The SENNO Protocol
V1.0

Table of contents

The Senno Protocol

Senno Protocol was created to solve an ongoing market failure in which user private data is used by advertisers without the user being compensated for the use of his private data. In order to confront this issue, the Senno protocol unifies multiple digital ID's of a user from various platforms under a single idChain which contains all his data from the linked ID's and rewards the user for the use of his personal data by advertisers which request access it. The protocol will disrupt the personal data industry by allowing both users and the companies which they have signed up with to receive a significant portion of the revenues generated from the use of customer data.

For more information regarding market need and solution see **The Senno Protocol Whitepaper**

The protocol consists of two main components – A Control network developed using Neo smart contracts and a distributed data storage which supports both a consortium and a public working modes.

Example flow of creating the Senno ID by the protocol:

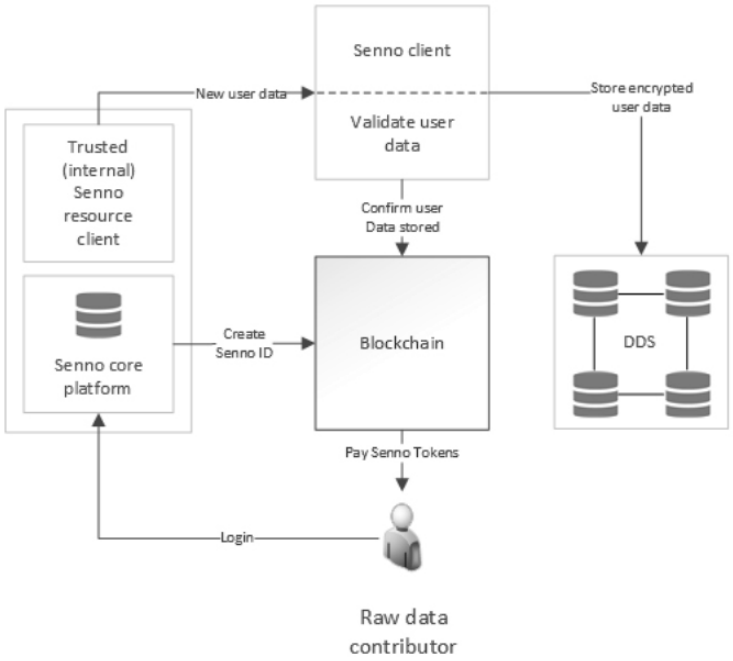

Figure 1.Example flow

System Architecture

Senno architecture was designed to efficiently allow collection, control and distribution of personal data which is stored on a decentralized environment and to oversee the integrity of this data using blockchain smart contracts. The heart of Senno is the Senno Protocol which manages the collection of customer digital ID's ("idChain") and stores personal information on distributed data storage, while rewarding users to their idChain Senno wallet. The on-chain smart contracts of the protocol are executed on the NEO blockchain.

System Layers

The senno architecture consists of four independent layers interacting with one another using blockchain smart contracts to provide a secure distributed ecosystem which stores and efficiently analyzes user data.

Layer Roles

- Control Network – Implemented on the Neo blockchain using smart contracts.
- Business logic layer – Installed on the hardware layer, containing the functions which communicate with the control network.
- Data Storage layer – A Distributed data storage which is based on IPFS / Blockchain.
- Hardware layer – Hardware end systems running the Senno Business logic.

