of an object payload. All messages, except first one, SHOULD be payload chunks. Chunk messages SHOULD be sent in the direct order of fragmentation.

Extended headers can change `Put` behaviour: * __NEOFS__NETMAP_EPOCH
Will use the requsted version of Network Map for object placement calculation.

Please refer to detailed `XHeader` description.

Statuses: - **OK** (0, SECTION_SUCCESS):
object has been successfully saved in the container; - Common failures (SECTION_FAILURE_COMMON); - **LOCKED** (2050, SECTION_OBJECT):
placement of an object of type TOMBSTONE that includes at least one locked object is prohibited; - **LOCK_NON_REGULAR_OBJECT** (2051, SECTION_OBJECT):
placement of an object of type LOCK that includes at least one object of type other than REGULAR is prohibited; - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
object storage container not found; - **ACCESS_DENIED** (2048, SECTION_OBJECT):
write access to the container is denied; - **TOKEN_NOT_FOUND** (4096, SECTION_SESSION):
(for trusted object preparation) session private key does not exist or has been deleted; - **TOKEN_EXPIRED** (4097, SECTION_SESSION):
provided session token has expired.

Request Body: PutRequest.Body

PUT request body

Field	Type	Description
init	Init	Initial part of the object stream
chunk	bytes	Chunked object payload

Response Body PutResponse.Body

PUT Object response body

Field	Type	Description
object_id	ObjectID	Identifier of the saved object

Method Delete

Delete the object from a container. There is no immediate removal guarantee. Object will be marked for removal and deleted eventually.

Extended headers can change `Delete` behaviour: * __NEOFS__NETMAP_EPOCH
Will use the requsted version of Network Map for object placement calculation.

Please refer to detailed XHeader description.

Statuses: - **OK** (0, SECTION_SUCCESS):
object has been successfully marked to be removed from the container; - Common failures (SECTION_FAILURE_COMMON); - **LOCKED** (2050, SECTION_OBJECT):
deleting a locked object is prohibited; - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
object container not found; - **ACCESS_DENIED** (2048, SECTION_OBJECT):
delete access to the object is denied; - **TOKEN_EXPIRED** (4097, SECTION_SESSION):
provided session token has expired.

Request Body: DeleteRequest.Body

Object DELETE request body

Field	Type	Description
address	Address	Address of the object to be deleted

Response Body DeleteResponse.Body

Object DELETE Response has an empty body.

Field	Type	Description
tombstone	Address	Address of the tombstone created for the deleted object

Method Head

Returns the object Headers without data payload. By default full header is returned. If `main_only` request field is set, the short header with only the very minimal information will be returned instead.

Extended headers can change Head behaviour: * __NEOFS__NETMAP_EPOCH
Will use the requsted version of Network Map for object placement calculation.

Please refer to detailed XHeader description.

Statuses: - **OK** (0, SECTION_SUCCESS):
object header has been successfully read; - Common failures (SECTION_FAILURE_COMMON); - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
object container not found; - **ACCESS_DENIED** (2048, SECTION_OBJECT):
access to operation HEAD of the object is denied; - **OBJECT_NOT_FOUND** (2049, SECTION_OBJECT):
object not found in container; - **TOKEN_EXPIRED** (4097, SECTION_SESSION):
provided session token has expired; - **OBJECT_ALREADY_REMOVED** (2052, SECTION_OBJECT):
the requested object has been marked as deleted.

Request Body: HeadRequest.Body

Object HEAD request body

Field	Type	Description
address	Address	Address of the object with the requested Header
main_only	bool	Return only minimal header subset
raw	bool	If `raw` flag is set, request will work only with objects that are physically stored on the peer node

Response Body HeadResponse.Body

Object HEAD response body

Field	Type	Description
header	HeaderWithSignature	Full object's `Header` with `ObjectID` signature
short_header	ShortHeader	Short object header
split_info	SplitInfo	Meta information of split hierarchy.

Method Search

Search objects in container. Search query allows to match by Object Header's filed values. Please see the corresponding NeoFS Technical Specification section for more details.

Extended headers can change `Search` behaviour: * __NEOFS__NETMAP_EPOCH
Will use the requsted version of Network Map for object placement calculation.

Please refer to detailed XHeader description.

Statuses: - **OK** (0, SECTION_SUCCESS):
objects have been successfully selected; - Common failures (SECTION_FAILURE_COMMON); - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
search container not found; - **ACCESS_DENIED** (2048, SECTION_OBJECT):
access to operation SEARCH of the object is denied; - **TOKEN_EXPIRED** (4097, SECTION_SESSION):
provided session token has expired.

Request Body: SearchRequest.Body

Object Search request body

Field	Type	Description
container_id	ContainerID	Container identifier were to search
version	uint32	Version of the Query Language used
filters	Filter	List of search expressions

Response Body SearchResponse.Body

Object Search response body

Field	Type	Description
id_list	ObjectID	List of `ObjectID`s that match the search query

Method GetRange

Get byte range of data payload. Range is set as an (offset, length) tuple. Like in `Get` method, the response uses gRPC stream. Requested range can be restored by concatenation of all received payload chunks keeping the receiving order.

