

Tailored IoT & BigData Sandboxes and Testbeds for Smart,
Autonomous and Personalized Services in the European
Finance and Insurance Services Ecosystem



D4.11 – Blockchain Tokenization and Smart Contracts – II

Revision Number	3.0
Task Reference	T4.4
Lead Beneficiary	IBM
Responsible	Fabiana Fournier
Partners	IBM, BOUN, ENG, GFT, and HPE
Deliverable Type	Report (R)
Dissemination Level	Public
Due Date	2021-07-31
Delivered Date	2021-07-30
Internal Reviewers	GFT, CCA
Quality Assurance	INNOV
Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Pierre-Paul Sondag
Programme	HORIZON 2020 - ICT-11-2018



This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 856632

Contributing Partners

Partner Acronym	Role ¹	Author(s) ²
IBM	Lead Beneficiary	Fabiana Fournier, Inna Skarbovsky
BOUN	Contributor	Can Ozturan, Alper Sen, Baran Kılıç
GFT	Internal Reviewer	Ernesto Troiano
CCA	Internal Reviewer	Paul Lefrere
INNOV	Quality Assurance	John Soldatos

Revision History

Version	Date	Partner(s)	Description
0.1	2020-06-15	IBM	ToC Version
0.2	2020-06-20	IBM	Initial contributions to Sections 1 and 2
0.3	2020-06-30	BOUN	Initial contribution to Section 3
0.4	2020-07-05	BOUN	Final contribution to Section 3
0.5	2020-07-10	IBM	Updated Sections 1-3
0.6	2020-07-13	IBM	Section 4 and executive summary
1.0	2020-07-15	IBM	First Version for Internal Review
1.1	2021-07-22	GFT CCA	Internal Review
2.0	2021-07-27	IBM	Version for Quality Assurance
3.0	2020-07-29	IBM	Version for Submission

¹ Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

² Can be left void

Executive Summary

Encouraged by the success of the ERC 20 implementation on top of Hyperledger Fabric reported in the first version of this deliverable, we started to work in parallel in two of the envisioned potential directions devised as future steps: extension to more elaborated standards; and building a privacy preserving federated machine learning framework in which tokens play the role of trading mechanism for insights in a data marketplace.

The document in hand, titled D4.11 “Blockchain Tokenization and Smart Contracts – II” summarizes the work realized within the context of T4.4 “Tokenization and Smart Contracts Finance and Insurance Services” in work package 4 from month 15 to month 22 of the project around the implementation of the ERC 1155, the most advanced Ethereum standard that includes both fungible and non-fungible tokens. The work on evolving the privacy preserving federated machine learning framework, is the focus of D4.14 “Encrypted Data Querying and Personal Data Market – II”.

The purpose of this deliverable is to present the design and implementation work of the ERC 1155 standard on top of Hyperledger Fabric, the blockchain platform selected for the INFINITECH project. It is important to note, that although the deliverable is of type “R” (only Report), we provide a full demonstrator of the token workflows implemented as smart contracts in Fabric along with a recording of the work using an illustrative example.

Digital tokens have started to get traction in the financial and insurance sectors in the last years. Our ultimate goal is that corporates, banks, and FinTech companies will benefit from tokenization capabilities provided by the INFINITECH platform in real use cases. One of the ways to do so, can be through the INFINITECH marketplace. The INFINITECH project is developing and making available through its marketplace various technological assets in the form of BigData, AI applications, and services that will be of interest to financial institutions, government agencies, businesses, and end users. These assets can be represented as tokens on a blockchain and offered to interested parties that can consume or trade them. A standardized token interface like ERC-20 and ERC-1155 means these tokenized assets can be accessible by outside services which support such standards. This will enable tokenized assets to be easily marketed at a global level.

Table of Contents

1	Introduction.....	7
1.1	Objective of the Deliverable.....	7
1.2	Insights from other Tasks and Deliverables.....	8
1.3	Structure.....	8
2	Background.....	9
3	ERC 1155 token standard implementation	11
3.1	Design	11
3.1.1	Workflows/use cases.....	12
3.1.2	Sequence Diagrams	24
3.2	Demonstrator	37
4	Conclusions.....	38
5	Appendix A: Literature.....	39

List of Figures

Figure 1 - Comparison between the UTXO and account models	10
Figure 2 – Get balance of sequence diagram	24
Figure 3 – Get balance of batch sequence diagram	25
Figure 4 – Transfer from sequence diagram	26
Figure 5 – Batch transfer from sequence diagram	27
Figure 6 – Set approval for all sequence diagram	28
Figure 7 – Is approved for all sequence diagram	28
Figure 8 – Mint sequence diagram.....	29
Figure 9 – Mint batch sequence diagram.....	30
Figure 10 – Burn sequence diagram.....	31
Figure 11 – Burn batch sequence diagram.....	32
Figure 12 – Set URI sequence diagram.....	33
Figure 13 – URI sequence diagram.....	34
Figure 14 – Batch transfer from multi recipient sequence diagram	35
Figure 15 – Broadcast token existence sequence diagram	36
Figure 16 – Client account ID sequence diagram	36
Figure 17 – Different types of transfers and their parameters. Note that P1-P5 are client account ids in base64-encoded format.....	37

List of Tables

Table 1 – Standard ERC 1155 functions.....	11
Table 2 – Additional implemented functions	12
Table 3 – Balance Of use case	13
Table 4 - Balance of Batch use case.....	13
Table 5 – Transfer From use case	14

Table 6 – Batch Transfer From use case	15
Table 7 – Set Approval For All use case	16
Table 8 – Is Approved For All use case	17
Table 9 – Mint use case	17
Table 10 – Mint Batch use case	18
Table 11 – Burn use case	19
Table 12 – Burn Batch use case	19
Table 13 - Set URI use case.....	20
Table 14 – URI use case	21
Table 15 – Batch Transfer From Multi Recipient use case	22
Table 16 – Broadcast Token Existence use case.....	23
Table 17 – Client Account ID use case.....	23

Abbreviations/Acronyms

Abbreviation	Definition
API	Application Programming Interface
BC	Blockchain
CA	Certificate Authority
DoA	Description of Action
DLT	Distributed Ledger Technology
ERC	Ethereum Request for Comments
M	Month
MCC	Multiversion Concurrency Control
MS	Milestone
MVCC	MultiVersion Concurrency Control
NFT	Non-fungible token
R	Report
SDK	Software Development Kit
T	Task
URI	Uniform Resource Identifier
UTXO	Unspent Transaction Output
WP	Work Package

1 Introduction

The INFINITECH consortium decided to exploit the Hyperledger Fabric open source enterprise-grade permissioned Distributed Ledger Technology (DLT) platform [1] as the underlying blockchain platform in the project. However, Hyperledger Fabric (simply Fabric) lacks built-in support for tokens. Our first deliverable (D) on tokenization, D4.10 “Blockchain tokenization and smart contracts – I”, submitted at month 14 (M14) of the project, revolved around the implementation of the ERC 20 standard [2], the de-facto standard in many financial and other blockchain applications. However, ERC 20 supports only fungible tokens. In this deliverable we extend the work to include the ERC 1155 [3] standard, a superior token standard for Ethereum based tokens that allows developers to create fungible, non-fungible, and semi-fungible tokens, all through one smart contract in Hyperledger Fabric.

During the first stages of the project we investigated several potential directions for further research. These are described extensively in D4.10. Two of these recognized threads were (refer to D4.10):

1. Extending the work to include additional standards, especially inclusion of non-fungible tokens (NFTs).
2. Leveraging tokens and BC technology as a means of incentivization and crypto currency in a marketplace for insights. The latter includes a framework for federated machine learning with privacy preserving guarantees.

This deliverable describes the work carried out from M15 to M22 of the project around extending Hyperledger Fabric (simply Fabric) to support tokenization (first point) by BOUN and IBM. D4.14 “Encrypted Data Querying and Personal Data Market – II” to be submitted as well at M22 describes the work performed from M15 to M22 of the project by IBM and FBK around point 2 and introduces our devised framework.

We assume the reader is familiar with DLTs and specifically Fabric. Therefore, BC basic terminology and concepts such as peers, organizations, orderers, certificate authority (CA), channels, and chaincodes (smart contracts in Hyperledger Fabric) are known to the reader and out of the scope for this document. For a thorough understanding of how Fabric blockchain technology works, refer to the “Hands-on Blockchain with Hyperledger” [4].