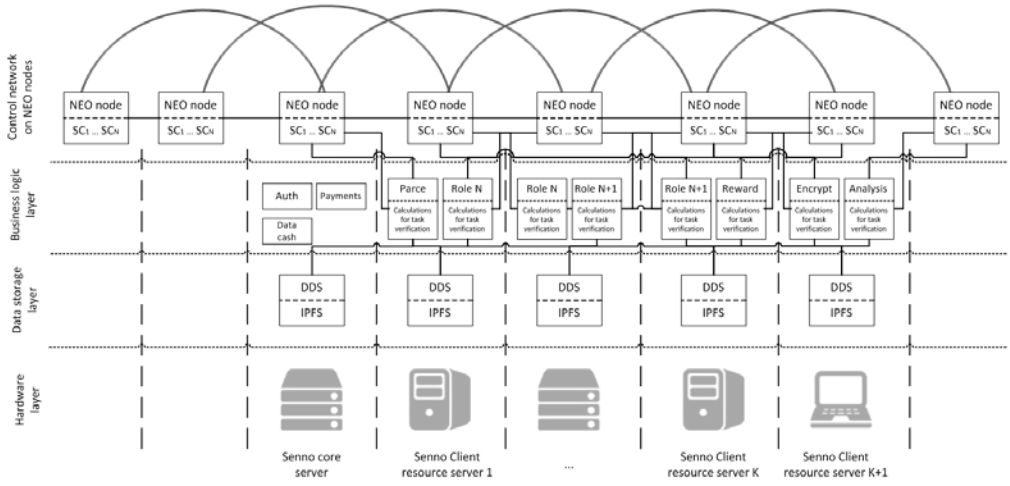

Figure 2.System layers

Data Flow

User data is received in the Hardware layer and transferred by the Senno Core (Business logic layer) to the blockchain control network. There, an array of smart contracts unify the data to a Senno idChain, encrypt it and send it to the Distributed data storage using the Senno clients.

System Architecture Diagram

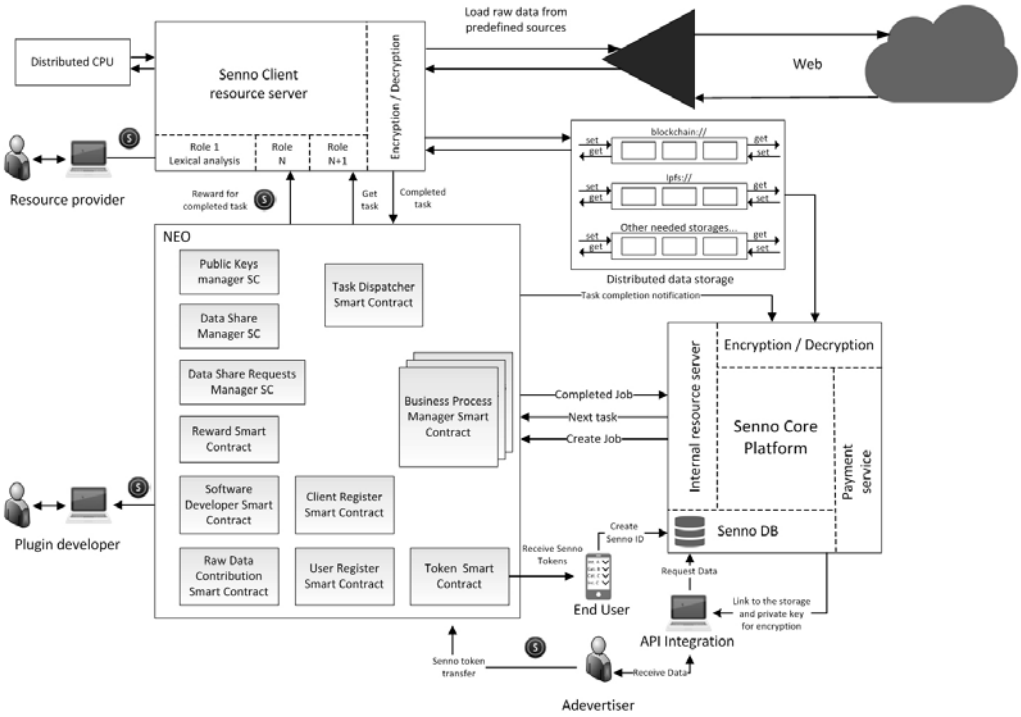

Figure 3. Senno network architecture

The network architecture was designed to support multiple functions while remaining decentralized, low cost and secure. The Senno Core is capable of processing requests from users to create an ID chain or login, from advertisers to pull data and 3rd parties accessing system functions via API. All functions ("Tasks") are managed by the blockchain "Control network" - A set of smart contracts which are in charge of the business logic and rewarding. Tasks are distributed to the Senno Clients ("Nodes") from the control network based on availability and connection speed. The node which received the task from the control network, decrypts it, processes the request and communicates with the distributed data storage ("DDS").

Examples of requests to the DDS:

a. Storing new data (User creating new ID chain / adding userData).
b. Retrieval of existing data (Advertiser pull of data).

Once the task is complete - data is encrypted and stored or data retrieved is from the DDS - the node which performed the task notifies the control network and triggers the Token smart contract, which in turn transfers Senno tokens to the user, node and storage contributor.