Extended headers can change `GetRange` behaviour: * __NEOFS__NETMAP_EPOCH
Will use the requsted version of Network Map for object placement calculation. * __NEOFS__NETMAP_LOOKUP_DEPTH
Will try older versions of Network Map to find an object until the depth limit is reached.

Please refer to detailed `XHeader` description.

Statuses: - **OK** (0, SECTION_SUCCESS):
data range of the object payload has been successfully read; - Common failures (SECTION_FAILURE_COMMON);
- **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
object container not found; - **ACCESS_DENIED** (2048, SECTION_OBJECT):
access to operation RANGE of the object is denied; - **OBJECT_NOT_FOUND** (2049, SECTION_OBJECT):
object not found in container; - **TOKEN_EXPIRED** (4097, SECTION_SESSION):
provided session token has expired; - **OBJECT_ALREADY_REMOVED** (2052, SECTION_OBJECT):
the requested object has been marked as deleted. - **OUT_OF_RANGE** (2053, SECTION_OBJECT):
the requested range is out of bounds.

Request Body: GetRangeRequest.Body

Byte range of object's payload request body

Field	Type	Description
address	Address	Address of the object containing the requested payload range
range	Range	Requested payload range
raw	bool	If `raw` flag is set, request will work only with objects that are physically stored on the peer node.

Response Body GetRangeResponse.Body

Get Range response body uses streams to transfer the response. Because object payload considered a byte sequence, there is no need to have some initial preamble message. The requested byte range is sent as a series chunks.

Field	Type	Description
chunk	bytes	Chunked object payload's range.
split_info	SplitInfo	Meta information of split hierarchy.

Method GetRangeHash

Returns homomorphic or regular hash of object's payload range after applying XOR operation with the provided `salt`. Ranges are set of (offset, length) tuples. Hashes order in response corresponds to the ranges order in the request. Note that hash is calculated for XORed data.

Extended headers can change `GetRangeHash` behaviour: * __NEOFS__NETMAP_EPOCH
Will use the requsted version of Network Map for object placement calculation. * __NEOFS__NETMAP_LOOKUP_DEPTH
Will try older versions of Network Map to find an object until the depth limit is reached.

Please refer to detailed `XHeader` description.

Statuses: - **OK** (0, SECTION_SUCCESS):
data range of the object payload has been successfully hashed; - Common failures (SECTION_FAILURE_COMMON); - **CONTAINER_NOT_FOUND** (3072, SECTION_CONTAINER):
object container not found; - **ACCESS_DENIED** (2048, SECTION_OBJECT):
access to operation RANGEHASH of the object is denied; - **OBJECT_NOT_FOUND** (2049, SECTION_OBJECT):
object not found in container; - **OUT_OF_RANGE** (2053, SECTION_OBJECT):
the requested range is out of bounds. - **TOKEN_EXPIRED** (4097, SECTION_SESSION):
provided session token has expired.

Request Body: GetRangeHashRequest.Body

Get hash of object's payload part request body.

Field	Type	Description
address	Address	Address of the object that containing the requested payload range
ranges	Range	List of object's payload ranges to calculate homomorphic hash
salt	bytes	Binary salt to XOR object's payload ranges before hash calculation
type	ChecksumType	Checksum algorithm type

Response Body GetRangeHashResponse.Body

Get hash of object's payload part response body.

Field	Type	Description
type	ChecksumType	Checksum algorithm type
hash_list	bytes	List of range hashes in a binary format

Message GetResponse.Body.Init

Initial part of the `Object` structure stream. Technically it's a set of all `Object` structure's fields except `payload`.

Field	Type	Description
object_id	ObjectID	Object's unique identifier.
signature	Signature	Signed `ObjectID`
header	Header	Object metadata headers

Message HeaderWithSignature

Tuple of a full object header and signature of an `ObjectID`.
Signed `ObjectID` is present to verify full header's authenticity through the following steps:

1. Calculate SHA-256 of the marshalled `Header` structure
2. Check if the resulting hash matches `ObjectID`
3. Check if `ObjectID` signature in `signature` field is correct

Field	Type	Description
header	Header	Full object header
signature	Signature	Signed `ObjectID` to verify full header's authenticity

Message PutRequest.Body.Init

Newly created object structure parameters. If some optional parameters are not set, they will be calculated by a peer node.

Field	Type	Description
object_id	ObjectID	ObjectID if available.
signature	Signature	Object signature if available
header	Header	Object's Header
copies_number	uint32	Number of the object copies to store within the RPC call. By default object is processed according to the container's placement policy.

Message Range

Object payload range.Ranges of zero length SHOULD be considered as invalid.

Field	Type	Description
offset	uint64	Offset of the range from the object payload start
length	uint64	Length in bytes of the object payload range

Message SearchRequest.Body.Filter

Filter structure checks if the object header field or the attribute content matches a value.