It should be noted that although this deliverable is typed as Report, we also provide an implementation and demo of our work (refer to Section 3). Furthermore, the code, currently available at: <https://gitlab.infinittech-h2020.eu/blockchain/erc1155-tokenization>, will be released to the open source community to enable developers to leverage this work in building their own blockchain applications. The demonstrator can be accessed through the INFINITECH marketplace at: <https://marketplace.infinittech-h2020.eu/infinittech/erc1155-token-smart-contract-for-hyperledger>.

1.1 Objective of the Deliverable

The purpose of this deliverable is to report the outcomes of the work carried out within the context of Task 4.4 (T4.4) “Tokenization and Smart Contracts Finance and Insurance Services” from month 15 (M15) to M22 of the project in relation to the implementation of the ERC 1155 standard. In this second iteration, as explained in the previous paragraph, the work has been carried out in two parallel threads that are the continuation of two of the envisioned potential extensions of the work reported in D4.10 “Blockchain Tokenization and Smart Contracts – I”, i.e., extension to a more mature and extensive tokens standard (ERC 1155) and the collaboration between Task 4.4 (T4.4) – Tokenization and Smart Contracts Finance and Insurance Services and T4.5 – Secure and Encrypted Queries over Blockchain Data towards a trust-preserving framework for federated learning.

Hence, the main objectives of the deliverable at hand are as follows:

- To document main characteristics of the ERC 1155 tokens standard.

- To implement the ERC 1155 standard on top of Hyperledger Fabric, thus enhancing the latter with capabilities to support both fungible and non-fungible tokens.
- To demonstrate the feasibility of the approach by implementing a demonstrator around the work.

It should be noted that, according to the INFINITECH Description of Action (DoA), Task 4.4 lasts until M30, and therefore another (last) version of the deliverable will be released at M30 with deliverable D4.12. Although this deliverable is of type report (“R”), we include the description of the implemented functions as well as the implementation and a demonstrator.

1.2 Insights from other Tasks and Deliverables

Deliverable D4.11 is released in the scope of Work Package 4 (WP4) “Interoperable Data Exchange and Semantic Interoperability” activities, and documents the outcomes of the work performed within the context of T4.4 “Tokenization and Smart Contracts Finance and Insurance Services” on extending the efforts carried out during the first 14 months of the project and reported in the first iteration of the deliverable (D4.10). Therefore, D4.11 heavily relies on D4.10, specifically the background and motivation for tokenization in the financial and insurance sectors, as well as the introduction to the tokens world – key terminology and main concepts. D4.11 expands one of the envisioned future directions in D4.10, i.e., the implementation of a more mature tokens standard that supports both fungible and non-fungible tokens.

In addition, D4.11 is a blockchain-oriented deliverable, hence related to the two previous deliverables around blockchain applications carried out in the scope of the INFINITECH project, i.e., D4.7 “Permissioned Blockchain for Finance and Insurance – I” (submitted at M11) and D4.8 “Permissioned Blockchain for Finance and Insurance – II” (submitted at M19).

1.3 Structure

This document is structured as follows:

- Section 1: introduces the document, describing the context of the outcomes of the work performed within the task and highlighting its relation to other tasks and deliverables of the project;
- Section 2: provides a brief background;
- Section 3: details the implementation of the work; and
- Section 4 summarizes and concludes the document.

2 Background

Just for the sake of completeness, we remind readers very briefly about some of the concepts used throughout this document. For more details refer to D4.10.

Asset tokenization refers to the representation of any asset into its digital form for trading which can later be bought, sold, exchanged or redeemed for any other digital or physical value.

Cryptographic tokens, or ‘tokens’ for short, are programmable digital units of value that are recorded on a distributed ledger protocol such as a blockchain. Representing assets as tokens allows use of the blockchain ledger to establish the unique state and ownership of an item, and the transfer of ownership using a consensus mechanism that is trusted by multiple parties. As long as the ledger is secured, the asset is immutable and cannot be transferred without the owner’s consent. Tokens can represent tangible assets, such as goods moving through a supply chain or a financial asset being traded. Tokens can also represent intangible assets such as loyalty points. Tokens can be fungible (that is, have the same value as other tokens of the same class) or non-fungible (unique, not interchangeable), or a combination of the two. Because tokens cannot be transferred without the consent of the owner, and transactions are validated on a distributed ledger, representing assets as tokens allows reducing the risk and difficulty of transferring assets across multiple parties.

Digitization of assets is a process in which the rights to an asset are converted into a digital token on a blockchain. Ownership rights are transmitted and traded on a digital platform, and the real-world assets on the blockchain are represented by digital tokens.

In D4.10 “Blockchain tokenization and smart contracts – I”, we gave a broad background to the tokenization world. As we have previously stated in D4.10, we selected Ethereum standards due to their popularity and to be compatible with the rest of the partners in the consortium that develop in Ethereum. To provide a common form of tokens in Ethereum, a standard interface referred to as Ethereum Request for Comments (ERC) was introduced. We chose to start with the ERC 20 Token Standard [2] which is the most widely used and most general token standard that provides basic functionality to transfer tokens, as well as allows tokens to be approved so they can be spent by another on-chain third party. It lists six mandatory and three optional functions as well as two events to be implemented by conforming Application Programming Interface (API). In the scope of D4.10 we have implemented the ERC 20 as chaincode (smart contract in Fabric jargon).

However, ERC 20 deals only with fungible tokens (interchangeable tokens), hence, a most natural extension to the work was to extend it to include non-fungible tokens (unique, not interchangeable). ERC 1155 is a standard interface for contracts that manage multiple token types. A single deployed contract may include any combination of fungible tokens, non-fungible tokens or other configurations (e.g. semi-fungible tokens).

For tokens to be useful, they need to be transferable. The transfer of a token on a blockchain is initiated by the owner, creating a transaction. This transaction informs the network about how much tokens are changing hands and who the new owner is. In D4.10 we have also reviewed the two models for bookkeeping of token transactions, namely *UTXO* (unspent transaction output) and the *account model*.

The UTXO model only records transaction receipts. The UTXO model does not incorporate accounts or wallets at the protocol level. The model is based entirely on individual transactions, grouped in blocks. Since there is no concept of accounts or wallets on the protocol level, the “burden” of maintaining a user’s balance is shifted to the client-side. The sum of all the unspent transaction outputs it can control determines the current balance. The account based transaction model represents assets as balances within accounts, similar to bank accounts.

On the other hand, a transaction in the account based model triggers nodes to decrement the balance of the sender’s account and increment the balance of the recipient’s account. In an account based model, there is an account for each participant that holds a balance of tokens. A *mint* transaction creates tokens in an account, while a *transfer* transaction debits the caller’s account and credits another account. The account model keeps track of all balances, as a global state. This state can be understood as a database of all accounts, private key and contract code controlled, and the current balances of the different assets on the network.

While the account model is more intuitive and easier when it comes to enabling smart contracts, adding and subtracting balances makes it easier for developers to create transactions that require state information or involve multiple parties), the UTXO allows for the simpler parallelization of transactions in smart contracts.

Figure 1 illustrates the main difference between the two models. Let’s assume that Bob has \$5 in banknotes and \$5 in coupons. Bob transfers \$7 to Alice and pays a \$0.07 fee to Charlie. In the UTXO model, the \$7 and \$0.07 are the outputs of this transaction, as well as an additional output of \$2.93 to Bob himself in change. In the account model Bob had a total of 10\$ from which he payed \$7.07 and his account balance results in 2.93\$.