Data Flow

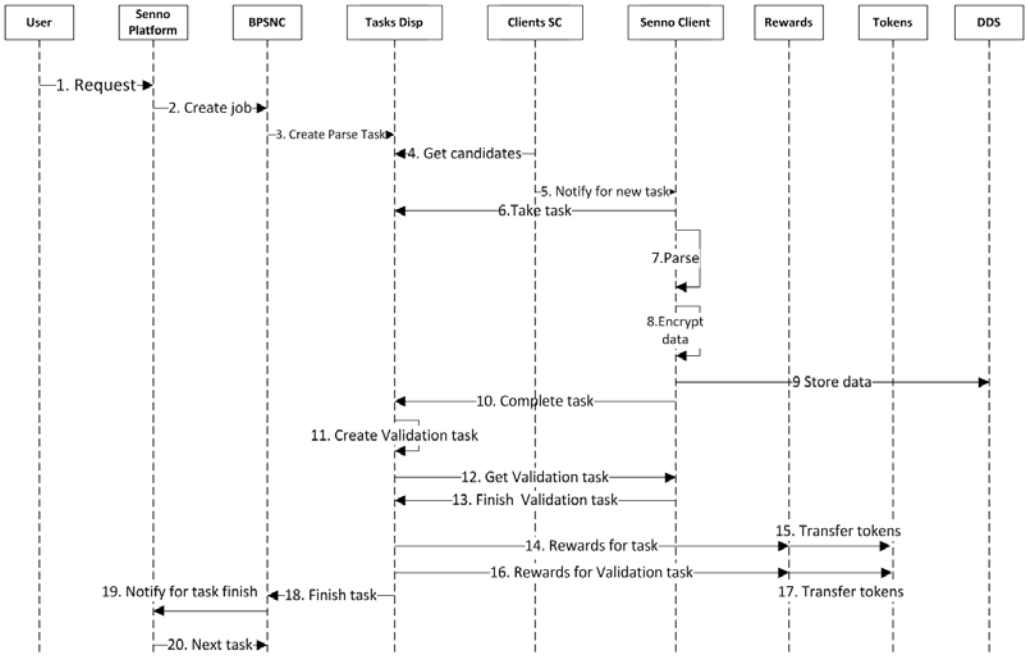

Figure 4.Data flow

Flow description

1. User logs in a client website.
2. Upon successful login, The Senno Core checks if the user has an ID chain, encrypts the data and sends a new job to the Control Network's BPM SC(Job examples: Create a new user ID chain if it does not exist or Retrieve user data if the chain already exists)
3. BPM calls task dispatcher with the user login ID to create the task.
4. If a new ID chain should be created Task dispatcher sends the task to the clients SC.
5. Senno Client gets a notification of a new task.
6. Senno Client confirms execution started.
7. Data is decrypted and parsed in the Senno client.
8. Data is encrypted in the Senno client.
9. Data is stored in the DDS.
10. Task dispatcher gets a notification of completed task starts validation.
11. Validation task is created in the Task dispatcher SC.
12. Senno Client received validation task.
13. Task completion is validated.
14. Rewards SC calculated the reward and call the Token SC for payment to the user and DDS contributor.
15. Tokens are transferred to the user and contributor.
16. Rewards SC calculate the reward for validation task.
17. Tokens are transferred to the Validating Senno client.
18. Task dispatcher SC confirms task completion to the BPM SC.
19. Completion notification is sent back to the core.

Senno Control Network

Senno control network was created to manage various business tasks (data share permissions, public keys management, client & user registration, contribution rewards and business process management). Depending on the settings the network can perform one dedicated role (e.g. permissions update) or several roles simultaneously (e.g. user registration \ confirm reward). The diagram below (figure 5.) shows the flow to the Senno Control network. The Blockchain technology is the basis layer of the architecture which provides distributed computing abilities, immutability and transparency of code execution. Basic architectural concepts of the solution are platform agnostic, NEO smart contracts have been used for the first implementation.