If no filters are set, search request will return all objects of the container, including Regular object, Tombstones and Storage Group objects. Most human users expect to get only object they can directly work with. In that case, `$Object:ROOT` filter should be used.

By default `key` field refers to the corresponding object's `Attribute`. Some Object's header fields can also be accessed by adding `$Object:` prefix to the name. Here is the list of fields available via this prefix:

- $Object:version
  version
- $Object:objectID
  object_id
- $Object:containerID
  container_id

- $Object:ownerID
  owner_id
- $Object:creationEpoch
  creation_epoch
- $Object:payloadLength
  payload_length
- $Object:payloadHash
  payload_hash
- $Object:objectType
  object_type
- $Object:homomorphicHash
  homomorphic_hash
- $Object:split.parent
  object_id of parent
- $Object:split.splitID
  16 byte UUIDv4 used to identify the split object hierarchy parts

There are some well-known filter aliases to match objects by certain properties:

- $Object:ROOT
  Returns only REGULAR type objects that are not split or that are the top level root objects in a split hierarchy. This includes objects not present physically, like large objects split into smaller objects without a separate top-level root object. Objects of other types like StorageGroups and Tombstones will not be shown. This filter may be useful for listing objects like `ls` command of some virtual file system. This filter is activated if the `key` exists, disregarding the value and matcher type.
- $Object:PHY
  Returns only objects physically stored in the system. This filter is activated if the `key` exists, disregarding the value and matcher type.

Note: using filters with a key with prefix `$Object:` and match type `NOT_PRESENT`is not recommended since this is not a cross-version approach. Behavior when processing this kind of filters is undefined.

Field	Type	Description
match_type	MatchType	Match type to use
key	string	Attribute or Header fields to match
value	string	Value to match

Message Header

Object Header

Field	Type	Description
version	Version	Object format version. Effectively, the version of API library used to create particular object
container_id	ContainerID	Object's container
owner_id	OwnerID	Object's owner
creation_epoch	uint64	Object creation Epoch
payload_length	uint64	Size of payload in bytes. `0xFFFFFFFFFFFFFFFF` means `payload_length` is unknown.
payload_hash	Checksum	Hash of payload bytes
object_type	ObjectType	Type of the object payload content
homomorphic_hash	Checksum	Homomorphic hash of the object payload
session_token	SessionToken	Session token, if it was used during Object creation. Need it to verify integrity and authenticity out of Request scope.
attributes	Attribute	User-defined object attributes
split	Split	Position of the object in the split hierarchy

Message Header.Attribute

`Attribute` is a user-defined Key-Value metadata pair attached to an object.

Key name must be an object-unique valid UTF-8 string. Value can't be empty. Objects with duplicated attribute names or attributes with empty values will be considered invalid.

There are some "well-known" attributes starting with `__NEOFS__` prefix that affect system behaviour:

- __NEOFS__UPLOAD_ID
  Marks smaller parts of a split bigger object
- __NEOFS__EXPIRATION_EPOCH
  Tells GC to delete object after that epoch

- __NEOFS__TICK_EPOCH
  Decimal number that defines what epoch must produce object notification with UTF-8 object address in a body (`0` value produces notification right after object put)
- __NEOFS__TICK_TOPIC
  UTF-8 string topic ID that is used for object notification

And some well-known attributes used by applications only:

- Name
  Human-friendly name
- FileName
  File name to be associated with the object on saving
- FilePath
  Full path to be associated with the object on saving. Should start with a '/' and use '/' as a delimiting symbol. Trailing '/' should be interpreted as a virtual directory marker. If an object has conflicting FilePath and FileName, FilePath should have higher priority, because it is used to construct the directory tree. FilePath with trailing '/' and non-empty FileName attribute should not be used together.
- Timestamp
  User-defined local time of object creation in Unix Timestamp format
- Content-Type
  MIME Content Type of object's payload

For detailed description of each well-known attribute please see the corresponding section in NeoFS Technical Specification.

Field	Type	Description
key	string	string key to the object attribute
value	string	string value of the object attribute

Message Header.Split

Bigger objects can be split into a chain of smaller objects. Information about inter-dependencies between spawned objects and how to re-construct the original one is in the `Split` headers. Parent and children objects must be within the same container.

Field	Type	Description
parent	ObjectID	Identifier of the origin object. Known only to the minor child.
previous	ObjectID	Identifier of the left split neighbor
parent_signature	Signature	`signature` field of the parent object. Used to reconstruct parent.
parent_header	Header	`header` field of the parent object. Used to reconstruct parent.
children	ObjectID	List of identifiers of the objects generated by splitting current one.
split_id	bytes	16 byte UUIDv4 used to identify the split object hierarchy parts. Must be unique inside container. All objects participating in the split must have the same `split_id` value.

Message Object

Object structure. Object is immutable and content-addressed. It means `ObjectID` will change if the header or the payload changes. It's calculated as a hash of header field which contains hash of the object's payload.