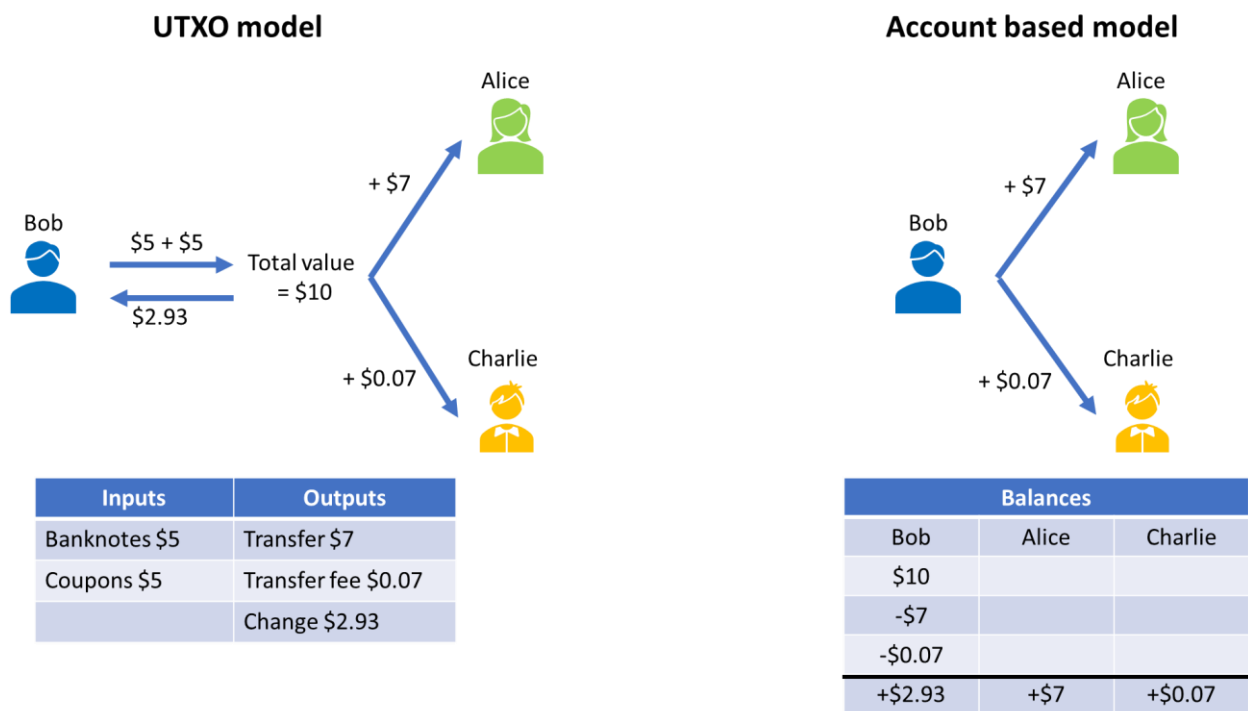


Figure 1 - Comparison between the UTXO and account models

Multiversion concurrency control (MCC or MVCC), is a concurrency control method commonly used by database management systems to provide concurrent access to the database. In blockchain, an MVCC conflict happens when two (or more) transactions attempt to write the same key in the ledger at the same time (key collision).

One of the common solutions to avoid collisions is batching. This technique relies on the fact that a chaincode can update the same key multiple times in the same execution. So, the application level collects some transactions and sends them as one (array or list) to the peer/s. Instead of sending individual transactions to the peer, the Software Development Kit (SDK) would batch transactions and send them to the chaincode and the chaincode is the one responsible for aggregating the updates from the transactions in the batch.

Chaincode must know how to handle the list of transactions and execute all of them at the same time. The result of the simulation will be the latest value from all transactions, and this value will be objective. When the block is committed, the application will take the new batch from the queue and will send it to the peers. So, while the block is committed, the application just collects new transactions and puts them in the queue. By fine tuning the size of the batch, a very high throughput can be achieved.

As will be explained in Section 3, in our ERC 1155 implementation, we applied the UTXO model to avoid collisions in Fabric while batching transactions to achieve maximum throughput.

3 ERC 1155 token standard implementation

After successfully implementing the ERC 20 standard as chaincode in Fabric, the next step was the implementation of the ERC 1155 on top of Fabric which is the most advanced Ethereum standard, due to its capability of allowing mix of multiple fungible and non-fungible token types in a single contract. ERC 1155 generalizes fungible ERC 20 and non-fungible ERC 721 [5] token standards by combining them under a single contract that offers multiple token types with multiple quantities. This section describes the design work and the development of the blockchain chaincodes.

3.1 Design

In addition to the six functions required by the ERC 1155 standard [3] as shown in Table 1, nine additional functions were implemented; these are given in Table 2 and explained below.

In Hyperledger Fabric, the distributed state is maintained as a versioned key-value store. When the same key is going to be updated in the same block, there can be a key conflict (collision) which will lead to transaction failure. This has some performance implications for the special case in which a user wants to issue a number of token transfers to multiple recipients. One cannot simply make multiple updates to the account balance of the user that will go into the same block. This is because such a scenario will update the same key value (the same account balance) in the same block and hence lead to transaction failures. To resolve this, one can wait some time so that the next transfer transaction to the next recipient will go into the next block. This, however, will lead to serious transaction-throughput performance degradation.

In order to solve the key conflict problem, our ERC 1155 implementation has been designed so that it uses the UTXO model that stores spent and unspent amounts of tokens and their owners. An additional *BatchTransferFromMultiRecipient* function (Table 2) was introduced so that if a user wants to make several transfers to multiple recipients, all of these can be batched together and can go into the same block. As a result, high transaction-throughput performance of the Hyperledger Fabric can be maintained.

The standard functions (Table 1) implement querying of token balance, transfer of single/multiple token type(s) with multiple quantities to single/multiple recipient(s).

Table 1 – Standard ERC 1155 functions

Function	Description
BalanceOf(owner, id)	Get the balance of the account “owner” for token “id”
BalanceOfBatch(owners, ids):	Get the balance of multiple account/token pairs
TransferFrom(from, to, id, value)	Transfers the tokens of type “id” of quantity “value” from the account “from” to the account to “to”
BatchTransferFrom(from, to, ids, values)	Transfers multiple tokens of type “id” of quantity “value” from the account “from” to account “to”
SetApprovalForAll(operator, approved)	Enable or disable “approval” for a third party (“operator”) to manage all of the caller's tokens.
IsApprovedForAll(owner, operator):	Queries the approval status of an “operator” for a given “owner”

Additional functions implemented beyond the ones required by the standard are token minting functionality, querying of client id, querying and setting of URI and a batched token transfer to multiple recipients as shown in Table 2. These are general purpose functions that anyone who deploys a token contract may expect to find/use in a token contract.

Mint/Burn – although not required, they are necessary to change the supply of tokens and create new fungible or non-fungible tokens. In a real implementation, they will be implemented unless the supply of the tokens is fixed beforehand. *MintBatch/BurnBatch* is only implemented to complement the *TransferFrom/BatchTransferFrom*.

ClientAccountID is a utility function and is special for Fabric because we do not have wallet addresses in Fabric and users need to know their account ID to transfer tokens.

SetURI and *URI* are optional in ERC 1155 (optional extension).

BatchTransferFromMultiRecipient is required to avoid key conflicts as explained above. This problem does not exist in Ethereum because in Ethereum, the transactions are ordered before they are executed.

BroadcastTokenExistence is also explained in ERC 1155 but it is not required (optional extension). It is only used if a token minter wants to announce the existence of a token without minting it.

Table 2 – Additional implemented functions

Function	Description
Mint(account,id,amount)	Mints and deposits the token of type “id” of quantity “amount” to the account “account”
MintBatch(account, ids, amounts)	Mints and deposits multiple tokens of type “id” of quantity “amount” to the account “account”
Burn(account, id, amount):	Withdraws the token of type “id” of quantity “amount” from the account “account” and burns the token
BurnBatch(account, ids, amounts):	Withdraws multiple tokens of type “id” of quantity “amount” from the account “account” and burns the tokens
BatchTransferFromMultiRecipient(from, to, ids, values):	Transfers multiple tokens of type “ids” of quantity “values” from the account “from” to accounts “to”
ClientAccountID()	Returns the client account id
URI(id)	Returns the URI for token type “id”
SetURI(uri)	Sets the URI
BroadcastTokenExistence(id)	Emits TransferSingle event from “0x0” to “0x0”, with the token creator as “operator”, and a value of 0 for the token of type “id”

Chaincode was developed in the Go language. We utilized the test network that is available at <https://github.com/hyperledger/fabric-samples/tree/main/test-network> to deploy the token smart contract. This network has two organizations and one orderer, one peer per organization, and one certificate authority (CA) per organization. We used Fabric version 2.3.1 and Fabric CA version 1.4.9.

3.1.1 Workflows/use cases

We follow the notation of the use cases and sequence diagrams introduced in D4.7 to specify the use cases and sequence diagrams for the functions developed.

3.1.1.1 Use Case: Balance Of

Returns the current balance for the given user ID and token ID.

