8

i3-MARKET Crypto Token and Data Monetization

8.1 Objectives

The federation of independent data spaces/marketplaces further calls for a highly secure, trusted, and cost-efficient payment solution.

At first a standard payment solution has been designed providing a protocol to exchange data with a non-repudiable and auditable accounting of data transfers. This ensures transparent billing and support for conflict resolution.

This protocol is based on a cryptographic proof exchange between data consumer and data provider and a final recording of this proof on the blockchain as “notarization” on the data exchange.

Then a tokenization solution has been designed providing a crypto token based on Ethereum standard ERC-1155 and the concept of “distributed treasury”, which means that each data marketplace joining the federation could exchange token for fiat money with a fixed value. The tokens minted by each marketplace are “tagged” differently so that there is always a link between the tokens and the issuer, which must provide the associated amount of fiat money during a “clearing” phase between the data marketplaces.

This allows instant currency exchange among all the stakeholders participating in the federation and also supports full audibility of all transactions.

However, until the landscape of cryptocurrencies and tokens is clarified, with the EU Parliament vote on adopting MiCA(https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52020PC0593) regulation, which is expected to establish harmonized rules for crypto-assets at the EU level, thereby providing legal certainty and guidance as to the usage of crypto tokens, the i3-MARKET Alliance decided to only use tokens as a means for distributing fees for the long-term sustainability of the system.

The choice to establish a new crypto token for real-time trading of data assets between federated data spaces and marketplaces in the i3-MARKET platform was made to overcome the boundaries of individual marketplaces and build a trustworthy working environment, one of the keys to establishing a single European data economy.

In particular, the problems solved by the adoption of the token as a currency (like the use of the blockchain in our non-repudiable protocol) are:

- to exchange value in a peer-to-peer manner, without the need for someone in between;
- to make sure that ownership is transferred and that information about the exchange cannot be tampered with.

This is extremely disruptive for the market because information is decentralized and the control is distributed among all network marketplaces, thus avoiding the designation of an impartial central intermediary. Decentralized communities provide certainty of identity, certainty of provenance and the smart contracts, like the treasury smart contract designed here, and certainty of execution ("if I pay with tokens, I get value in return") in the network.

The main technical contributions are the design and development of an *ad-hoc* blockchain-based non-repudiation protocol and the design of a crypto token solution based on the concept of "distributed treasury" to allow and trust the real-time trading of data assets among federated data spaces and marketplaces.

The designed non-repudiable protocol, instead of relying on trusted third parties for storing the non-repudiation evidence information, uses the Hyperledger BESU blockchain deployed on the i3-MARKET nodes that preserve both the proof of origin and proof of receipt of the parties involved. This is to ensure two things: one is that the information sent cannot be denied, for example, a DC has sent a message to a DP, so the DC cannot deny the behaviour. The other is that the recipient of the information cannot be denied. Similarly, DP has sent a message to DC, but DC cannot claim that it has not received this message. In this sense, the blockchain guarantees both the non-repudiation of information and the non-tampering of data by the parties involved.

The tokenization component and the cryptographic token flow have been designed to support all the different interactions between the subjects of i3-MARKET, namely data marketplace, data provider, and data consumer. To enable the trading of assets among the i3-MARKET network marketplaces, a custom flow has been designed consisting of four different phases, which, starting from the mint of the token from a data marketplace, allow it to be used as a means of payment for data or fees and to trace the path up to the return to the original marketplace, where it is burned. In particular, the use of the token in these phases enables payments that are easy to make, reliable, safe, and verifiable both within and between marketplaces and allows correct management to differentiate the tokens issued by different marketplaces.

8.2 Technical Requirements

For the components of the data monetization subsystem, the following requirements have been defined in the form of epics and user stories.

Standard payments:

Epics:

Name	**Description**	**Labels**
Standard payments	Standard payments refer to payments for a specific dataset or piece of data. Initially, it should support: payment in advance – *a posteriori* payment. In both cases, it should also support: pay per dataset or specific piece of data – subscription (flat rate within a specific set of conditions)	Epic data Consumer Data provider Data marketplace

User stories:

Name	**Description**	**Labels**
In advance payment	As a data provider, I want to be paid in advance for providing my data so that I can monetize them immediately	User story Data provider Data consumer Data marketplace
***A posteriori* payment**	As a data provider, I want to be paid *a posteriori* for providing my data so that I can have more consumers to subscribe my offering	User story Data marketplace Data consumer Data provider
Non-repudiation Protocol	As a data provider, I want to provide my data with a Non-repudiation Protocol so that I can bill data consumers based on reliable data exchanges As a data consumer, I want to consume data with a non-repudiation protocol so that I can contest wrong billings	User story Data consumer Data provider

Tokenization:

Epics:

Name	Description	Labels
Currency tokenization	The federation of independent data spaces/marketplaces further calls for a highly secure, trusted, and cost-efficient payment solution. Therefore, a suitable crypto currency solution that allows instant currency exchange among the participating data spaces/marketplaces and also supports full audibility of all transactions has to be provided	Epic Data marketplace Data consumer Data provider Data owner

User stories:

Name	Description	Labels
Provide crypto tokens (exchange in)	As a data marketplace, I want to provide crypto tokens to data consumer so that I will enable P2P payments for data exchange As a data consumer, I want to purchase crypto tokens from a data marketplace so that I can subscribe offering from other marketplaces in the i3-MARKET network	User story Data marketplace Data consumer
Payment with crypto tokens	As a data provider, I want to receive payment with crypto tokens so that I can receive instant payments	User story Data marketplace Data consumer Data provider Data owner
Withdraw crypto tokens (exchange out)	As a data provider/data owner, I want to receive fiat currency from a data marketplace so that I can monetize the crypto tokens received for providing my data As a data consumer, I want to receive fiat currency from a data marketplace so that I can monetize my crypto tokens if I leave the i3-MARKET network	User story Data marketplace Data consumer Data provider Data owner
Clearing	As a data marketplace, I want to receive fiat currency for the tokens emitted by other data marketplaces so that I can monetize these tokens if I leave the i3-MARKET network or with a specific scheduling	User story Data marketplace

8.3 Solution Design/Blocks

The following are the high-level capabilities provided by the data monetization subsystem:

1. Standard payments: In advance or *a posteriori* payment for a specific dataset or piece of data.
2. Tokenization: Creation of a crypto token solution for instant currency exchange among the participating data spaces/marketplaces.
3. Pricing manager: Managing of i3-MARKET cost and price model.

From Figure 8.1, the data monetization subsystem block interacts with following two building blocks:

– Data storage system: The data monetization subsystem uses the data Storage system for recording crypto token transactions.
– Backplane system: The data monetization subsystem has been used from the Backplane system for accounting and executing payment operations for data purchases and tokenization operations.
– Data access system: The data monetization subsystem has been used from the data access system for accounting and/or executing payments for data exchanges.

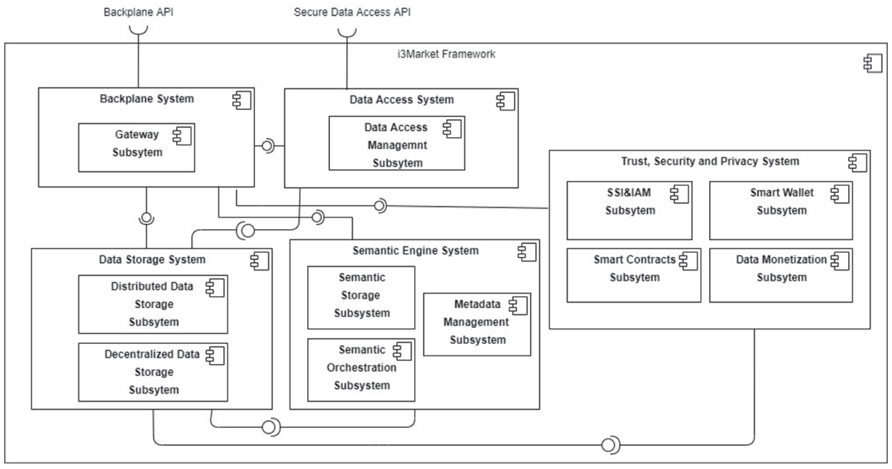

Backplane API
Secure Data Access API
i3Market Framework
Backplane System
Gateway Subsytem
Data Access System
Data Access Managemnt Subsytem
Trust, Security and Privacy System
SSI&IAM Subsytem
Smart Wallet Subsytem
Smart Contracts Subsytem
Data Monetization Subsytem
Data Storage System
Distributed Data Storage Subsytem
Decentralized Data Storage Subsytem
Semantic Engine System
Semantic Storage Subsystem
Metadata Management Subsystem
Semantic Orchestration Subsystem

Figure 8.1 Backplane architecture.

See Figure 8.2 for the *specific component diagram.*

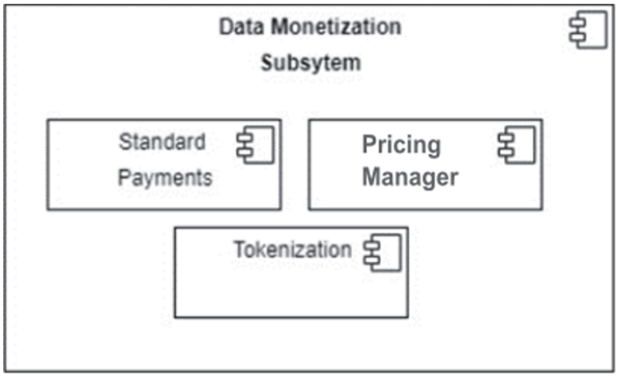


Figure 8.2 Data monetization components.