For non-regular object types payload format depends on object type specified in the header.

Field	Type	Description
object_id	ObjectID	Object's unique identifier.
signature	Signature	Signed object_id
header	Header	Object metadata headers
payload	bytes	Payload bytes

Message ShortHeader

Short header fields

Field	Type	Description
version	Version	Object format version. Effectively, the version of API library used to create particular object.
creation_epoch	uint64	Epoch when the object was created
owner_id	OwnerID	Object's owner
object_type	ObjectType	Type of the object payload content
payload_length	uint64	Size of payload in bytes. `0xFFFFFFFFFFFFFFFF` means `payload_length` is unknown
payload_hash	Checksum	Hash of payload bytes
homomorphic_hash	Checksum	Homomorphic hash of the object payload

Message SplitInfo

Meta information of split hierarchy for object assembly. With the last part one can traverse linked list of split hierarchy back to the first part and assemble the original object. With a linking object one can assemble an object right from the object parts.

Field	Type	Description
split_id	bytes	16 byte UUID used to identify the split object hierarchy parts.
last_part	ObjectID	The identifier of the last object in split hierarchy parts. It contains split header with the original object header.
link	ObjectID	The identifier of a linking object for split hierarchy parts. It contains split header with the original object header and a sorted list of object parts.

Emun MatchType

Type of match expression

Number	Name	Description
0	MATCH_TYPE_UNSPECIFIED	Unknown. Not used
1	STRING_EQUAL	Full string match
2	STRING_NOT_EQUAL	Full string mismatch
3	NOT_PRESENT	Lack of key
4	COMMON_PREFIX	String prefix match

Emun ObjectType

Type of the object payload content. Only REGULAR type objects can be split, hence TOMBSTONE, STORAGE_GROUP and LOCK payload is limited by the maximum object size.

String presentation of object type is the same as definition: * REGULAR * TOMBSTONE * STORAGE_GROUP * LOCK

Number	Name	Description
0	REGULAR	Just a normal object
1	TOMBSTONE	Used internally to identify deleted objects
2	STORAGE_GROUP	StorageGroup information
3	LOCK	Object lock

neo.fs.v2.refs

Message Address

Objects in NeoFS are addressed by their ContainerID and ObjectID.

String presentation of `Address` is a concatenation of string encoded `ContainerID` and `ObjectID` delimited by '/' character.

Field	Type	Description
container_id	ContainerID	Container identifier

Field	Type	Description
object_id	ObjectID	Object identifier

Message Checksum

Checksum message. Depending on checksum algorithm type, the string presentation may vary:

- TZ
  Hex encoded string without `0x` prefix
- SHA256
  Hex encoded string without `0x` prefix

Field	Type	Description
type	ChecksumType	Checksum algorithm type
sum	bytes	Checksum itself

Message ContainerID

NeoFS container identifier. Container structures are immutable and content-addressed.

`ContainerID` is a 32 byte long SHA256[35] hash of stable-marshalled container message.

String presentation is a base58[36] encoded string.

JSON value will be data encoded as a string using standard base64 encoding with paddings. Either standard[37] or URL-safe[38] base64 encoding with/without paddings are accepted.

Field	Type	Description
value	bytes	Container identifier in a binary format.

[35] https://csrc.nist.gov/publications/detail/fips/180/4/final
[36] https://tools.ietf.org/html/draft-msporny-base58-02
[37] https://tools.ietf.org/html/rfc4648#section-4
[38] https://tools.ietf.org/html/rfc4648#section-5

Message ObjectID

NeoFS Object unique identifier. Objects are immutable and content-addressed. It means `ObjectID` will change if the `header` or the `payload` changes.

`ObjectID` is a 32 byte long SHA256[39] hash of the object's `header` field, which, in it's turn, contains the hash of the object's payload.

String presentation is a base58[40] encoded string.

JSON value will be data encoded as a string using standard base64 encoding with paddings. Either standard[41] or URL-safe[42] base64 encoding with/without paddings are accepted.

Field	Type	Description
value	bytes	Object identifier in a binary format

Message OwnerID

`OwnerID` is a derivative of a user's main public key. The transformation algorithm is the same as for Neo3 wallet addresses. Neo3 wallet address can be directly used as `OwnerID`.

`OwnerID` is a 25 bytes sequence starting with Neo version prefix byte followed by 20 bytes of ScrptHash and 4 bytes of checksum.

String presentation is a Base58 Check[43] Encoded string.

JSON value will be data encoded as a string using standard base64 encoding with paddings. Either standard[44] or URL-safe[45] base64 encoding with/without paddings are accepted.

Field	Type	Description
value	bytes	Identifier of the container owner in a binary format

[39] https://csrc.nist.gov/publications/detail/fips/180/4/final
[40] https://tools.ietf.org/html/draft-msporny-base58-02
[41] https://tools.ietf.org/html/rfc4648#section-4
[42] https://tools.ietf.org/html/rfc4648#section-5
[43] https://en.bitcoin.it/wiki/Base58Check_encoding
[44] https://tools.ietf.org/html/rfc4648#section-4
[45] https://tools.ietf.org/html/rfc4648#section-5

Message Signature

Signature of something in NeoFS.