Table 3 – Balance Of use case

Stakeholders involved	Anyone with read access.
Pre-conditions	The requested account exists, the requesting party is a legal end-user in the network and has read access.
Post-conditions	The invoker receives a reply of the balance of the requested account for given token ID.
Data Attributes	Account address, token ID.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the requester are determined. 2. The existence of the requested account is determined. 3. The current balance of the requested account for the requested token ID is returned.
Pass Metrics	Requested information is returned to the invoker.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The invoker has no access. • The requested account does not exist.

3.1.1.2 Use Case: Balance of Batch

Returns the current balance for the given user IDs and token IDs.

Table 4 - Balance of Batch use case

Stakeholders involved	Anyone with read access
Pre-conditions	The requested accounts exist, the requesting party is a legal end-user in the network and has read access.
Post-conditions	The invoker receives a reply of the balances of the requested accounts for the given token IDs
Data Attributes	Account addresses, token IDs

Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the requester are determined 2. The existence of the requested accounts are determined 3. The current balance of the requested accounts for the requested token IDs are returned
Pass Metrics	Requested information is returned to the invoker
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The invoker has no access. • The requested accounts do not exist.

3.1.1.3 Use Case: Transfer From

Transfers the specified amount of tokens for a given token ID from the sender’s account to the recipient’s account, adjusts the new balance of the sender’s account, the new balance of the recipient’s account.

Table 5 – Transfer From use case

Stakeholders involved	Invoker (operator), sender (from), recipient (to).
Pre-conditions	<ul style="list-style-type: none"> • The sender’s account exists, and has the sufficient amount of tokens. • The invoker is either token owner (sender) or is approved to transfer for the given token ID and sender account. • The invoker party is a legal end-user in the network and has read access, so it is approved to transfer tokens from owner’s account.
Post-conditions	The sender’s balance and the recipient’s balance are updated with the new amount of tokens, the transaction is written on the chain.
Data Attributes	Sender’s account address, recipient’s address, token ID, amount of tokens to transfer.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker (spender) account are determined. 2. The existence of the sender’s account is determined. 3. The existence of the recipient’s account is determined. 4. Enough balance of tokens in the sender’s account is verified 5. The invoker’s permission to transfer is determined. 6. The new balances for the sender’s account and recipient’s accounts are calculated and updated on the ledger.

Pass Metrics	<ol style="list-style-type: none"> 1. The balances in the world state are updated. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The sender’s account does not exist. • The recipient’s account does not exist. • The sender’s account balance does not have enough tokens. • The invoker has no access or was not permitted by sender (using <i>SetApprovalForAll</i> workflow) to transfer tokens on their behalf.

3.1.1.4 Use Case: Batch Transfer From

Transfers the specified amount of tokens for the given token IDs from the sender’s account to the recipient’s account, adjusts the new balance of the sender’s account, the new balance of the recipient’s account.

Table 6 – Batch Transfer From use case

Stakeholders involved	Invoker (operator), sender (from), recipient (to)
Pre-conditions	<ul style="list-style-type: none"> • The sender’s account exists, and has the sufficient amount of tokens. • The invoker is either token owner (sender) or is approved to transfer for the given token IDs and sender account. • The invoker party is a legal end-user in the network and has read access, it is approved to transfer tokens from owner’s account.
Post-conditions	The sender’s balance and the recipient’s balance are updated with the new amount of tokens, the transaction is written on the chain.
Data Attributes	Sender’s account address, recipient’s address, token IDs, amounts of tokens to transfer.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker (spender) account are determined. 2. The existence of the sender’s account is determined. 3. The existence of the recipient’s account is determined. 4. Enough balance of tokens in the sender’s account is verified 5. The invoker’s permission to transfer is determined. 6. The new balances for the sender’s account and recipient’s accounts are calculated and updated on the ledger.

Pass Metrics	<ol style="list-style-type: none"> 1. The balances in the world state are updated. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The sender’s account does not exist. • The recipient’s account does not exist. • The sender’s account balance does not have enough tokens. • The invoker has no access or was not permitted by sender (using <i>SetApprovalForAll</i> workflow) to transfer tokens on their behalf.

3.1.1.5 Use Case: Set Approval For All

Enables or disables approval for a third party (operator) to manage all of the invoker's tokens.

Table 7 – Set Approval For All use case

Stakeholders involved	Invoker, approved account (operator).
Pre-conditions	<ul style="list-style-type: none"> • The invoker’s account exists.
Post-conditions	The operator account is either approved or disapproved to manage all of the invoker's tokens.
Data Attributes	Operator account address, approval or disapproval.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker account are determined. 2. The operator account is set as approved or disapproved.
Pass Metrics	<ol style="list-style-type: none"> 1. The approval status for the operator address is set. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The invoker’s account does not exist

3.1.1.6 Use Case: Is Approved For All

Queries the approval status of an operator for a given owner.

Table 8 – Is Approved For All use case

Stakeholders involved	Invoker, owner account, operator account
Pre-conditions	The invoker's account exists.
Post-conditions	The invoker receives a reply of approval status for an operator for a given owner.
Data Attributes	Owner account address, operator account address.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker account are determined. 2. Approval status is returned.
Pass Metrics	Requested information is returned to the invoker.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The invoker's account does not exist.

3.1.1.7 Use case: Mint

Mints and transfers tokens to a given address for a given token ID.

Table 9 – Mint use case

Stakeholders involved	An account with minter role.
Pre-conditions	<ul style="list-style-type: none"> • The invoker has minter role. • Mint amount is positive.
Post-conditions	The recipient's balance is updated with the new amount of token, the transaction is written on the chain.
Data Attributes	Recipient account address, token ID, token amount.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker account are determined. 2. Non-negative amount of mint tokens is verified. 3. The recipients's balance is updated with the amount of minted tokens for the given token ID.

Pass Metrics	<ol style="list-style-type: none"> 1. The balances in the world state are updated. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The invoker’s account does not exist. • The invoker account does not belong to organization allowed to mint. • The mint amount specified in the function arguments is negative.

3.1.1.8 Use case: Mint Batch

Mints and transfers tokens to a given address for a given token IDs.

Table 10 – Mint Batch use case

Stakeholders involved	An account with minter role.
Pre-conditions	<ul style="list-style-type: none"> • The invoker has minter role. • Mint amount is positive.
Post-conditions	The recipient’s balance is updated with the new amount of token, the transaction is written on the chain.
Data Attributes	Recipient account address, token IDs, token amounts.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker account are determined. 2. Non-negative amount of mint tokens is verified. 3. The recipient’s balance is updated with the amount of minted tokens for the given token IDs.
Pass Metrics	<ol style="list-style-type: none"> 1. The balances in the world state are updated. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The invoker’s account does not exist. • The invoker account does not belong to organization allowed to mint. • The mint amount specified in the function arguments is negative.

3.1.1.9 Use case: Burn

Withdraws tokens from a given address for a given token ID and burns the tokens.

Table 11 – Burn use case

Stakeholders involved	An account with burner role.
Pre-conditions	<ul style="list-style-type: none"> ● The invoker has burner role. ● Burn amount is positive. ● The account whose tokens are withdrawn has enough tokens to burn.
Post-conditions	The account balance is updated with the new amount of tokens, the transaction is written on the chain.
Data Attributes	Account address, token ID, token amount.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker account are determined. 2. Non-negative amount of burned tokens is verified. 3. The balance of account is checked for sufficient tokens. 4. The account balance is updated with the amount of burned tokens for the given token ID.
Pass Metrics	<ol style="list-style-type: none"> 1. The balances in the world state are updated. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> ● The invoker’s account does not exist. ● The invoker account does not belong to the organization allowed to burn. ● The burn amount specified in the function arguments is negative.

3.1.1.10 Use case: Burn Batch

Withdraws tokens from a given address for the given token IDs and burns the tokens

Table 12 – Burn Batch use case

Stakeholders involved	An account with burner role.
Pre-conditions	<ul style="list-style-type: none"> ● The invoker has burner role. ● Burn amounts are positive.

	<ul style="list-style-type: none"> The account whose tokens are withdrawn has enough tokens to burn.
Post-conditions	The account balance is updated with the new amount of tokens, the transaction is written on the chain.
Data Attributes	Account address, token IDs, token amounts.
Normal Flow	<ol style="list-style-type: none"> The authentication and access roles for the invoker account are determined. Non-negative amount of burned tokens is verified. The balance of account is checked for sufficient tokens. The account balance is updated with the amount of burned tokens for the given token ID.
Pass Metrics	<ol style="list-style-type: none"> The balances in the world state are updated. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> The invoker’s account does not exist. The invoker account does not belong to the organization allowed to burn. The burn amounts specified in the function arguments are negative.