The data monetization subsystem is in charge of providing "standard payments", "pricing manager", and "tokenization" capabilities.

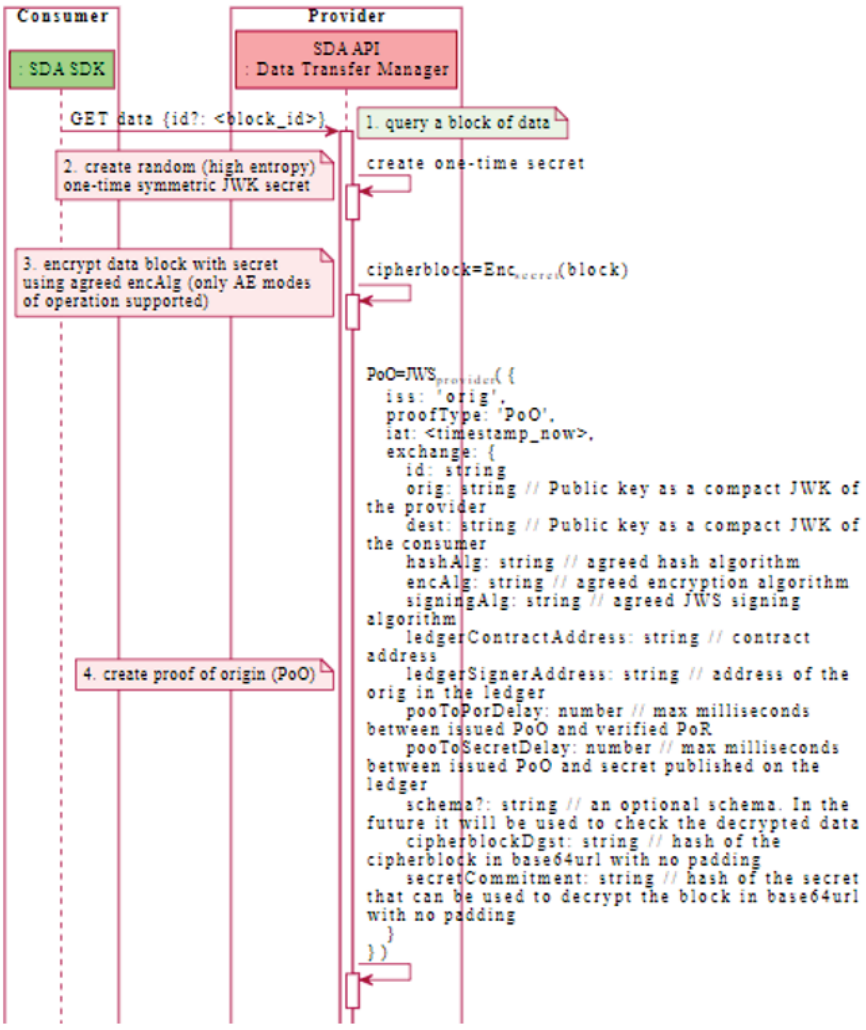
Inside, we can find:

– component "standard payments" responsible for managing the payments, in advance or *a posteriori*, for a specific dataset or piece of data;
– component "tokenization" responsible for the creation of a crypto token for instant currency exchange and other tokenization operations among the participating data spaces/marketplaces;
– component "pricing manager" responsible for managing the price and the cost model.

8.4 Standard Payment

The Non-repudiation Protocol aims at preventing parties in a data exchange from falsely denying having taken part in that exchange. The protocol flow begins when a data consumer requests a block of data from the data provider. See Figure 8.3 for details.

At first, the data provider has to generate a one-time symmetric secret key (JWK) for a given JWA algorithm identifier; this secret is going to be used to encrypt the data block required by the data consumer (Figure 8.4). After the data encryption, the data provider builds a proof of origin JWT object, containing information about the parties (source, destination, etc.), the timestamp, the hash algorithm used, the hash of the block, the secret key, and the encrypted cipherblock. See Figure 8.5 for more details.


Consumer
Provider
: SDA SDK
SDA API
: Data Transfer Manager
GET data {id?: <block_id>}
1. query a block of data
2. create random (high entropy) one-time symmetric JWK secret
create one-time secret
3. encrypt data block with secret using agreed encAlg (only AE modes of operation supported)
4. create proof of origin (PoO)

$\text{cipherblock}=\text{Enc}_{\text{secret}}(\text{block})$

```
PoO=JWS_provider({
  iss: 'orig',
  proofType: 'PoO',
  iat: <timestamp_now>,
  exchange: {
    id: string
    orig: string // Public key as a compact JWK of
the provider
    dest: string // Public key as a compact JWK of
the consumer
    hashAlg: string // agreed hash algorithm
    encAlg: string // agreed encryption algorithm
    signingAlg: string // agreed JWS signing
algorithm
    ledgerContractAddress: string // contract
address
    ledgerSignerAddress: string // address of the
orig in the ledger
    pooToPorDelay: number // max milliseconds
between issued PoO and verified PoR
    pooToSecretDelay: number // max milliseconds
between issued PoO and secret published on the
ledger
    schema?: string // an optional schema. In the
future it will be used to check the decrypted data
    cipherblockDgst: string // hash of the
cipherblock in base64url with no padding
    secretCommitment: string // hash of the secret
that can be used to decrypt the block in base64url
with no padding
  }
})
```

Figure 8.3 NRP Phase 1 — consumer gets cipherblock and non-repudiable proof of origin.

This object is then signed with the private key of the data provider and returned to the data consumer.

The data consumer, at this point (Figure 8.5), can validate the proof received using the data provider public key. If the validation is successful, he can store the proof in his local memory. After having completed these steps, he generates another proof, the proof or receipt. This proof is generated as

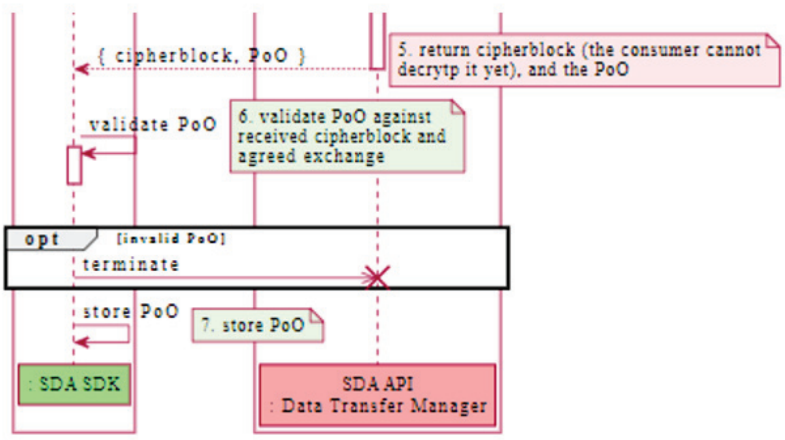
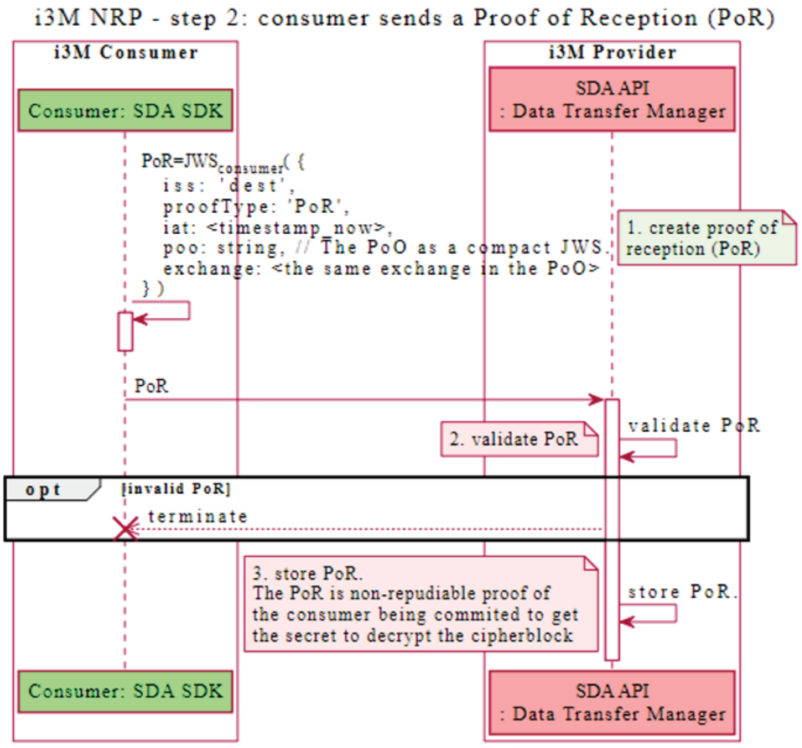

{ cipherblock, PoO }
5. return cipherblock (the consumer cannot decrytp it yet), and the PoO
validate PoO
6. validate PoO against received cipherblock and agreed exchange
opt [invalid PoO]
terminate
store PoO
7. store PoO
: SDA SDK
SDA API : Data Transfer Manager

Figure 8.4 NRP Phase 1 Part 2.