Field	Type	Description
key	bytes	Public key used for signing
sign	bytes	Signature
scheme	SignatureScheme	Scheme contains digital signature scheme identifier

Message SignatureRFC6979

RFC 6979 signature.

Field	Type	Description
key	bytes	Public key used for signing
sign	bytes	Deterministic ECDSA with SHA-256 hashing

Message SubnetID

NeoFS subnetwork identifier.

String representation of a value is base-10 integer.

JSON representation is an object containing a single `value` number field.

Field	Type	Description
value	fixed32	4-byte integer subnetwork identifier.

Message Version

API version used by a node.

String presentation is a Semantic Versioning 2.0.0 compatible version string with ‘v’ prefix. i.e. `vX.Y`, where X is the major number, Y is the minor number.

Field	Type	Description
major	uint32	Major API version
minor	uint32	Minor API version

Emun ChecksumType

Checksum algorithm type.

Number	Name	Description
0	CHECKSUM_TYPE_UNSPECIFIED	Unknown. Not used
1	TZ	Tillich-Zemor homomorphic hash function
2	SHA256	SHA-256

Emun SignatureScheme

Signature scheme describes digital signing scheme used for (key, signature) pair.

Number	Name	Description
0	ECDSA_SHA512	ECDSA with SHA-512 hashing (FIPS 186-3)
1	ECDSA_RFC6979_SHA256	Deterministic ECDSA with SHA-256 hashing (RFC 6979)
2	ECDSA_RFC6979_S	Deterministic ECDSA with SHA-256 hashing using WalletConnect API. Here the algorithm is the same, but the message format differs.

neo.fs.v2.reputation

Service “ReputationService”

`ReputationService` provides mechanisms for exchanging trust values with other NeoFS nodes. Nodes rate each other’s reputation based on how good they process requests and set a trust level

based on that rating. The trust information is passed to the next nodes to check and aggregate unless the final result is recorded.

Method AnnounceLocalTrust

Announce local client trust information to any node in NeoFS network.

Statuses: - **OK** (0, SECTION_SUCCESS): local trust has been successfully announced; - Common failures (SECTION_FAILURE_COMMON).

Request Body: AnnounceLocalTrustRequest.Body

Announce node's local trust information.

Field	Type	Description
epoch	uint64	Trust assessment Epoch number
trusts	Trust	List of normalized local trust values to other NeoFS nodes. The value is calculated according to EigenTrust++ algorithm and must be a floating point number in [0;1] range.

Response Body AnnounceLocalTrustResponse.Body

Response to the node's local trust information announcement has an empty body because the trust exchange operation is asynchronous. If Trust information does not pass sanity checks, it is silently ignored.

Method AnnounceIntermediateResult

Announce the intermediate result of the iterative algorithm for calculating the global reputation of the node in NeoFS network.

Statuses: - **OK** (0, SECTION_SUCCESS): intermediate trust estimation has been successfully announced; - Common failures (SECTION_FAILURE_COMMON).

Request Body: AnnounceIntermediateResultRequest.Body

Announce intermediate global trust information.

Field	Type	Description
epoch	uint64	Iteration execution Epoch number
iteration	uint32	Iteration sequence number
trust	PeerToPeerTrust	Current global trust value calculated at the specified iteration

Response Body AnnounceIntermediateResultResponse.Body

Response to the node's intermediate global trust information announcement has an empty body because the trust exchange operation is asynchronous. If Trust information does not pass sanity checks, it is silently ignored.

Message GlobalTrust

Global trust level to NeoFS node.

Field	Type	Description
version	Version	Message format version. Effectively, the version of API library used to create the message.
body	Body	Message body
signature	Signature	Signature of the binary body field by the manager.

Message GlobalTrust.Body

Message body structure.

Field	Type	Description
manager	PeerID	Node manager ID
trust	Trust	Global trust level

Message PeerID

NeoFS unique peer identifier is a 33 byte long compressed public key of the node, the same as the one stored in the network map.

String presentation is a base58[46] encoded string.

JSON value will be data encoded as a string using standard base64 encoding with paddings. Either standard[47] or URL-safe[48] base64 encoding with/without paddings are accepted.

Field	Type	Description
public_key	bytes	Peer node's public key

Message PeerToPeerTrust

Trust level of a peer to a peer.

Field	Type	Description
trusting_peer	PeerID	Identifier of the trusting peer
trust	Trust	Trust level

Message Trust

Trust level to a NeoFS network peer.