3.1.1.11 Use case: Set URI

Sets the URI for tokens.

Table 13 - Set URI use case

Stakeholders involved	An account with minter/burner role.
Pre-conditions	<ul style="list-style-type: none"> The invoker has minter/burner role.
Post-conditions	The URI is updated with the given URI, the transaction is written on the chain.
Data Attributes	New URI.

Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker account are determined. 2. New URI is set.
Pass Metrics	<ol style="list-style-type: none"> 1. The URI in the world state are updated. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The invoker’s account does not exist, • The invoker’s account does not have the required role.

3.1.1.12 Use case: URI

Queries the URI for tokens.

Table 14 – URI use case

Stakeholders involved	Anyone with read access
Pre-conditions	<ul style="list-style-type: none"> • The invoker’s account exists.
Post-conditions	The invoker receives a reply of URI.
Data Attributes	Token ID
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker account are determined. 2. URI is returned.
Pass Metrics	Requested information is returned to the invoker.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • No URI is previously set, • The invoker’s account does not exist.

3.1.1.13 Use Case: Batch Transfer From Multi Recipient

Transfers the specified amount of tokens for the given token IDs from the sender’s account to the recipient’s accounts, adjusts the new balance of the sender’s account, the new balance of the recipient’s account.

Table 15 – Batch Transfer From Multi Recipient use case

Stakeholders involved	Invoker (operator), sender (from), recipients (tos)
Pre-conditions	<ul style="list-style-type: none"> • The sender’s account exists, and has the sufficient amount of tokens. • The invoker is either the token owner (sender) or is approved to transfer for the given token IDs and sender account. • The invoker party is a legal end-user in the network and has read access, and is approved to transfer tokens from owner’s account.
Post-conditions	The sender’s balance and the recipient’s balance are updated with the new amount of tokens, the transaction is written on the chain.
Data Attributes	Sender’s account address, recipient’s addresses, token IDs, amounts of tokens to transfer.
Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the invoker (spender) account are determined. 2. The existence of the sender’s account is determined. 3. The existence of the recipients’ accounts are determined. 4. Enough balance of tokens in the sender’s account is verified 5. The invoker’s permission to transfer is determined. 6. The new balances for the sender’s account and each of the recipients’ accounts are calculated and updated on the ledger.
Pass Metrics	<ol style="list-style-type: none"> 1. The balances in the world state are updated. 2. The transaction is available on the chain.
Fail Metrics	<p>Requested information cannot be returned if:</p> <ul style="list-style-type: none"> • The sender’s account does not exist, • A recipient’s account does not exist, • The sender’s account balance does not have enough tokens, • The invoker has no access or was not permitted by sender (using <i>SetApprovalForAll</i> workflow) to transfer tokens on their behalf.

3.1.1.14 Use Case: Broadcast Token Existence

Emits TransferSingle event from “0x0” to “0x0”, with the token creator as “operator”, and a value of 0 for a given token ID. A minter account can use this function to announce a new token without minting the token.

The existing tokens do not need to be announced since “Mint”, “MintBatch” functions already create Transfer events. Third parties can detect the existence of this new token by listening to the Transfer events.

Table 16 – Broadcast Token Existence use case

Stakeholders involved	An account with minter/burner role.
Pre-conditions	<ul style="list-style-type: none"> The invoker has minter/burner role.
Post-conditions	TransferSingle event is emitted.
Data Attributes	Token ID.
Normal Flow	<ol style="list-style-type: none"> The authentication and access roles for the invoker account are determined. TransferSingle event with from “0x0” to “0x0”, with the token creator as “operator”, and a value of 0 is emitted for a given token ID.
Pass Metrics	TransferSingle event is emitted.
Fail Metrics	Requested information cannot be returned if: <ul style="list-style-type: none"> The invoker has no minter/burner role.

3.1.1.15 Use Case: Client Account ID

Returns the ID of the account. The client ID is simply a base64-encoded concatenation of the issuer and subject from the client identity's enrolment certificate.

Table 17 – Client Account ID use case

Stakeholders involved	Account owner
Pre-conditions	The owner account exists, the requesting party is a legal end-user in the network and has read access
Post-conditions	The invoker receives a reply of client ID
Data Attributes	None

Normal Flow	<ol style="list-style-type: none"> 1. The authentication and access roles for the requester are determined. 2. The existence of the owner account is determined. 3. Client ID is returned.
Pass Metrics	Requested information is returned to the invoker
Fail Metrics	Requested information cannot be returned if: <ul style="list-style-type: none"> • The owner account does not exist.

3.1.2 Sequence Diagrams

In the previous section all the relevant use cases of the designed blockchain application were documented. Henceforth, we present the sequence diagram for each use case, depicting the interactions between the stakeholders and the components of the designed blockchain solution, as well as the interactions between the various components. As in the previous section, we follow the notation of the sequence diagrams introduced in D4.7.

3.1.2.1 Get balance of

In the Balance Of use case, an authorized account in the blockchain network queries the balance of a specified account for a token ID. Upon the authentication of the user and their role as a reader, the tokenization client interacts with blockchain components in order to check the balance of the specified account for a given token ID by querying and reading the query result upon decrypting them (Figure 2).

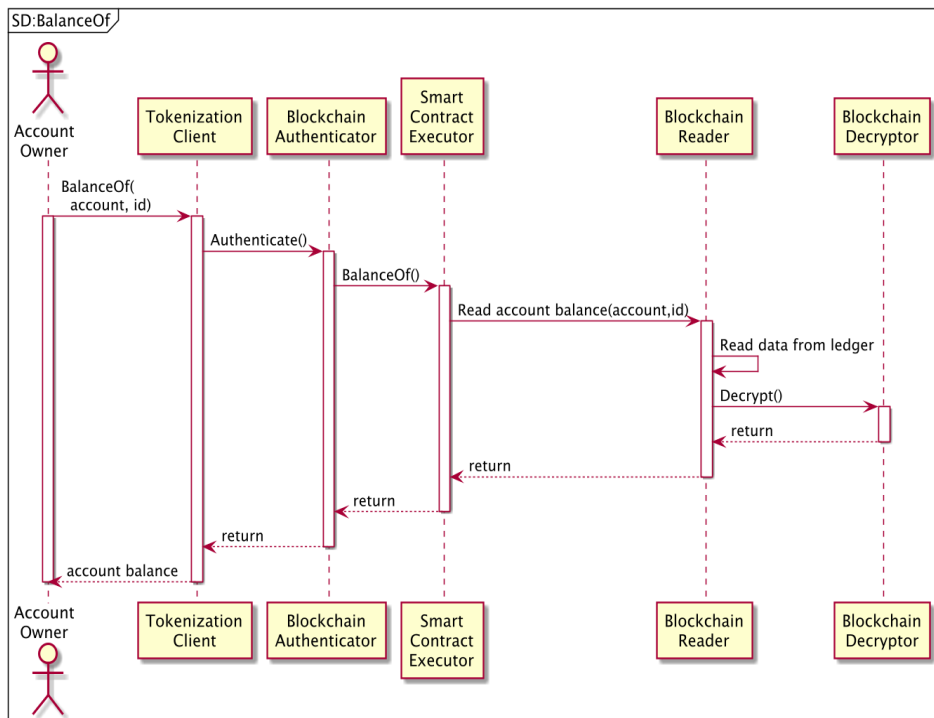


Figure 2 – Get balance of sequence diagram

3.1.2.2 Get balance of batch

In Balance Of Batch use case, an authorized account in the blockchain network queries the balance of a specified account for the given token IDs. Upon the authentication of the user and their role as a reader, the tokenization client interacts with blockchain components in order to check the balance of the specified account for the given token IDs by querying and reading the query result upon decrypting them (Figure 3).