Control Network Flow

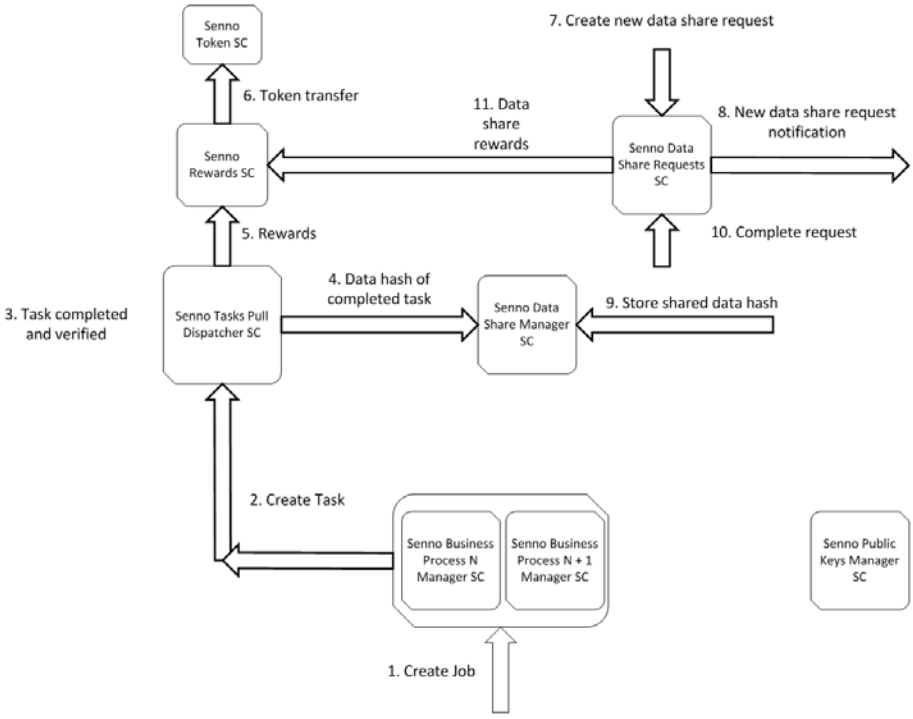

Figure 5.Control network flow

Smart Contracts[1]

Business Process Manager (BPM) Smart Contract –The entry point of the control network which streamlines business processes in the form of tasks from the Senno Core which are packaged inside a Job object. Each business process will have its own specific BPM settings and smart contract.

[1] **view on github**

Tasks Pull Dispatcher Smart Contract (SC) The task dispatcher receives a job from the BPM Smart Contract, accepts completed and verified tasks and transfers hash of completed tasks to the Share Manager.

Rewards SC – A smart contract for calculating tokens reward, based on performed work type and its quantity. Senno Rewards Smart Contract algorithm should define the reward distribution details, for example it could give 90% of reward tokens to the node that performed main task and 10 % to all validates.

Token SC – is a smart contract which stores wallet's state and transfers reward to the participant's wallets according to the Senno Reward SC calculations.

Data Share Requests Manager (DSRM) SC manages and stores data sharing requests and validates state of completed requests.

Data Share Manager (DSM) SC stores data hashes of completed tasks and information on the data that was provided to the consumer, actual data is stored in DDS. The DSM also initiates reward process for data sharing.

Public Keys manager SC is a register of the public keys which are used for the encryption during the storage stage.

Client Register SC - Registration of Senno clients in the system for further tasks distribution.

User Register SC - store user lists, their digital identity info, links to the DDS, registration info and NEO wallets

Software Development SC – Completes a transaction and pays reward to the software developer once data feed plugin has been used

Raw Data Contribution (RDC) SC – Manage access to the data categories for different data consumers.

The Senno Distributed Data Storage

Senno Distributed Data Storage (DDS) was created in order to establish the important values of privacy and freedom of Information; it is managed by a network of data storage nodes which stores the data generated by platform tasks in distributed and fault tolerant way, thus providing immutability and high availability. It's suitable for big volumes of data exchange between the platform nodes.

Additional advantages of DDS:

- Saves bandwidth by collecting content from multiple nodes and pieces instead of from one server
- Supports "offline" content which is relevant in low connectivity areas such as 3rd world or rural areas (same way as git works)
- Censorship resistant

Hybrid storage concept

The DDS was designed to support various protocols of distributed data storage. The two protocols that are being implemented for the first version are: IPFS and MultiChain based dedicated blockchain:

- IPFS allows users to store any amount of data and operate on these data as files. File access can be achieved from any PC connected to the Internet.
- The dedicated blockchain uses specialized streams for storing large amounts of data, as well as a POA Module for secure access (connection) to the network, It is based on the technology of permissioned blockchain.