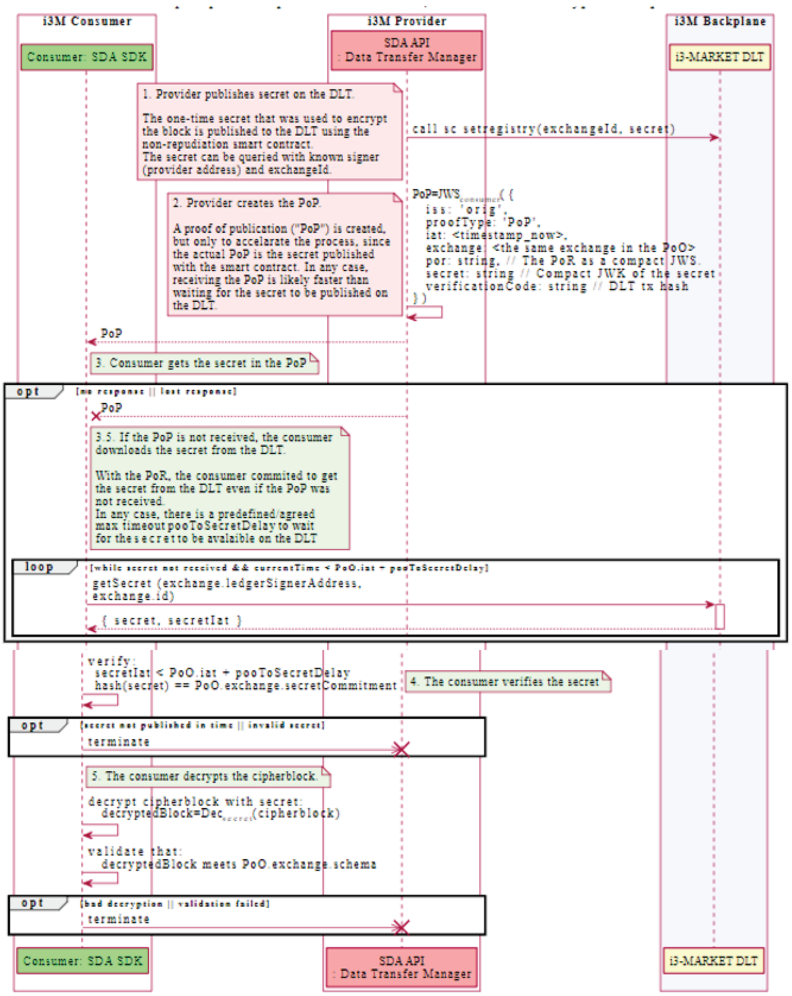

i3M NRP - step 2: consumer sends a Proof of Reception (PoR)
i3M Consumer
i3M Provider
Consumer: SDA SDK
SDA API : Data Transfer Manager
1. create proof of reception (PoR)
PoR
2. validate PoR
validate PoR
opt [invalid PoR]
terminate
3. store PoR. The PoR is non-repudiable proof of the consumer being commited to get the secret to decrypt the cipherblock
store PoR.

```
PoR=JWS_consumer( {
  iss: 'dest',
  proofType: 'PoR',
  iat: <timestamp_now>,
  poo: string, // The PoO as a compact JWS.
  exchange: <the same exchange in the PoO>
} )
```

Figure 8.5 NRP consumer sends PoR.

another JWT object containing information about the parties (iss, sub, etc.), the timestamp, the hash of the received proof of origin, and the hash algorithm used.

The proof of receipt is then signed with the data consumer private key and sent back to the data provider. Once proof of receipt is received, the data provider can validate it using the data consumer public key; see Figure 8.6.



1. Provider publishes secret on the DLT.

The one-time secret that was used to encrypt the block is published to the DLT using the non-repudiation smart contract.
The secret can be queried with known signer (provider address) and exchangeId.

```
call sc setregistry(exchangeId, secret)
```

2. Provider creates the PoP.

A proof of publication ("PoP") is created, but only to accelarate the process, since the actual PoP is the secret published with the smart contract. In any case, receiving the PoP is likely faster than waiting for the secret to be published on the DLT.

```
PoP=JWS_consumer({
  iss: 'orig',
  proofType: 'PoP',
  iat: <timestamp_now>,
  exchange: <the same exchange in the PoO>
  por: string, // The PoR as a compact JWS.
  secret: string // Compact JWK of the secret
  verificationCode: string // DLT tx hash
})
```

3. Consumer gets the secret in the PoP

3.5. If the PoP is not received, the consumer downloads the secret from the DLT.

With the PoR, the consumer commited to get the secret from the DLT even if the PoP was not received.
In any case, there is a predefined/agreed max timeout pooToSecretDelay to wait for the secret to be avalaible on the DLT

```
getSecret (exchange.ledgerSignerAddress,
exchange.id)
{ secret, secretIat }
```

```
verify:
  secretIat < PoO.iat + pooToSecretDelay
  hash(secret) == PoO.exchange.secretCommitment
```

4. The consumer verifies the secret

5. The consumer decrypts the cipherblock.

```
decrypt cipherblock with secret:
  decryptedBlock=Dec_secret(cipherblock)
validate that:
  decryptedBlock meets PoO.exchange.schema
```

Figure 8.6 NRP provider publishes the secret, and consumer decrypts the cipherblock.

The provider now publishes the one-time secret that was used to encrypt the block on the i3M BESU blockchain. A proof of publication is then created but only to accelerate the process, since the actual proof of publication is the secret published within the smart contract. The PoP is then sent to the data consumer.

If the data consumer does not receive the proof and the key in a predefined/agreed max timeout, he can retrieve them directly from the blockchain. Once having received the proof of publication and the secret key, the data consumer can validate the proof of publication with the auditable accounting public key and verify that the hash of the key received is equal to the hash of the key previously received in the proof of origin.

As the last step, if the verification is successful, he can decrypt the block with the secret key and validate it with the hashed block included in the proof of origin. If some validation or verification problems arise, the flow will enter a conflict resolution phase.

8.5 Tokenization

The tokenization process and the components used to create and manage a cryptographic token for instant currency exchange and other tokenization operations among participating i3-MARKET actors are represented in Figure 8.7.

i3-MARKET tokenization architecture:

Starting from the right of the architecture in Figure 8.21, to manage the operational flows between the various data spaces/marketplaces involved in

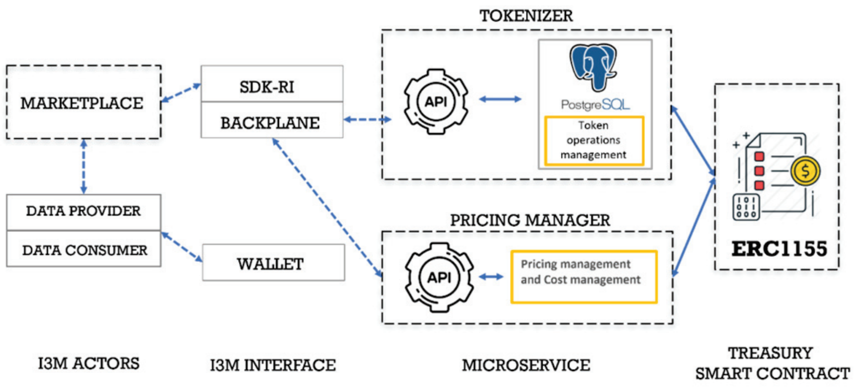


Figure 8.7 Tokenization process.

currency exchange within the i3-MARKET platform, a specific i3-MARKET treasury smart contract has been created. This smart contract contains and maintains for each wallet the token balance of the data marketplaces, DP, DC, and community members. More specifically, it is responsible for managing the secure transfer of tokens between the parties and for tracking immutably payments made in both tokens and fiat money.

To enable interaction with the treasury smart contract functionalities, we have created two microservices, the tokenizer and the pricing manager. The tokenizer allows the i3-MARKET actors to interact with the treasury smart contract and keep track of all the marketplace operations, and the pricing manager manages the data price and the fees. These two services are integrated with the i3M Backplane and the SDK-RI to be used from an i3-MARKET DM.

Treasury smart contract operations:

The most important features involving the tokenization operations are presented below; these have been implemented within the treasury contract, which extends the ERC-1155 standard.

- **Register a data marketplace:**

To register a new data marketplace and its token type, a mapping to bond the data marketplace addresses and the index identifier of the new token type is required.

The function that inserts a new marketplace in the smart contract, increments an index variable and is added in the mapping of the marketplace address as key and using a unit value as the identifier of the new token type.

```
contract I3-MARKETTreasury is ERC1155 {

    uint public index = 0;
    mapping(address => uint) public mpIndex;

    constructor() public ERC1155("https://i3-MARKET.com/marketplace/{id}.json"){
    }

    /*
    * add a new Data Marketplace in the platform
    */
    function addMarketplace(address _mpAdd) external onlySameAdd(_mpAdd)
onlyNewMpAdd(_mpAdd) {
        index += 1;
        marketplaces.push(_mpAdd);
        mpIndex[_mpAdd] = index;
    }
}
```

- **Exchange in:**

The exchange method must be called by a data marketplace, which issues and transfers the right amount of tokens (of its token type) to the user who pays in fiat money.

ERC1155 function:

_mint(address account, uint256 id, uint256 amount, bytes data)

- **Fee payment:**

The payment method should transfer the token fees, taken "arbitrarily" from the token types available in the data consumer balance, to the data provider.