Field	Type	Description
peer	PeerID	Identifier of the trusted peer
value	double	Trust level in [0:1] range

[46] https://tools.ietf.org/html/draft-msporny-base58-02
[47] https://tools.ietf.org/html/rfc4648#section-4
[48] https://tools.ietf.org/html/rfc4648#section-5

neo.fs.v2.session

Service “SessionService”

`SessionService` allows to establish a temporary trust relationship between two peer nodes and generate a `SessionToken` as the proof of trust to be attached in requests for further verification. Please see corresponding section of NeoFS Technical Specification for details.

Method Create

Open a new session between two peers.

Statuses: - **OK** (0, SECTION_SUCCESS): session has been successfully opened; - Common failures (SECTION_FAILURE_COMMON).

Request Body: CreateRequest.Body

Session creation request body

Field	Type	Description
owner_id	OwnerID	Session initiating user’s or node’s key derived `OwnerID`
expiration	uint64	Session expiration `Epoch`

Response Body CreateResponse.Body

Session creation response body

Field	Type	Description
id	bytes	Identifier of a newly created session
session_key	bytes	Public key used for session

Message ContainerSessionContext

Context information for Session Tokens related to ContainerService requests.

Field	Type	Description
verb	Verb	Type of request for which the token is issued
wildcard	bool	Spreads the action to all owner containers. If set, container_id field is ignored.
container_id	ContainerID	Particular container to which the action applies. Ignored if wildcard flag is set.

Message ObjectSessionContext

Context information for Session Tokens related to ObjectService requests

Field	Type	Description
verb	Verb	Type of request for which the token is issued
address	Address	Related Object address

Message RequestMetaHeader

Meta information attached to the request. When forwarded between peers, request meta headers are folded in matryoshka style.

Field	Type	Description
version	Version	Peer's API version used
epoch	uint64	Peer's local epoch number. Set to 0 if unknown.
ttl	uint32	Maximum number of intermediate nodes in the request route
x_headers	XHeader	Request X-Headers
session_token	SessionToken	Session token within which the request is sent
bearer_token	BearerToken	`BearerToken` with eACL overrides for the request
origin	RequestMetaHeade	`RequestMetaHeader` of the origin request

Field	Type	Description
magic_number	uint64	NeoFS network magic. Must match the value for the network that the server belongs to.

Message RequestVerificationHeader

Verification info for the request signed by all intermediate nodes.

Field	Type	Description
body_signature	Signature	Request Body signature. Should be generated once by the request initiator.
meta_signature	Signature	Request Meta signature is added and signed by each intermediate node
origin_signature	Signature	Signature of previous hops
origin	RequestVerificationHeader	Chain of previous hops signatures

Message ResponseMetaHeader

Information about the response

Field	Type	Description
version	Version	Peer's API version used
epoch	uint64	Peer's local epoch number
ttl	uint32	Maximum number of intermediate nodes in the request route
x_headers	XHeader	Response X-Headers
origin	ResponseMetaHeader	`ResponseMetaHeader` of the origin request
status	Status	Status return

Message ResponseVerificationHeader

Verification info for the response signed by all intermediate nodes

Field	Type	Description
body_signature	Signature	Response Body signature. Should be generated once by an answering node.
meta_signature	Signature	Response Meta signature is added and signed by each intermediate node
origin_signature	Signature	Signature of previous hops
origin	ResponseVerificationHeader	Chain of previous hops signatures

Message SessionToken

NeoFS Session Token.

Field	Type	Description
body	Body	Session Token contains the proof of trust between peers to be attached in requests for further verification. Please see corresponding section of NeoFS Technical Specification for details.
signature	Signature	Signature of `SessionToken` information

Message SessionToken.Body

Session Token body

Field	Type	Description
id	bytes	Token identifier is a valid UUIDv4 in binary form
owner_id	OwnerID	Identifier of the session initiator
lifetime	TokenLifetime	Lifetime of the session
session_key	bytes	Public key used in session

Field	Type	Description
object	ObjectSessionContext	ObjectService session context
container	ContainerSessionContext	ContainerService session context

Message SessionToken.Body.TokenLifetime

Lifetime parameters of the token. Field names taken from rfc7519.

Field	Type	Description
exp	uint64	Expiration Epoch
nbf	uint64	Not valid before Epoch
iat	uint64	Issued at Epoch

Message XHeader

Extended headers for Request/Response. They may contain any user-defined headers to be interpreted on application level.

Key name must be a unique valid UTF-8 string. Value can't be empty. Requests or Responses with duplicated header names or headers with empty values will be considered invalid.

There are some "well-known" headers starting with `__NEOFS__` prefix that affect system behaviour:

- __NEOFS__NETMAP_EPOCH
  Netmap epoch to use for object placement calculation. The `value` is string encoded `uint64` in decimal presentation. If set to '0' or not set, the current epoch only will be used.
- __NEOFS__NETMAP_LOOKUP_DEPTH
  If object can't be found using current epoch's netmap, this header limits how many past epochs the node can look up through. The `value` is string encoded `uint64` in decimal presentation. If set to '0' or not set, only the current epoch will be used.