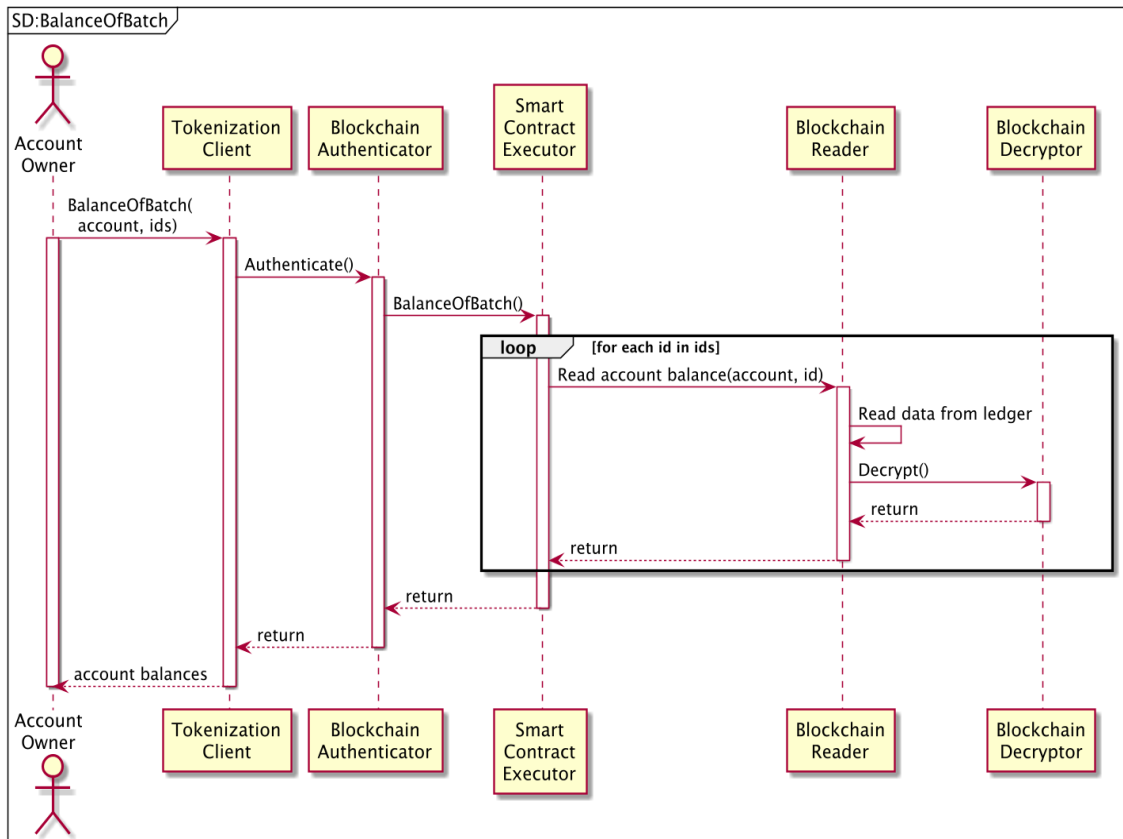


Figure 3 – Get balance of batch sequence diagram

3.1.2.3 Transfer from

In the Transfer From use case, the transfer of tokens of a token ID from one account to another is handled. The tokenization client interacts with the blockchain components in order to retrieve the accounts balance from the ledger, decrypt the query result and update the balance of both accounts, decreasing the amount of tokens in the origin account and increasing the amount of tokens in the recipient account through a new transaction in the ledger (Figure 4).

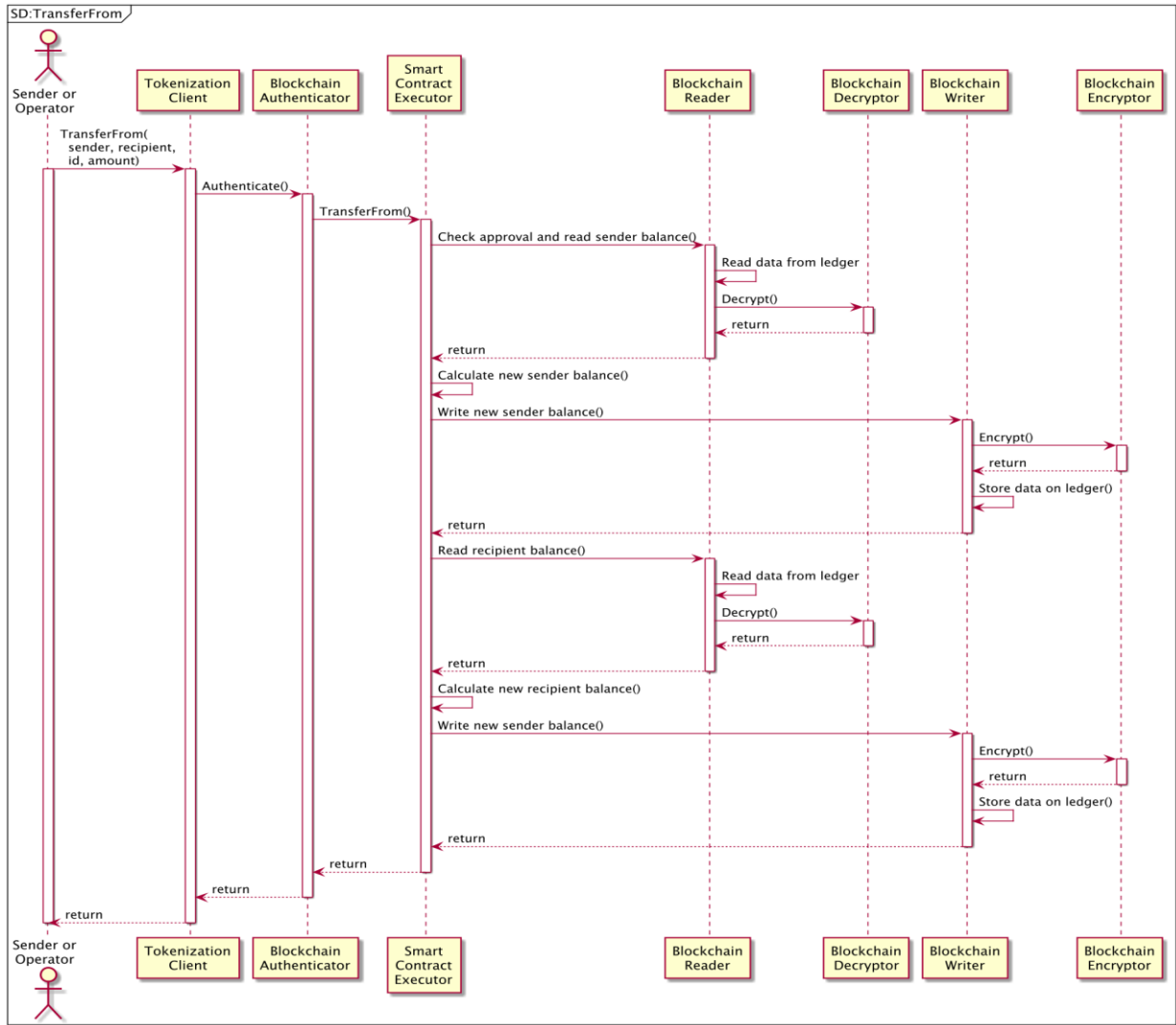


Figure 4 – Transfer from sequence diagram

3.1.2.4 Batch transfer from

In the Batch Transfer From use case, the transfer of tokens of multiple token IDs from one account to another is handled. The tokenization client interacts with the blockchain components in order to retrieve the accounts balance from the ledger, decrypt the query result and update the balance of both accounts, decreasing the amount of tokens in the origin account and increasing the amount of tokens in the recipient account through a new transaction in the ledger (Figure 4).