Senno have used the Hybrid approach with it's the Senno protocol in order to offer its clients a dynamic set of solutions that covers every system requirements scenario.

As a rule of thumb, the option of using IPFS as a data protocol is more appropriate for open systems. In the case where private networks are used, it is better to use a dedicated blockchain solution (e.g. MultiChain), which allows the data to be saved on-chain and provides tailored security features that supports a consortium mode.

How it Works

The procedure for working with DDS in each mode is shown in the figure #6:

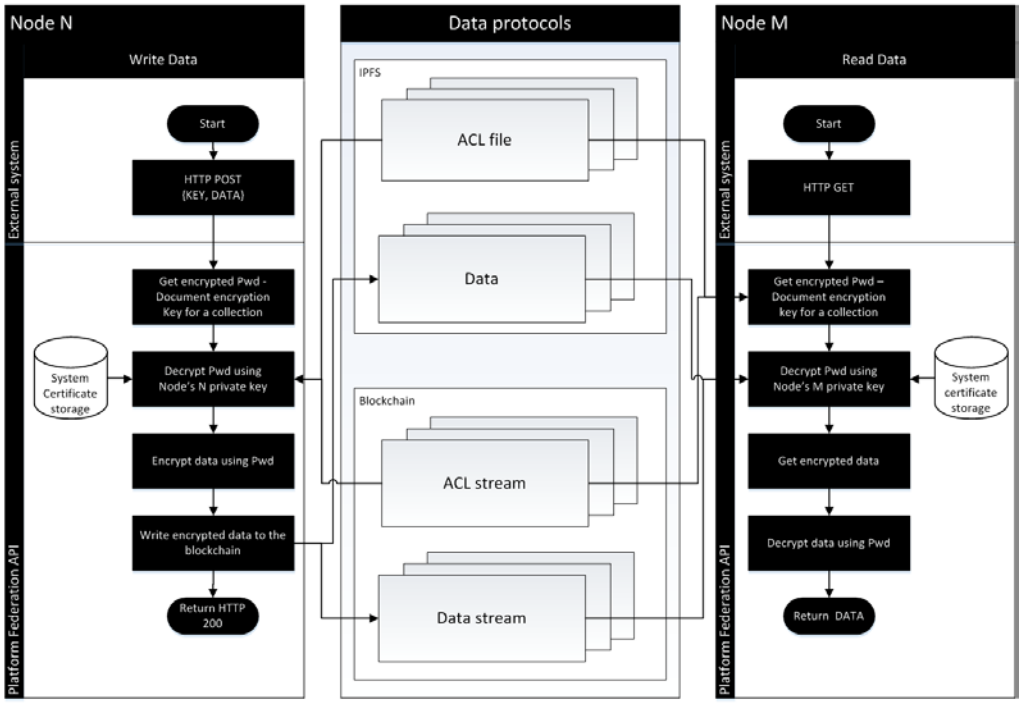

Figure 6. DDS architecture

The data owner saves preliminary encrypted private data version which he will be able to access by himself, access to other users can be granted by the owner if needed.

Common features for both modes:

1. Decentralized system based on the blockchain technology which means data are distributed between all the nodes with no authority controlling the data access.
2. Upon the adding of new participants into the existing userID's access list, the previously published data becomes automatically available for them.
3. Upon the removal of participants from the access list, new published data will not be available to them.
4. The platform is not designed to delete any data, thus no participant can remove a userData from a userID available to other participants. In order to restrict access to userData, the encryption keys can be deleted and accessing the data becomes impossible.
5. With each userData update, the complete history of changes is saved and every userData version is available at any time.

IPFS Mode features:

1. Each file and all of the blocks within it are given a unique fingerprint called a cryptographic hash
2. IPFS removes duplications across the network and tracks version history for every file.
3. Each network node stores only content it is interested in, and additional indexing information that helps figure out who is storing what
4. When looking up files, the network is asked to find nodes storing the content behind a unique hash.