Field	Type	Description
key	string	Key of the X-Header

Field	Type	Description
value	string	Value of the X-Header

Emun ContainerSessionContext.Verb

Container request verbs

Number	Name	Description
0	VERB_UNSPECIFIED	Unknown verb
1	PUT	Refers to container.Put RPC call
2	DELETE	Refers to container.Delete RPC call
3	SETEACL	Refers to container.SetExtendedACL RPC call

Emun ObjectSessionContext.Verb

Object request verbs

Number	Name	Description
0	VERB_UNSPECIFIED	Unknown verb
1	PUT	Refers to object.Put RPC call
2	GET	Refers to object.Get RPC call
3	HEAD	Refers to object.Head RPC call
4	SEARCH	Refers to object.Search RPC call
5	DELETE	Refers to object.Delete RPC call
6	RANGE	Refers to object.GetRange RPC call
7	RANGEHASH	Refers to object.GetRangeHash RPC call

neo.fs.v2.status

Message Status

Declares the general format of the status returns of the NeoFS RPC protocol. Status is present in all response messages. Each RPC of NeoFS protocol describes the possible outcomes and details of the operation.

Each status is assigned a one-to-one numeric code. Any unique result of an operation in NeoFS is unambiguously associated with the code value.

Numerical set of codes is split into 1024-element sections. An enumeration is defined for each section. Values can be referred to in the following ways:

- numerical value ranging from 0 to 4,294,967,295 (global code);
- values from enumeration (local code). The formula for the ratio of the local code (`L`) of a defined section (`S`) to the global one (`G`): `G = 1024 * S + L`.

All outcomes are divided into successful and failed, which corresponds to the success or failure of the operation. The definition of success follows the semantics of RPC and the description of its purpose. The server must not attach code that is the opposite of the outcome type.

See the set of return codes in the description for calls.

Each status can carry a developer-facing error message. It should be a human readable text in English. The server should not transmit (and the client should not expect) useful information in the message. Field `details` should make the return more detailed.

Field	Type	Description
code	uint32	The status code
message	string	Developer-facing error message
details	Detail	Data detailing the outcome of the operation. Must be unique by ID.

Message Status.Detail

Return detail. It contains additional information that can be used to analyze the response. Each code defines a set of details that can be attached to a status. Client should not handle details that are not covered by the code.

Field	Type	Description
id	uint32	Detail ID. The identifier is required to determine the binary format of the detail and how to decode it.
value	bytes	Binary status detail. Must follow the format associated with ID. The possibility of missing a value must be explicitly allowed.

Emun CommonFail

Section of failed statuses independent of the operation.

Number	Name	Description
0	INTERNAL	[**1024**] Internal server error, default failure. Not detailed. If the server cannot match failed outcome to the code, it should use this code.
1	WRONG_MAGIC_NUMBER	[**1025**] Wrong magic of the NeoFS network. Details: - [**0**] Magic number of the served NeoFS network (big-endian 64-bit unsigned integer).
2	SIGNATURE_VERIF	[**1026**] Signature verification failure.

Emun Container

Section of statuses for container-related operations.

Number	Name	Description
0	CONTAINER_NOT_FOUND	[**3072**] Container not found.
1	EACL_NOT_FOUND	[**3073**] eACL table not found.

Emun Object

Section of statuses for object-related operations.

Number	Name	Description
0	ACCESS_DENIED	[**2048**] Access denied by ACL. Details: - [**0**] Human-readable description (UTF-8 encoded string).
1	OBJECT_NOT_FOUND	[**2049**] Object not found.
2	LOCKED	[**2050**] Operation rejected by the object lock.
3	LOCK_NON_REGULAR_OBJECT	[**2051**] Locking an object with a non-REGULAR type rejected.
4	OBJECT_ALREADY_REMOVED	[**2052**] Object has been marked deleted.
5	OUT_OF_RANGE	[**2053**] Invalid range has been requested for an object.

Emun Section

Section identifiers.

Number	Name	Description
0	SECTION_SUCCESS	Successful return codes.
1	SECTION_FAILURE_COMMON	Failure codes regardless of the operation.
2	SECTION_OBJECT	Object service-specific errors.
3	SECTION_CONTAINER	Container service-specific errors.
4	SECTION_SESSION	Session service-specific errors.

Emun Session

Section of statuses for session-related operations.

Number	Name	Description
0	TOKEN_NOT_FOUND	[**4096**] Token not found.
1	TOKEN_EXPIRED	[**4097**] Token has expired.

Emun Success

Section of NeoFS successful return codes.

Number	Name	Description
0	OK	[**0**] Default success. Not detailed. If the server cannot match successful outcome to the code, it should use this code.

neo.fs.v2.storagegroup

Message StorageGroup

StorageGroup keeps verification information for Data Audit sessions. Objects that require paid storage guarantees are gathered in `StorageGroups` with additional information used for the proof of storage. `StorageGroup` only contains objects from the same container.