The ERC1155 function used for the payment:

```
safeBatchTransferFrom(address from, address to,
uint256[] ids, uint256[] amounts, bytes data)
```

Example: Starting from the first token type in the balance loop until the amount is covered.

```
function configurePayment(address from, uint256 amount) private view returns
(uint256[] memory ids, uint256[] memory amounts) {
uint256[] memory mpIds = new uint256[](index);
        uint256[] memory mpTokens = new uint256[](index);
        for (uint256 i = 0; i < index && amount != 0; ++i) {
            uint256 mpBalance = super.balanceOf(from, i + 1);
            if (mpBalance != 0) {
                mpIds[i] = i + 1;
                mpTokens[i] = getMarketplaceNeededTokens(mpBalance, amount);
                amount = amount - mpTokens[i];
            }
        }
        require(amount == 0, "NOT ENOUGH TOKENS");
        return (mpIds, mpTokens);
}
```

- **Exchange out:**

The *exchange-out* method should transfer the right amount of token, taken "arbitrarily" (first the tokens belonging to the data marketplace in the exchange out and once finished, the others) from the token types available in the balance, from a community member to a data marketplace.

The ERC1155 function used for the *exchange-out* operation:

```
safeBatchTransferFrom(address from, address to,
uint256[] ids, uint256[] amounts, bytes data)
```

- **Clearing:**

The clearing method should be called for every token type present in the data marketplace balance, aside from the token type the data marketplace has created.

The ERC1155 function used for the clearing operation:

`safeTransferFrom(address from, address to, uint256 id, uint256 amount, bytes data)`

```
struct ClearingOperation{
    string transferId;
    address toAdd;
    uint tokenAmount;
}

function clearing(ClearingOperation[] memory _clearingOps) external payable
onlyMp(msg.sender){

    //clearing for each marketplace contained
    for (uint i = 0; i < _clearingOps.length; ++i){
        isMarketplace(_clearingOps[i].toAdd,"ADD ISN'T A MP");
        if(_clearingOps[i].tokenAmount > minimumClearingThreshold) {
            super.safeTransferFrom(msg.sender,_clearingOps[i].toAdd,mpIndex[_clear
ingOps[i].toAdd], _clearingOps[i].tokenAmount, "0x0");

            //create transaction with isPaid param to False as Fiat money payment
is not completed yet
            txs[_clearingOps[i].transferId] =
TokenTransfer(_clearingOps[i].transferId, msg.sender, _clearingOps[i].toAdd,
_clearingOps[i].tokenAmount, false, "");
            emit TokenTransferred(_clearingOps[i].transferId, "clearing",
msg.sender, _clearingOps[i].toAdd);
        }
    }
}
```

- **Exchange-out and clearing strategy:**

Since a marketplace has first to collect the fiat money from all the DM involved in an exchange-out operation before transferring the money requested, here we describe the suggested strategy that each marketplace should implement in its code. Other strategies can be used in agreement with the network marketplaces.

Requirements:

- Set a number variable **X** as the interval of days that a marketplace collects exchange-out requests.

- Set a numeric variable **Z** as the interval of days in which a marketplace must wait for other marketplaces to pay in fiat money for the tokens sent via clearing.

Strategy flow:

Marketplace ordered steps for the exchange-out operation:

1. Starting from the first day a marketplace starts operating, in the first **X** days, the marketplace should collect all the exchange-out requests from the users (community).
2. At the end of **X** days, the marketplace asks to exchange the tokens in its balance that belong to other marketplaces.
3. Now the marketplace should wait for another **Z** days so that all the other marketplaces can pay with fiat money the tokens sent with the clearing operation.
4. Once the **Z** days have passed and the fiat money from the clearings are collected, the marketplace can pay out the users that in the first **X** days requested the exchange-out of tokens.

The marketplace can restart in parallel this flow and collect another round of exchange-out requests at the end of point 2.

Tracking of token transfers (exchange out, clearing):

The token payment process involves storing in the blockchain the history of the transactions made once the token transfer is completed.

We want to save the transfer operation in a mapping of structs, where the key identifier is a unique value generated outside.

```
// object that stores the token transfer information
struct TokenTransfer {
    uint transferId;
    address fromAddress;
    address toAddress;
    uint tokenAmount;
    bool isPaid; //True if the fiat money payment has been completed
                 //False if only the token transfer is completed
    string transferCode;

}

//mapping to track all the token transfer transactions
mapping(uint => TokenTransfer) public transactions;
```

Tokenizer service:

The main purpose of the tokenizer service is to allow the i3-MARKET actors (i.e., marketplace, data provider, and data consumer) to call the i3-MARKET treasury smart contract methods and interact with the i3-MARKET token flow; see Figure 8.8. The tokenizer is a Node.js backend service with a local Postgres database to persist the marketplace token activities.

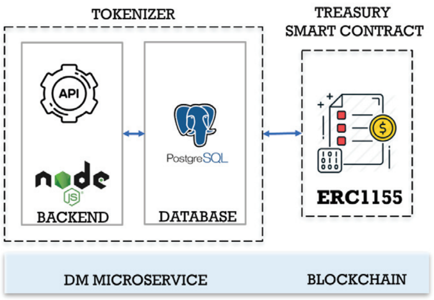
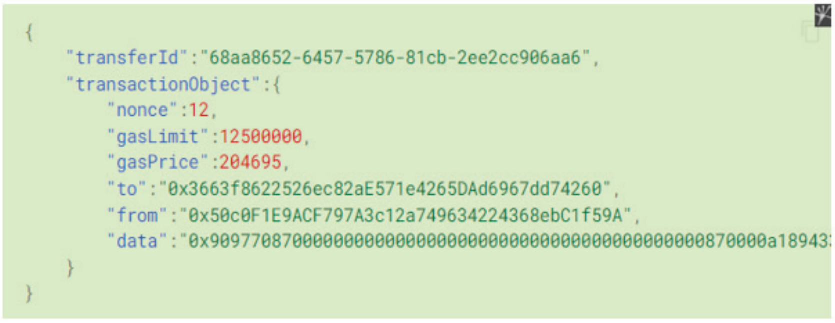


Figure 8.8 Tokenizer architecture.

Each data marketplace needs an instance of the tokenizer and a dedicated local database. The tokenizer tracks the status of transactions made by a marketplace on the treasury smart contract using blockchain events, while the full history of each transaction is stored in the local database for verification and error prevention.

To deploy a transaction on the i3M BESU blockchain, the transaction should be first signed with the i3-MARKET Wallet. The tokenizer *post* operations create a transaction object that must be signed and then deployed separately using the flow presented below.

1. The first step is to create a new raw transaction using one of the *post* operations available (i.e., exchangeIn, exchangeOut, clearing, fee-payment, etc.). After a successful transaction, the payload of the response will be a transaction object like this one:

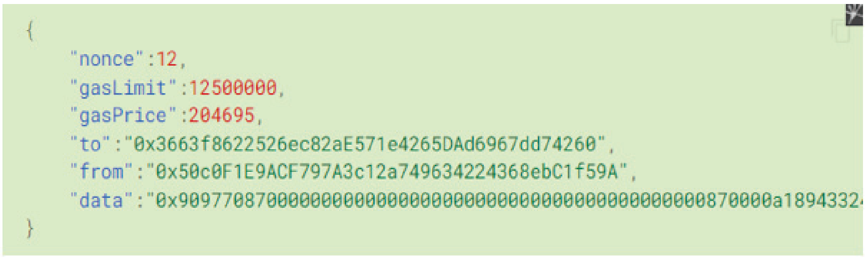


```
{
    "transferId":"68aa8652-6457-5786-81cb-2ee2cc906aa6",
    "transactionObject":{
        "nonce":12,
        "gasLimit":12500000,
        "gasPrice":204695,
        "to":"0x3663f8622526ec82aE571e4265DAd6967dd74260",
        "from":"0x50c0F1E9ACF797A3c12a749634224368ebC1f59A",
        "data":"0x909770870000000000000000000000000000000000000000870000a18943
    }
}
```

With this operation, the marketplace tokenizer service saves in its database the operation with the operation_name (i.e., exchangeIn, exchangeOut, clearingIn, clearingOut, fee-payment, etc.) with status *UNSIGNED-OPERATION*, the address of the user involved, the date, and a unique transferID to get this operation at a later time:

TRANSFERID	OPERATION	STATUS	USER	DATE
1111	operation_name	unsigned_operation	address	date

2. Now the raw transaction has to be signed with the i3M wallet. Only the fields contained in the "transactionObject" are used for the signing transaction operation. Below is the object to be signed:



```
{
    "nonce":12,
    "gasLimit":12500000,
    "gasPrice":204695,
    "to":"0x3663f8622526ec82aE571e4265DAd6967dd74260",
    "from":"0x50c0F1E9ACF797A3c12a749634224368ebC1f59A",
    "data":"0x909770870000000000000000000000000000000000000000870000a1894332
}
```

3. The next step is to deploy the signed transaction. Once the marketplace gets the signed raw transaction, it can call the deployment endpoint of the tokenization service /treasury/transactions/deploy-signed-transaction. The response of the request should be a long transaction object with information about the transaction.
4. When the operation deployment is successful, the marketplace tokenizer service updates in its database the previous operation with status open.