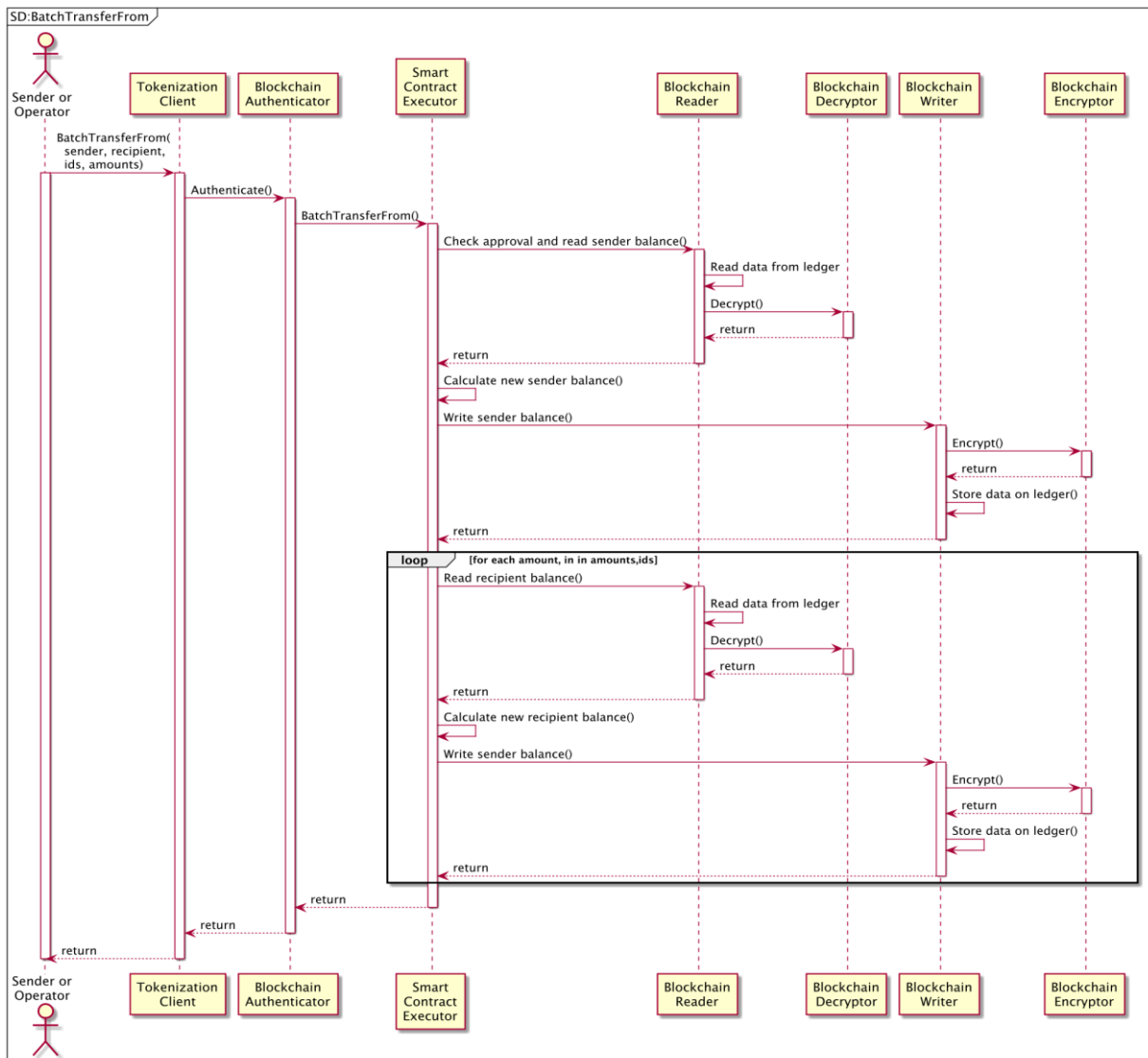


Figure 5 – Batch transfer from sequence diagram

3.1.2.5 Set approval for all

In the Set Approval For All use case, an account approves another account (operator) to manage all of the caller’s tokens. the transfer of tokens of multiple token IDs from one account to another is handled. The tokenization client interacts with the blockchain components in order to enable or disable the approval of the operator account (Figure 6).

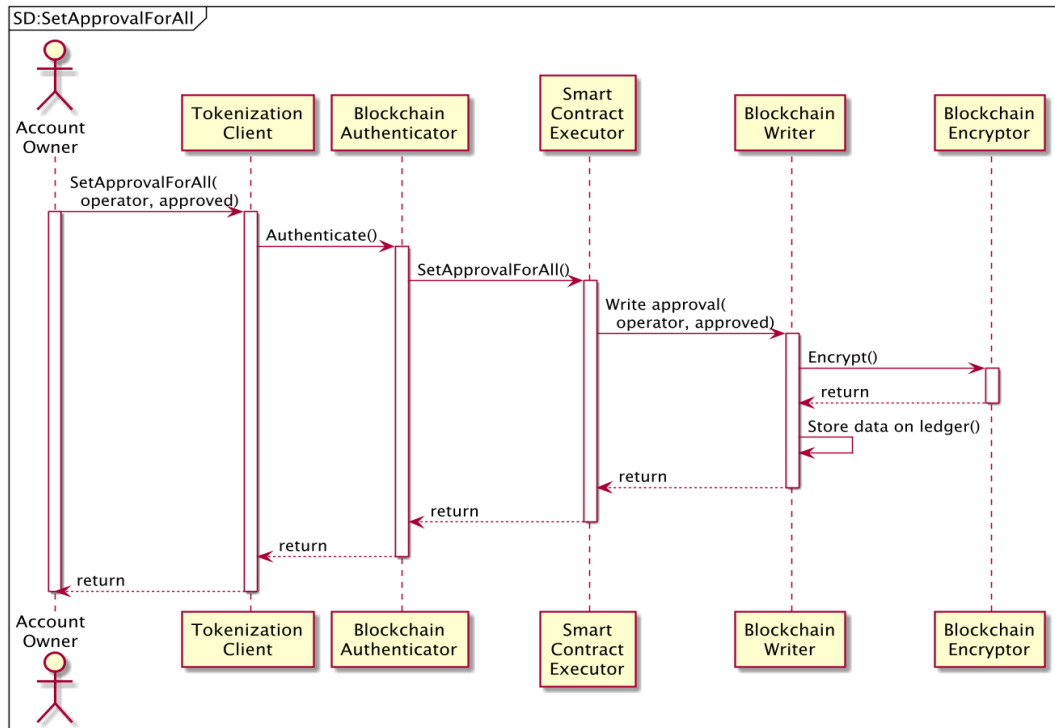


Figure 6 – Set approval for all sequence diagram

3.1.2.6 Is approved for all

In the Is Approved For All use case, an authorized account on the blockchain network queries whether an account is approved to handle another account’s (operator) tokens. Upon the authentication of the user and their role as a reader, the tokenization client interacts with blockchain components in order to check the approval status by querying and reading the query results upon decrypting them (Figure 6).

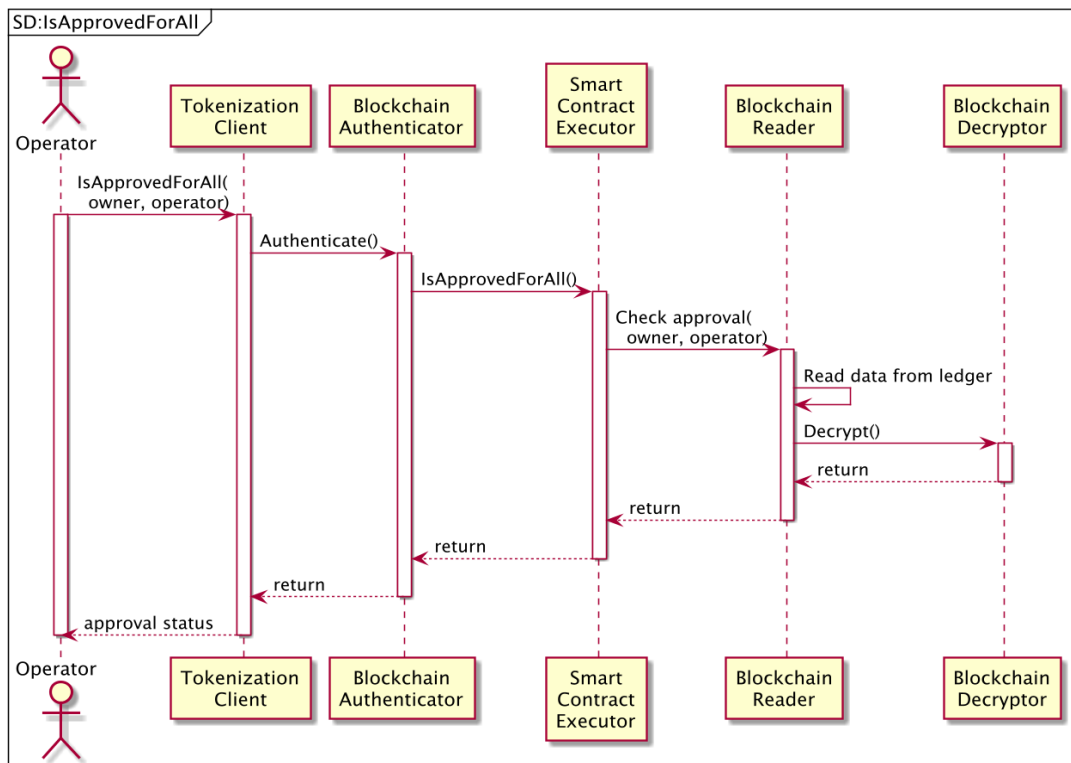


Figure 7 – Is approved for all sequence diagram

3.1.2.7 Mint

In the Mint use case, minting of new tokens for a given token ID by an account with the “Minter” role is performed. The tokenization client interacts with the blockchain components, which in turn, verify that the invoking account has a Minter role and check the balance of the account that will receive the minted tokens. The balance of the account is increased through a new transaction in the ledger (Figure 8).