Being an object payload, StorageGroup may have expiration Epoch set with `__NEOFS__EXPIRATION_EPOCH` well-known attribute. When expired, StorageGroup will be ignored by InnerRing nodes during Data Audit cycles and will be deleted by Storage Nodes.

Field	Type	Description
validation_data_size	uint64	Total size of the payloads of objects in the storage group
validation_hash	Checksum	Homomorphic hash from the concatenation of the payloads of the storage group members. The order of concatenation is the same as the order of the members in the `members` field.
expiration_epoch	uint64	DEPRECATED. Last NeoFS epoch number of the storage group lifetime
members	ObjectID	Strictly ordered list of storage group member objects. Members MUST be unique

neo.fs.v2.subnet

Message SubnetInfo

NeoFS subnetwork description

Field	Type	Description
id	SubnetID	Unique subnet identifier. Missing ID is equivalent to zero (default subnetwork) ID.
owner	OwnerID	Identifier of the subnetwork owner

neo.fs.v2.tombstone

Message Tombstone

Tombstone keeps record of deleted objects for a few epochs until they are purged from the NeoFS network.

Field	Type	Description
expiration_epoch	uint64	Last NeoFS epoch number of the tombstone lifetime. It's set by the tombstone creator depending on the current NeoFS network settings. A tombstone object must have the same expiration epoch value in `__NEOFS__EXPIRATION_EPOCH` attribute. Otherwise, the tombstone will be rejected by a storage node.
split_id	bytes	16 byte UUID used to identify the split object hierarchy parts. Must be unique inside a container. All objects participating in the split must have the same `split_id` value.
members	ObjectID	List of objects to be deleted.

Terms and definitions

Object An immutable piece of data with metadata in the form of a set of key-value headers. Object has a globally unique identifier.

Glossary

ACL Access Control List. 12, 21

Alphabet nodes Inner Ring nodes that share key with sidechain consensus nodes and send all NeoFS related multisigned transactions . 48

API Application Programming Interface. 9

CLI Command Line Interface. 96

dApp Decentralized Application. 9

EigenTrust EigenTrust algorithm is a reputation management algorithm[49] for peer-to-peer networks . 32, 33, 87, 161

GAS Utility Token Neo Blockchain's utility token . 12, 47, 49, 51, 52, 97

Global Trust the result of the EigenTrust algorithm is the trust in the network participant, which has been obtained regarding *all* Local Trusts of *all* nodes . 33

HRW HRW stands for Rendezvous hashing[50]. It helps to achieve 3 goals:

1. Select nodes uniformly from the whole netmap. This means that every node has a chance to be included in container nodes set.
2. Select nodes deterministically. Identical (netmap, storage policy) pair results in the same placement set on every storage node.
3. Prioritize nodes providing better conditions.

Nodes having more space, better price or better rating are to be selected with higher probability. Specific weighting algorithm is defined for NeoFS network as a whole and is beyond scope of this document. See NeoFS HRW implementation[51] for details . 88

[49] http://ilpubs.stanford.edu:8090/562/1/2002-56.pdf
[50] https://en.wikipedia.org/wiki/Rendezvous_hashing
[51] https://github.com/nspcc-dev/hrw

Local Trust trust of one node to another, calculated using *only* statistical information of their peer-to-peer network interactions. The Subject and Object of such a trust are peer-to-peer nodes . 33, 161

Multiaddress Multiaddress[52] is a format for encoding addresses from various well-established network protocols. It has two forms:

1. a human-readable version to be used when printing to the user (UTF-8);
2. a binary-packed version to be used in storage, transmissions on the wire, and as a primitive in other formats

. 35

N3 Main Net N3 Main Network Blockchain. See Neo Documentation[53] . 11, 48

NEO Token Token representing a share of ownership in the NEO blockchain . 49, 51

Neo Virtual Machine NeoVM is a lightweight virtual machine for executing Neo smart contracts. As the core component of Neo, NeoVM has Turing completeness and high consistency, which can implement arbitrary execution logic and ensure consistent execution results of any node in distributed network, providing strong support for decentralized applications. See Neo Documentation[54] . 102

NeoFS **N**eoFS **E**xaggerative **O**bject **F**ile **S**torage. Also known as **Neo F**ile **S**torage . 9

NeoFS Node Computer system with at least following properties:

- Running relevant version of NeoFS software
- Has a NeoFS Network-wide unique identifier and a key pair
- Has good enough connectivity with other nodes
- Serves requests using NeoFS API protocol

. 9, 96, 99

RoleManagement contract native N3 contract that manages list of public keys for specific roles such as *StateValidator*, *Oracle*, *NeoFSAlphabet* . 48

validator node In the NEO network, NEO holders can enroll themselves to be validators (consensus node candidates), and then be voted as consensus nodes. The voting status of validators and number of consensus nodes are stored in blockchain . 49

[52] https://multiformats.io/multiaddr/
[53] https://docs.neo.org/v3/docs/en-us/network/testnet.html
[54] https://docs.neo.org/docs/en-us/basic/neovm.html