TRANSFERID	OPERATION	STATUS	USER	DATE
1111	operation_name	open	address	Later-date

Pricing manager service:

Pricing manager is a Java microservice to configure and evaluate the price and the cost of data; see Figure 8.9. The microservice uses the i3-MARKET BESU blockchain and an in-memory database to persist data.

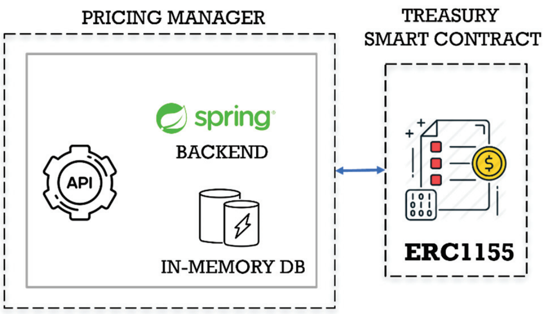


Figure 8.9 Pricing manager architecture.

The service APIs are logically divided into two subsets.

Pricing management:
This service allows to calculate the price of some data based on a preconfigured formula. The service, through the exposed APIs, allows you to manage the formula and customize the parameters and constants.

The formula and the constant values are stored in an in-memory database inside the service, as every marketplace can have a customized formula if needed.

Currently, in the i3-MARKET platform, all the data marketplaces will use the formula provided by AUEB.

Cost management:
This service can be used to calculate the fee of some data, which depends on the price of the data and the percentage of the fee. The fee percentage is stored in the blockchain and more specifically in the treasury smart contract.

8.6 Diagrams

The following diagrams describe the processes involving the components of the data monetization subsystem.

These requirements have been collected using the Trello Boards system taking into account functional use cases and general requirement reported by partners, stakeholders, big companies, and SMEs.

Standard payment:

A Non-repudiable Protocol is used for accounting data transfers. Based on the accounted data exchanges and smart contract information about data consumer (company name, VAT, billing address, etc.) and pricing, the data provider will invoice the data consumer; see Figure 8.5. The payment will be done using standard bank payment methods.

- **Accounting:**

The Non-repudiation Protocol aims at preventing parties in a data exchange from falsely denying having taken part in that exchange. To ensure the traceability of data exchanges and manage conflicts, the proof of origin of the data provider and the proof of receipt of the data consumer are stored in the immutable ledger; see Figure 8.10.

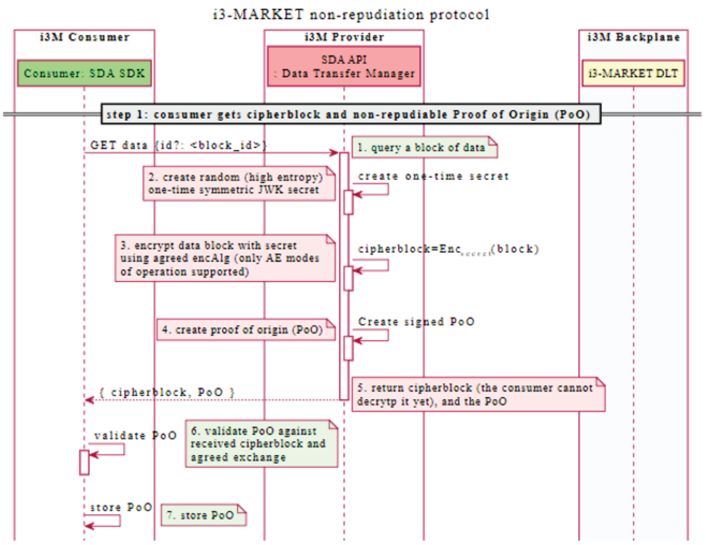

i3-MARKET non-repudiation protocol
i3M Consumer
Consumer: SDA SDK
i3M Provider
SDA API : Data Transfer Manager
i3M Backplane
i3-MARKET DLT
step 1: consumer gets cipherblock and non-repudiable Proof of Origin (PoO)
GET data {id?: <block_id>}
1. query a block of data
2. create random (high entropy) one-time symmetric JWK secret
create one-time secret
3. encrypt data block with secret using agreed encAlg (only AE modes of operation supported)
cipherblock=Enc_secret(block)
4. create proof of origin (PoO)
Create signed PoO
{ cipherblock, PoO }
5. return cipherblock (the consumer cannot decrytp it yet), and the PoO
validate PoO
6. validate PoO against received cipherblock and agreed exchange
store PoO
7. store PoO

Figure 8.10 NRP Part 1.

In the first step of the protocol, shown in Figure 8.11, a block of data is requested from the data consumer to the data provider. The data provider encrypts the requested block, creates the proof of origin, and returns this proof to the consumer with the encrypted block. At this point, the data consumer validates the proof of origin received from the data provider, stores the proof in its local memory, and creates the proof of receipt.

In the second step, the data consumer sends the proof to the data provider; see Figure 8.11. The data provider validates the proof of receipt and stores it in local storage. As a third step, the provider publishes the secret on the blockchain and sends the proof of publication to the data consumer.

The data consumer can obtain the proof of publication and the cryptographic key directly from the data provider or from the blockchain if this is not received within a maximum time; see Figures 8.12 and 8.13. The data consumer then checks that the key is received, that the PoP is valid, and that

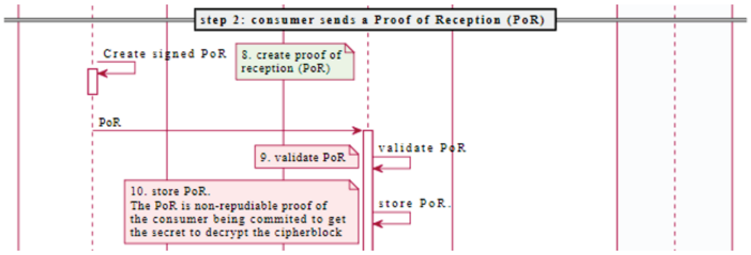

step 2: consumer sends a Proof of Reception (PoR)
Create signed PoR
8. create proof of reception (PoR)
PoR
validate PoR
9. validate PoR
10. store PoR. The PoR is non-repudiable proof of the consumer being commited to get the secret to decrypt the cipherblock
store PoR.

Figure 8.11 NRP Part 2.

the key hash is the same as the key commitment parameter included within the PoO. At this point, the data consumer decrypts and validates the previously received cipherblock.

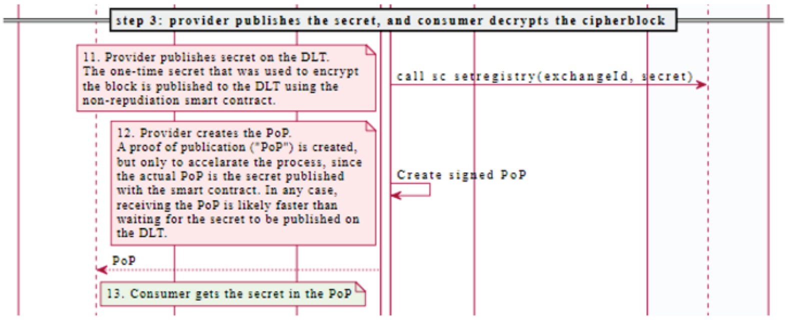

step 3: provider publishes the secret, and consumer decrypts the cipherblock
11. Provider publishes secret on the DLT. The one-time secret that was used to encrypt the block is published to the DLT using the non-repudiation smart contract.
call sc setregistry(exchangeId, secret)
12. Provider creates the PoP. A proof of publication ("PoP") is created, but only to accelarate the process, since the actual PoP is the secret published with the smart contract. In any case, receiving the PoP is likely faster than waiting for the secret to be published on the DLT.
Create signed PoP
PoP
13. Consumer gets the secret in the PoP

Figure 8.12 NRP Step 3 Part 1.

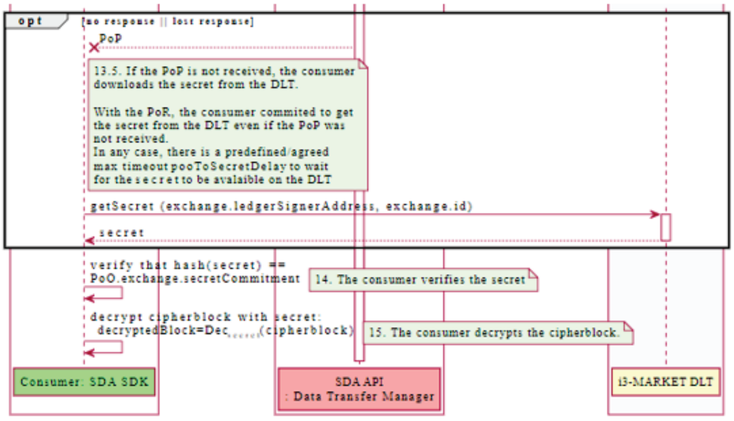

opt
[no response || lost response]
PoP
13.5. If the PoP is not received, the consumer downloads the secret from the DLT. With the PoR, the consumer commited to get the secret from the DLT even if the PoP was not received. In any case, there is a predefined/agreed max timeout pooToSecretDelay to wait for the secret to be avalaible on the DLT
getSecret (exchange.ledgerSignerAddress, exchange.id)
secret
verify that hash(secret) == PoO.exchange.secretCommitment
14. The consumer verifies the secret
decrypt cipherblock with secret: decryptedBlock=Dec(cipherblock)
15. The consumer decrypts the cipherblock.
Consumer: SDA SDK
SDA API : Data Transfer Manager
i3-MARKET DLT

Figure 8.13 NRP Step 3 Part 2.