Dedicated Blockchain Mode features:

1. "Permissioned blockchain" (consortium), i.e. in order to connect to it, a permit is required either from the coordinator node or from all the participants (consensus mode).
2. Once connection is established, a consortium participant can create public and private data userIDs. The confidential data in private userIDs are secured by AES encryption algorithms.
3. UserID access management is only available to userID creator.
4. The platform supports the userID data structuring in the JSON format and validates the userID userData format against JSON Schema.
5. The platform provides an API isolating a user from the details of the blockchain implementation and allowing users to work with the entities of a higher level («userData» and «userID»).
6. The platform supports sending customizable notifications to the participant's external systems. These notifications include information on data changes/updates which enables one participant's information systems to react to the data updates made by another participant's systems.

7. The blockchain platform underlying the solution is open-source software which guarantees absence of malicious logic and allows every participant to make sure this solution is totally safe.

DDS access

The system is designed in such a way that it allows using all the advantages of blockchain technologies (reliability, transparency, immutability, distributed storage, no single entity can control data access), with additional feature which supports sharing data in a secure way and control versions history.

Writing data

Requirements:

- Content
- Sender private key
- Receiver public key

Procedure:

1. Generate random session key.
2. Encrypt content using symmetric key.
3. Write content to the blockchain (userData stream) or IPFS (userData file)
4. Encrypt session key using asymmetric algorithm using sender's private key and receiver's public key.
5. Encrypted session key stored in the blockchain (ACL stream) or IPFS (ACL file)

Reading data

Requirements:

- Encrypted data
- Encrypted session key
- Public sender's key
- Private receiver's key

Procedure:

1. Get encrypted session key from the blockchain (userData stream) or IPFS (userData file).
2. Decrypt session key using asymmetric algorithm receiver's private key and sender's public key.
3. Get encrypted content from blockchain (userData stream) or IPFS (userData file).
4. Decrypt content using session key and symmetric algorithm.

The main advantage of the above accessibility approach is ability not to re-encrypt the userData (encrypt only once for the userData owner) and add only the new access client layer (ACL on the scheme).

By Design, once access is granted it cannot be taken away. The data consumer, having access, can always decrypt the data and access it.

Data versioning in the system allows access restriction for consumers but only to the new versions of information; e.g restricting access to new data.

Platform structure

The structure of the platform is designed to give the client a wide spectrum of abilities in data storage and business process reuse, combines with simplicity and intuitive logic.

The structure of the platform is outlined in Figure 7:

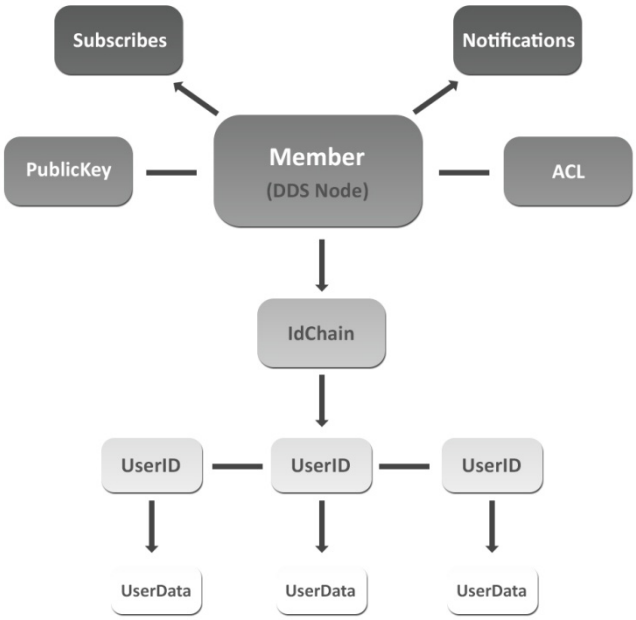

Figure 7.DDS Main components

Members (or DDS nodes) represent the module describing the node connected to blockchain.

IdChain is an array of UserIDs linking them under a single "chain ID" in the system.

UserID is the metadata with information about the user in which the user data is linked to.

UserData is a structured set of data in the JSON format.

Public key is an open key compliant with ECDiffieHellman, imported from the node system certificate. Each node must have a certificate with a public/private key pair. Before using the system, the certificate imports the public key into the system to grant access to the participants.

ACL are access permissions with the information on the node access to userIDs. They are used in private userIDs.