- **Invoice management:**

In this process, the data provider retrieves from the Backplane the information (based on the accounted data exchanges and the smart contract agreement) to produce the invoices for the data consumers.

Data consumers could check the invoices verifying the accounted information and pay the invoices with standard payment methods; see Figure 8.14.

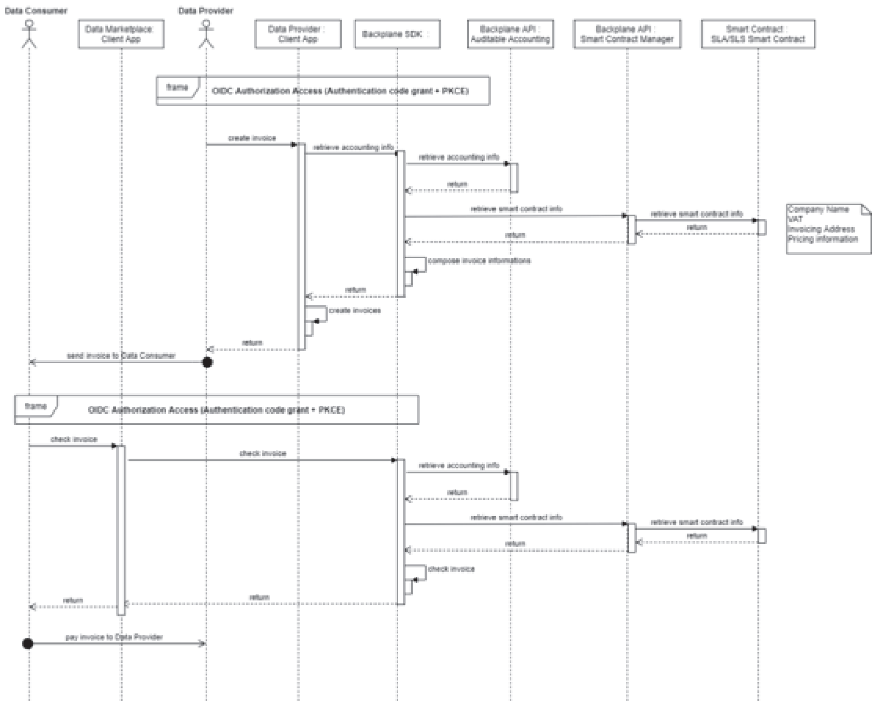

Data Consumer
Data Marketplace: Client App
Data Provider
Data Provider : Client App
Backplane SDK :
Backplane API : Auditable Accounting
Backplane API : Smart Contract Manager
Smart Contract : SLA/SLS Smart Contract
frame
OIDC Authorization Access (Authentication code grant + PKCE)
create invoice
retrieve accounting info
retrieve accounting info
return
retrieve smart contract info
retrieve smart contract info
return
return
Company Name
VAT
Invoicing Address
Pricing information
compose invoice informations
return
create invoices
return
send invoice to Data Consumer
frame
OIDC Authorization Access (Authentication code grant + PKCE)
check invoice
check invoice
retrieve accounting info
return
retrieve smart contract info
retrieve smart contract info
return
return
check invoice
return
return
pay invoice to Data Provider

Figure 8.14 Invoicing process.

Tokenization:

An i3-MARKET crypto token has been created customizing Ethereum ERC-1155 standard. The treasury smart contract contains and maintains the different balances for each data marketplace and user in the i3-MARKET network. When a data consumer obtains tokens from a data marketplace paying fiat money (*exchange in*), both the total balance of data consumer wallet and the specific data marketplace balance will be increased.

During the *payment for data* phase between a data consumer and a data provider, the data consumer can pay the data price in fiat money or in tokens to the data provider. In addition to the data price payment, the data consumer will pay some fees in tokens to the i3M community, the provider data marketplace, and the consumer data marketplace; see Figure 8.15.

A community member (or a DP if we enable data price payments in tokens) will be able to ask fiat money for his token balance from any of the network DM (exchange out) and the amount of token will be transferred from the total balance of community member wallet to the balance of DM wallet; the community member can pay with tokens belonging to the DM with which it is doing the operation or with tokens belonging to other DMs.

Finally, a DM will be able to ask for the clearing of tokens distributed by the other DMs (clearing) – Figure 8.15. For each specific DM balance of the requesting DM wallet, a clearing request will be created, and all the DMs involved will be notified and should pay fiat money to the requesting DM and confirm clearing execution. On clearing confirmation, the tokens will be transferred from requesting DM wallet to clearing DM wallet (requesting DM already approved the transfer during the clearing request) in Figure 8.16.

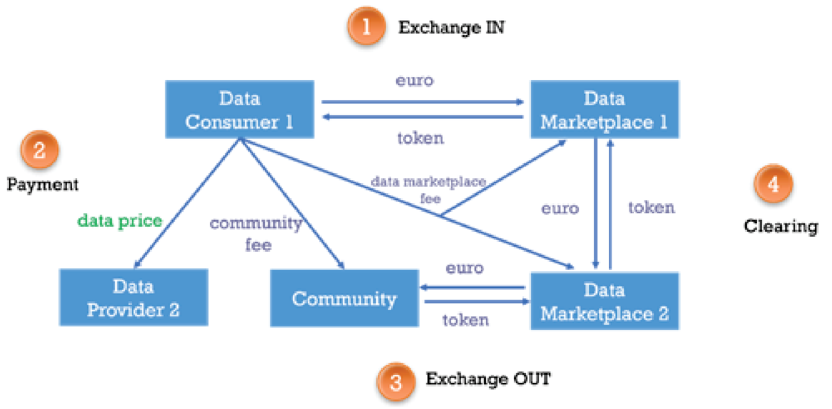


Figure 8.15 Tokenization model.

- **Exchange in:**

In the *exchange in* phase, the user requests a specific amount of tokens from a data marketplace, which, upon receiving the payment in fiat money from the user, returns the tokens in the amount requested as depicted in Figure 8.16.

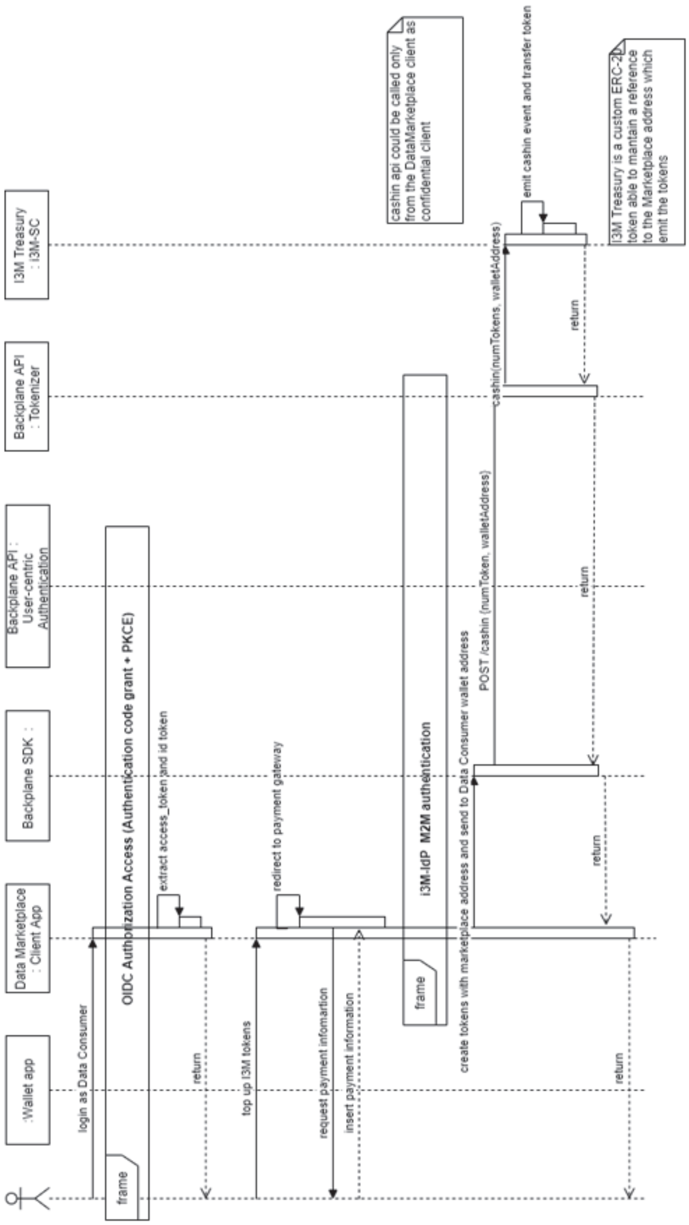

:Wallet app
Data Marketplace : Client App
Backplane SDK :
Backplane API : User-centric Authentication
Backplane API : Tokenizer
I3M Treasury : i3M-SC
frame
login as Data Consumer
OIDC Authorization Access (Authentication code grant + PKCE)
extract access_token and id token
return
top up I3M tokens
redirect to payment gateway
request payment information
insert payment information
frame
i3M-IdP M2M authentication
create tokens with marketplace address and send to Data Consumer wallet address
POST /cashin (numToken, walletAddress)
cashin(numTokens, walletAddress)
emit cashin event and transfer token
return
return
return
return
cashin api could be called only from the DataMarketplace client as confidential client
I3M Treasury is a custom ERC-20 token able to mantain a reference to the Marketplace address which emit the tokens

Figure 8.16 Exchange in process.