Subscribers contain the information about the subscribers of the changed/added userData notifications.

Notifications are a tool for informing the subscribers about the changed/added userData in the userIDs.

Technical solution

The dedicated blockchain platform provides a software application with a HTTP REST API allowing users to work with userIDs, userData, and subscriptions. At the bedrock of the unsanctioned access prevention system lays a hybrid cryptosystem, this makes it possible to combine the work speed of symmetric encryption algorithms with the reliability of asymmetric algorithms. In addition, it addresses the problem of encrypted data multiplication. Before getting recorded to the blockchain, the data is encrypted by the symmetrical Advanced Encryption Standard (AES) using the randomly generated key. After that, the key is encrypted by the asymmetric algorithm standard ECDiffielHellman individually for every single system participant with access to the user data. Then, these individually encrypted key copies are recorded to the blockchain. Only the system participant whose public key was used to encrypt the symmetric key can extract it and use it to decrypt the data.

The public key registry is stored in a dedicated blockchain stream which allows us to solve the common problem of hybrid cryptosystems – the exchange of participants' public keys. Current solutions suppose using a third-party solution for that or using external data exchange systems. This might lead to the problems resulting from the single point of failure or to the disclosure of the data to the third party when substituting the public key. Storing all the available public keys in a shared blockchain stream guarantees permanent availability of the necessary keys for exchange and its validity, at the first version of the system Senno will store this data on the NEO blockchain.

The system stores the set of userID metadata including the data validation scheme (JSON Schema) and the information about the userID access level in a separate blockchain stream.

Editing metadata through the API is only available to the userID creator in order to be secure from the direct change of the metadata in blockchain. The system scans for the initial userData version describing the metadata of this very userID and only accepts the updates made by the author of the initial version of the userData.

The platform enables receiving the data using pull methods by acquiring the userData from the userID by its ID, or within the push module by subscribing to the changes of the userData. To organize the subscription, it's mandatory to provide a service URL that will be called upon the emergence of a new userData in the userID.

Data Modules and API

MemberModule

- Address – the unique address of the system participant
- IsMe – the evidence that participant's address belongs to the current server
- Name – participant's associative name in the system
- Description – additional data on system participant

The REST API to work with the DDS nodes

Method	Signature	Description	Parameters	Return result
GET	/api/v1/manage/members	Get the list of system participants.		Member Module array or Error 500
POST	/api/v1/manage/members	System participant registration.	MemberModule	Error 200 or 400, or 500
DELETE	/api/v1/manage/members/{address}	Remove a participant from the system.	address - Participant address	Error 200 or 400, or 404 or 500
GET	/api/v1/manage/members/{address}	Get system participant data by the key.	address - Participant address	Member Module or Error 400 or 404, or 500
PUT	/api/v1/manage/members/{address}	Edit the system participant data.	address - Participant address and MemberModule	Error 200 or 400, or 404 or 500
POST	/api/v1/manage/members/sync	Update the system participant list from the node addresses list.		Error 200 or 500

IdChain

A list of all the userID's linking to a specific entity.

The idChain is created once a new user is added to the system. As the user adds additional userID's to the system, the idChain expands to include them.

The number of idChains in the system is unlimited.

idChainModule

- chainID – idChain unique id.
- userID – userIDModule array.
- Description – chain description.
- Key – idChain key.

- Protection – idChain security level
- WalletAddr – default generated NEP5 address.
- CreationDate – DataTime.

The REST API to work with ID Chain

Method	Signature	Description	Parameters	Return result
GET	/api/v1/manage/idchain	Get the idChain id list.		idChain Module array or Error 400 or 500
POST	/api/v1/manage/ idchains	Create a new idChain.	idChainModule	Error 200 or 400, or 500
GET	/api/v1/manage/ idchains/{ idchain }	Get the data describing the idChain properties	idChain – chainID	idChain Module or Error 400 or 404, or 500
PUT	/api/v1/manage/ idchains/{ idchain }	Update the existing idChain.	idChain – ChainID and idChainModule	Error 200 or 400 , or 500

UserID

The userID metadata management is a crucial part of running distributed data storage in a system.

The userID must be created before the system gets populated with data. The preliminary actions include userID naming (Latin characters), validation scheme, and access levels. In case of a protected userID, the array of addresses of the system nodes with access the userID data is mandatory.

The number of userID's in the system is unlimited and depends on the business needs in distributed data storage.

Only the owner of the userID can edit it.

Once created, the userID cannot be deleted from the system, instead the encryption keys are deleted and accessing the data becomes impossible.

userIDModule

- **Name –** user name.
- **Title –** user title.
- **Description** – user description.
- **Version** – userID version (read only). Every time the array of participant addresses or protection level changes, the userID version automatically increases by 1.
- **Key** – userID key.
- **userIDKeyModule** – the module describes the rules of the unique key generation for the userID data.
- **Fields** – the list of user data properties from the values of which the key is generated.
- **Separator** – property value separator.
- **Protection** – userID security level. Currently it can take the following values: "public" - open data userID, "private" – userID - based protection level.
- **Schema** – JSON object describing the userID data validation. If left blank, the user data won't be validated during saving. Validation format is JSON Schema.
- **Members** – the array of node addresses with access to the userID data. If the userID is public, the array is not passed. In case of IPFS there shouldn't be any node with access (array will not exist) => this parameter only exist in consortium with private userID.

The REST API to work with the userID metadata

Method	Signature	Description	Parameters	Return result
GET	/api/v1/manage/userid	Get the userID names list.		userIDModule array or Error 400 or 500
POST	/api/v1/manage/userids	Create a new userID.	userID Module	Error 200 or 400, or 500
GET	/api/v1/manage/userids/{userid}	Get the data describing the userID properties.	userID – user name	userIDModule or Error 400 or 404, or 500
PUT	/api/v1/manage/userids/{userid}	Update the existing userID.	userID – user name and userID Module	Error 200 or 400 , or 500

UserData

The main information storage entity in the system is a userData.

The format of the userData is JSON.

Each userData saved in the system contains the following parameters:

userDataModule

Key – the key generated based on the rules of key generation specified in the userID metadata. For private userID's, the key is stored in an encrypted with AES algorithm.

TxID – the unique userData ID assigned upon saving the userData to a userID.

Data – the user data itself passed to be saved in the JSON format. For private userID's, the data is stored in an encrypted form with AES 256.

The module describes the data received from the client to add/update the userData:

userDataRequestModule

Data – userData value fields.

ACLs – the list of system participants with access to the userData. For future use with protection levels by the key and identifier.

The REST API to work with userData

Method	Signature	Description	Parameters	Return result
GET	/api/v1/userids/{user id}/keys	Get the list of unique keys	userID – userID name, skip – the number of elements skipped in the page-by-page input, take – the number of elements received in the page-by-page input.	The array of userID keys or Error 400 or 401, or 404 or 500
GET	/api/v1/userids/{user id}/{txid}	Get the userData version by the ID	userID – userID name, txid – unique userData ID.	userData Module or Error 400 or 401, or 404 or 500
GET	/api/v1/userids/{user id}/userdata/{key}	Get the userData by the key	userID – userID name, key – userData key.	userData Module or Error 400 or 401, or 404 or 500
PUT	/api/v1/userids/{user id}/userdata /{key}	Keyed userData update	userID – userID name, key – userData key, userDataModule	Error 200 or 400 or 401, or 404 or 500

GET	/api/v1/userids/{user id}/userdata /search	Get the userData by the parameters	userID – userID name, parameters – search parameters in the JSON format (the parameters used to form the userData key).	userData Module or Error 400 or 401, or 404 or 500
POST	/api/v1/userids/{user id}/userdata /search	Get the userData by the parameters	userID – userID name, parameters – search parameters in the JSON format (the parameters are passed in the JSON format. e.g: {country:"UK", gender:"male"} {age:"22-45","hobby:sports"})	userData Module or Error 400 or 401, or 404 or 500
GET	/api/v1/userids/{user id}/userdata /{key}/versions	Get the list of userData changes	userID – userID name, key – userData key.	userData Module or Error 400 or 401, or 404 or 500
GET	/api/v1/userids/{user id}/userdata /{key}/versions/{txi d}	Get the specified userData version	userID – userID name, key – userData key, txid – unique userData ID.	userData Module or Error 400 or 401, or 404 or 500
POST	/api/v1/userids/{user id}/userdata	Create new userData	userID – userID name, userDataModule	Error 200 or 400 or 401, or 404 or 500