As shown in the flow above, when the fiat money payment is received by the data marketplace, it authenticates with the Backplane so that it can call the “exchange in” method of the treasury smart contract; see Figure 8.17. The smart contract mints the tokens directly into the address of the user who requested the tokens; see Figure 8.17.

- **Payment:**

In the *payment* phase, first, the user logs in as data consumer and then can start a data exchange. As a first step, the data consumer makes a first

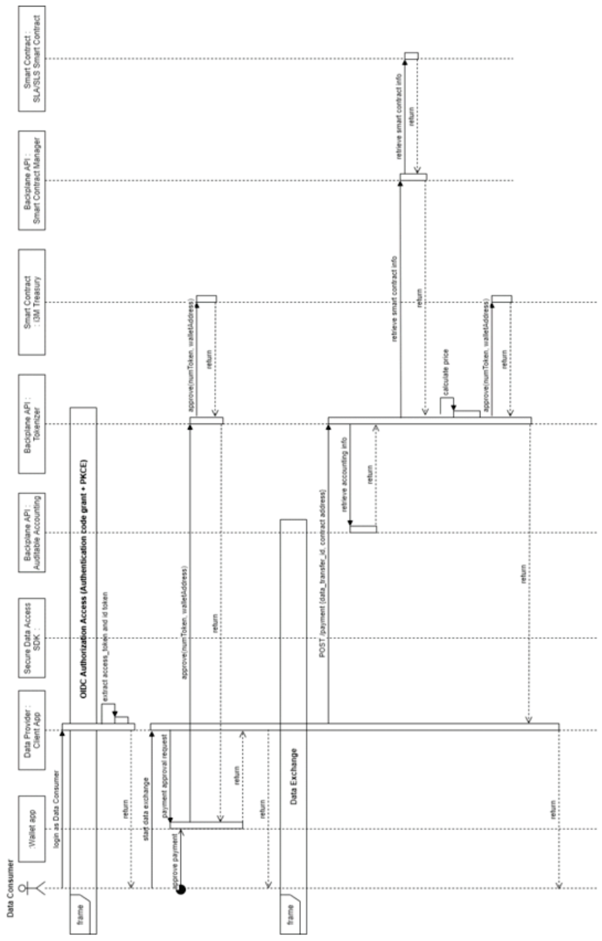

Data Consumer
:Wallet app
Data Provider : Client App
Secure Data Access SDK :
Backplane API : Auditable Accounting
Backplane API : Tokenizer
Smart Contract : i3M Treasury
Backplane API : Smart Contract Manager
Smart Contract : SLA/SLS Smart Contract
frame
OIDC Authorization Access (Authentication code grant + PKCE)
login as Data Consumer
extract access_token and id token
return
start data exchange
payment approval request
approve payment
approve(numToken, walletAddress)
return
Data Exchange
POST /payment (data_transfer_id, contract address)
retrieve accounting info
retrieve smart contract info
calculate price

Figure 8.17 Payment process.

transaction in the blockchain in which he inserts tokens on deposit and, therefore, commits himself to the data provider by offering a guarantee (monetary security). When the data exchange is concluded, the right amount of tokens are taken from the data consumer deposit and moved to the data provider balance; see Figure 8.18.

- **Exchange out:**

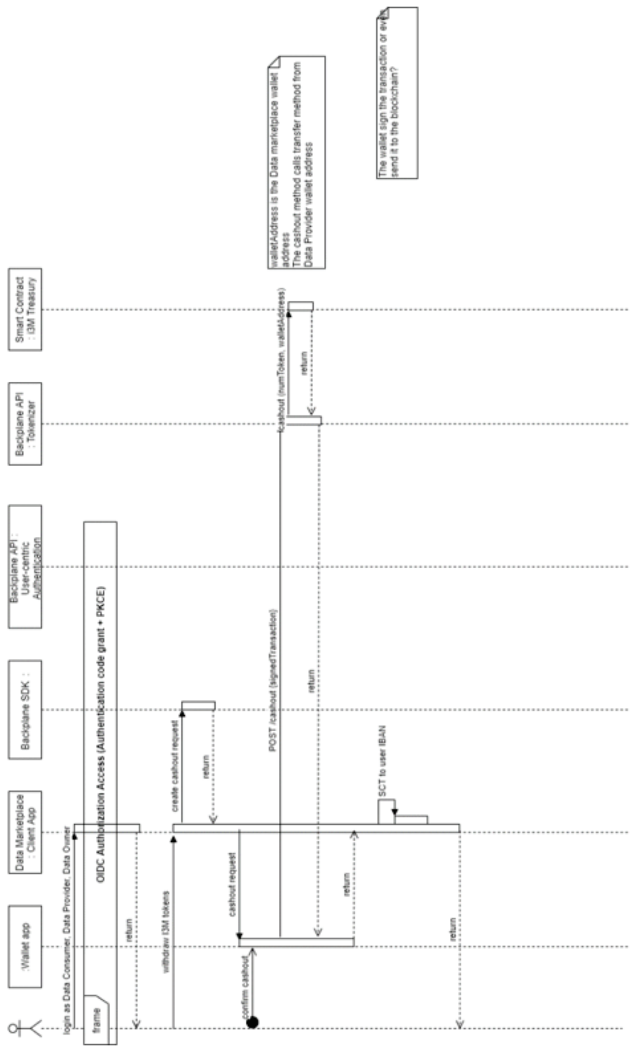

:Wallet app
Data Marketplace : Client App
Backplane SDK :
Backplane API : User-centric Authentication
Backplane API : Tokenizer
Smart Contract : I3M Treasury
frame
OIDC Authorization Access (Authentication code grant + PKCE)
login as Data Consumer, Data Provider, Data Owner
return
withdraw I3M tokens
create cashout request
return
cashout request
confirm cashout
POST /cashout (signedTransaction)
cashout (numToken, walletAddress)
return
return
return
SCT to user IBAN
return
walletAddress is the Data marketplace wallet address
The cashout method calls transfer method from Data Provider wallet address
The wallet sign the transaction or even send it to the blockchain?

Figure 8.18 Exchange out process.

In the *exchange out* phase, the data provider after the login with the data marketplace can start the withdrawal of the i3-MARKET tokens in his balance sheet (Figure 8.19). At first, he confirms the cash-out operation via his wallet application that calls the "exchange out" method in the treasury smart contract. If the operation is successful, the smart contract saves the information regarding the token transaction. At this point, the data marketplace proceeds with the payment in fiat money to the user IBAN and publishes the transaction identifier (TRN) in the blockchain to securely store the transaction in case of future conflicts.

- **Clearing request:**

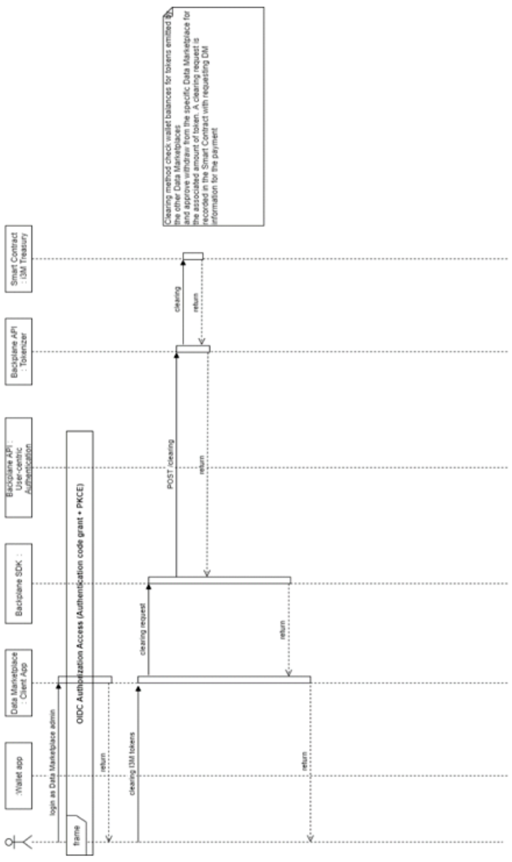

:Wallet app
Data Marketplace : Client App
Backplane SDK :
Backplane API : User-centric Authentication
Backplane API : Tokenizer
Smart Contract : I3M Treasury
frame
OIDC Authorization Access (Authentication code grant + PKCE)
login as Data Marketplace admin
return
clearing I3M tokens
clearing request
POST /clearing
clearing
return
return
return
return
Clearing method check wallet balances for tokens emitted by the other Data Marketplaces and approve withdraw from the specific Data Marketplace for the associated amount of token. A clearing request is recorded in the Smart Contract with requesting DM information for the payment

Figure 8.19 Clearing request process.

During the *clearing* phase, the data marketplace that wants to leave the i3-MARKET network should return the tokens in its balance to the corresponding data marketplace owners – see Figure 8.20. The clearing method should be called for every token type present in the data marketplace balance aside from the token type the data marketplace has created.

- **Clearing execution:**

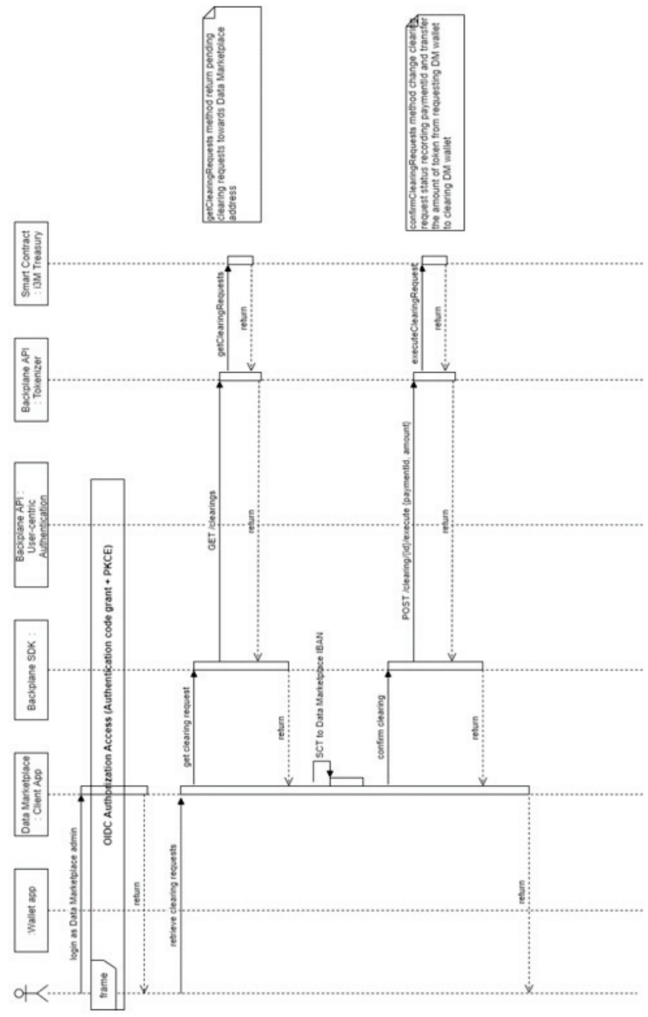

:Wallet app
Data Marketplace : Client App
Backplane SDK :
Backplane API : User-centric Authentication
Backplane API : Tokenizer
Smart Contract : i3M Treasury
getClearingRequests method return pending clearing requests towards Data Marketplace address
confirmClearingRequests method change clearing request status recording paymentId and transfer the amount of token from requesting DM wallet to clearing DM wallet
login as Data Marketplace admin
frame
OIDC Authorization Access (Authentication code grant + PKCE)
return
retrieve clearing requests
get clearing request
GET /clearings
getClearingRequests
return
return
return
SCT to Data Marketplace IBAN
confirm clearing
POST /clearing/{id}/execute {paymentId, amount}
executeClearingRequest
return
return
return
return

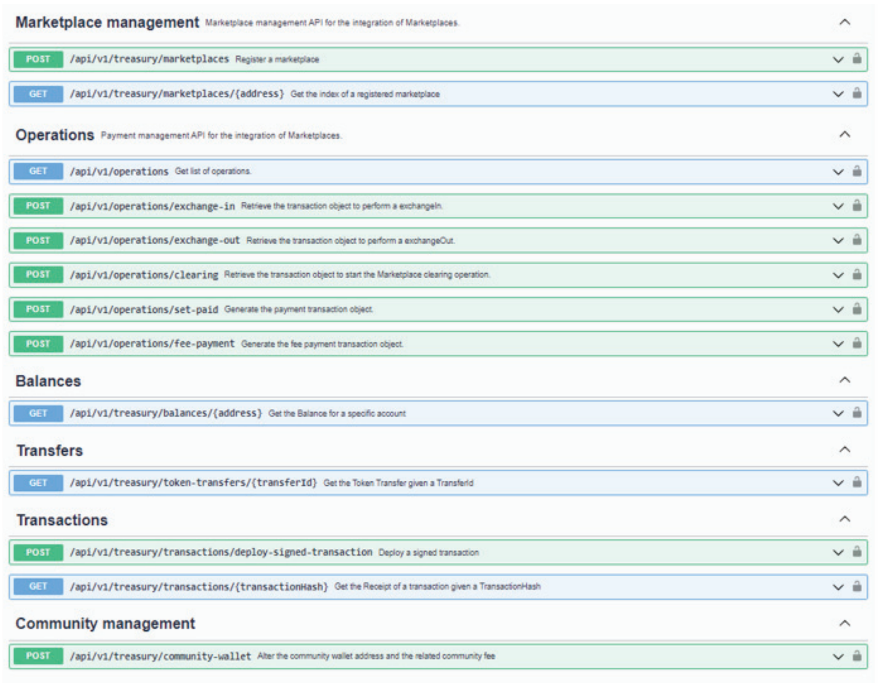
Figure 8.20 Clearing execution process.

In the *clearing execution* phase, the data marketplaces that receive a clearing request from a data marketplace must pay the corresponding value in fiat money of the tokens received.

8.7 Interfaces

The interfaces of the library of the Non-repudiation Protocol for standard payment of the treasury smart contract for tokenization and pricing manager microservices are presented below – see Figures 8.21 and 8.22.

Tokenization:



Marketplace management Marketplace management API for the integration of Marketplaces.

POST /api/v1/treasury/marketplaces Register a marketplace

GET /api/v1/treasury/marketplaces/{address} Get the index of a registered marketplace

Operations Payment management API for the integration of Marketplaces.

GET /api/v1/operations Get list of operations.

POST /api/v1/operations/exchange-in Retrieve the transaction object to perform a exchangeIn.

POST /api/v1/operations/exchange-out Retrieve the transaction object to perform a exchangeOut.

POST /api/v1/operations/clearing Retrieve the transaction object to start the Marketplace clearing operation.

POST /api/v1/operations/set-paid Generate the payment transaction object.

POST /api/v1/operations/fee-payment Generate the fee payment transaction object.

Balances

GET /api/v1/treasury/balances/{address} Get the Balance for a specific account

Transfers

GET /api/v1/treasury/token-transfers/{transferId} Get the Token Transfer given a TransferId

Transactions

POST /api/v1/treasury/transactions/deploy-signed-transaction Deploy a signed transaction

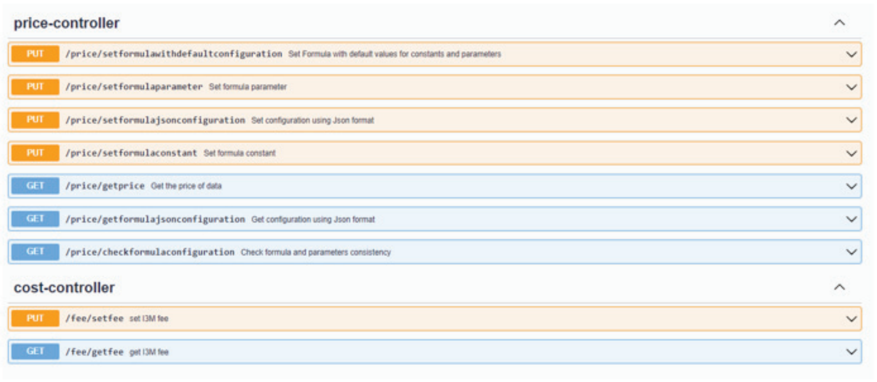
GET /api/v1/treasury/transactions/{transactionHash} Get the Receipt of a transaction given a TransactionHash

Community management

POST /api/v1/treasury/community-wallet Alter the community wallet address and the related community fee

Figure 8.21 Tokenization API.

Pricing manager:



price-controller

PUT /price/setformulawithdefaultconfiguration Set Formula with default values for constants and parameters
PUT /price/setformulaparameter Set formula parameter
PUT /price/setformulajsonconfiguration Set configuration using Json format
PUT /price/setformulaconstant Set formula constant
GET /price/getprice Get the price of data
GET /price/getformulajsonconfiguration Get configuration using Json format
GET /price/checkformulaconfiguration Check formula and parameters consistency

cost-controller

PUT /fee/setfee set I3M fee
GET /fee/getfee get I3M fee

Figure 8.22 Pricing manager API.

8.8 Background Technologies

To implement the solution for tokenization, the ERC-1155 multi-token standard has been chosen and customized.

The ERC-1155 standard was used to implement the i3-MARKET treasury contract, which outlines a smart contract interface that can represent any number of fungible and non-fungible token types. More specifically, the ERC-1155 multi-token standard allows each token ID to represent a new configurable token type, which can have its own metadata, supply, and other attributes.

Adapted to our solution, this token standard has been used to collect the token for the different marketplaces that adhere to the platform, where a new fungible token has to be created for every new marketplace that joins the consortium. This solution allows us to track at any time which token type and therefore which marketplace the tokens in the balance sheet for any participant in the network belong to.

This is particularly important because, for example, during a clearing operation, a marketplace should know to which different marketplaces its tokens in the balance sheet belong; therefore, it can send to each one the right amount of tokens to be converted and returned in fiat money.

It is also important to underline the advantages that the 1155 standard brings because in token standards like ERC-20 and ERC-721, a new separate contract has to be deployed for each token type or collection. This places a lot of redundant bytecodes on the Ethereum blockchain and limits certain functionalities by the nature of separating each token contract into its own permissioned address. With this new design, it is possible to transfer multiple token types at once, saving on transaction costs and removing the need to "approve" individual token contracts separately.