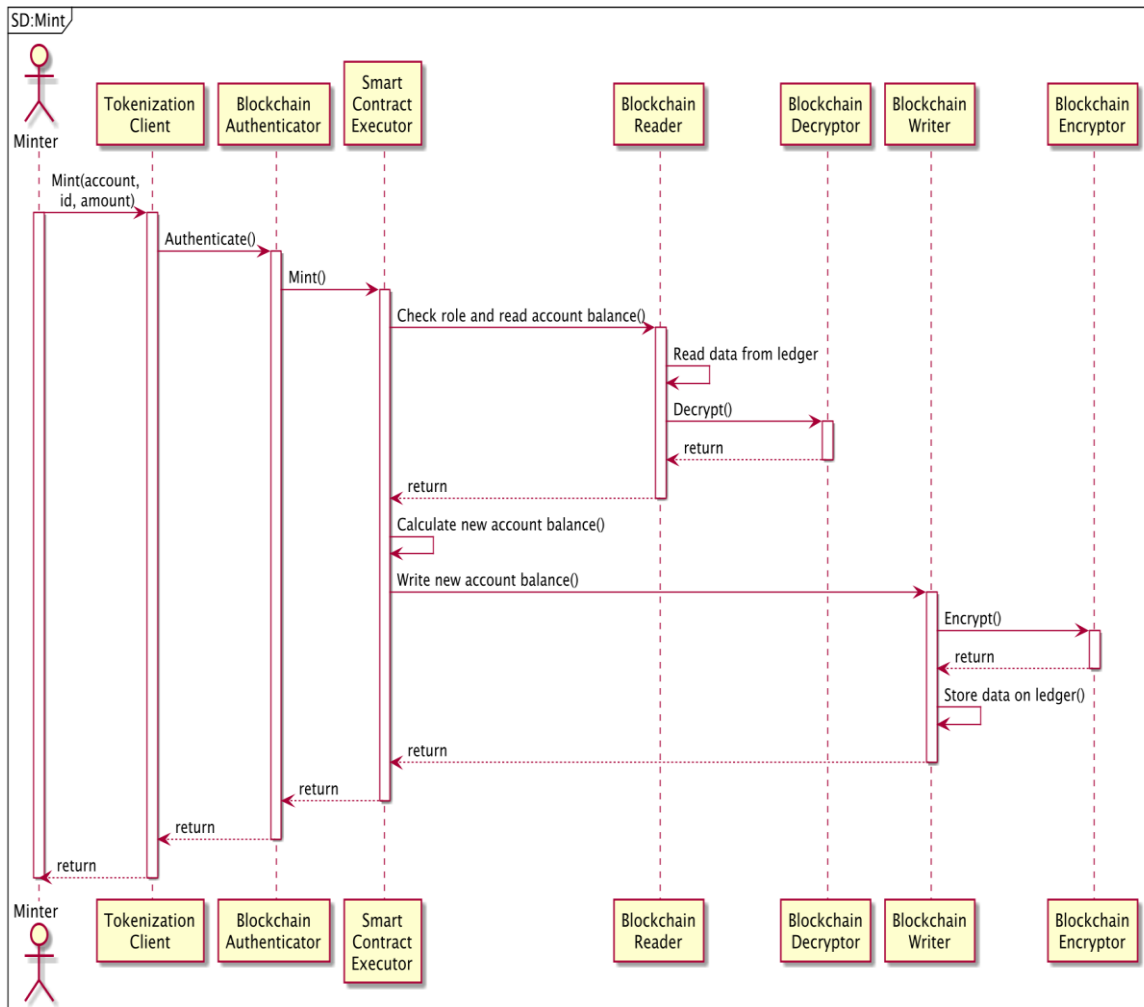


Figure 8 – Mint sequence diagram

3.1.2.8 Mint batch

In the Mint Batched use case, minting of new tokens for the given token IDs by an account with the “Minter” role is performed. The tokenization client interacts with the blockchain components, which in turn, verify that the invoking account has a Minter role and check the balance of the account that will receive the minted tokens. The balance of the account is increased through a new transaction in the ledger (Figure 9).

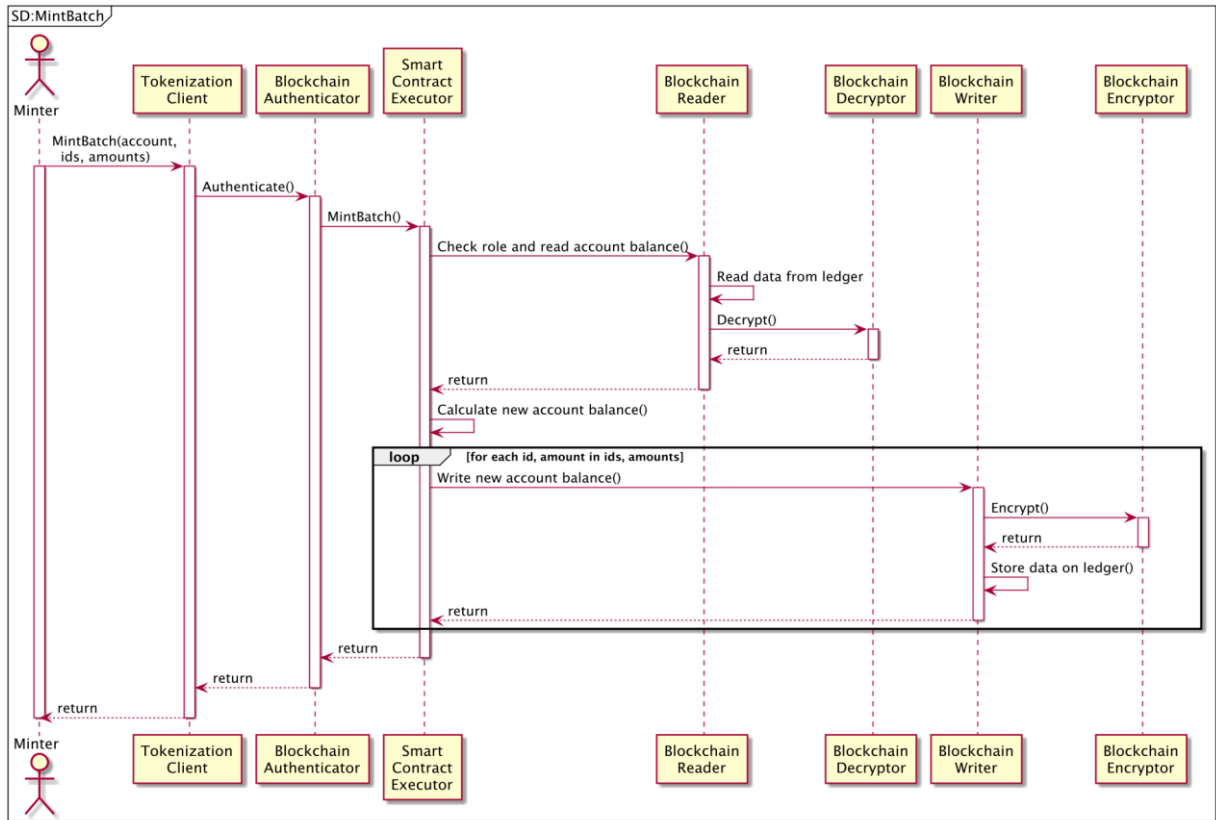


Figure 9 – Mint batch sequence diagram

3.1.2.9 Burn

In the Burn use case, burning of tokens for a given token ID by an account with the “Burner” role is performed. The tokenization client interacts with the blockchain components, which in turn, verify that the invoking account has a Burner role and check the balance of the account whose tokens will be burned. The balance of the account is decreased through a new transaction in the ledger (Figure 10).

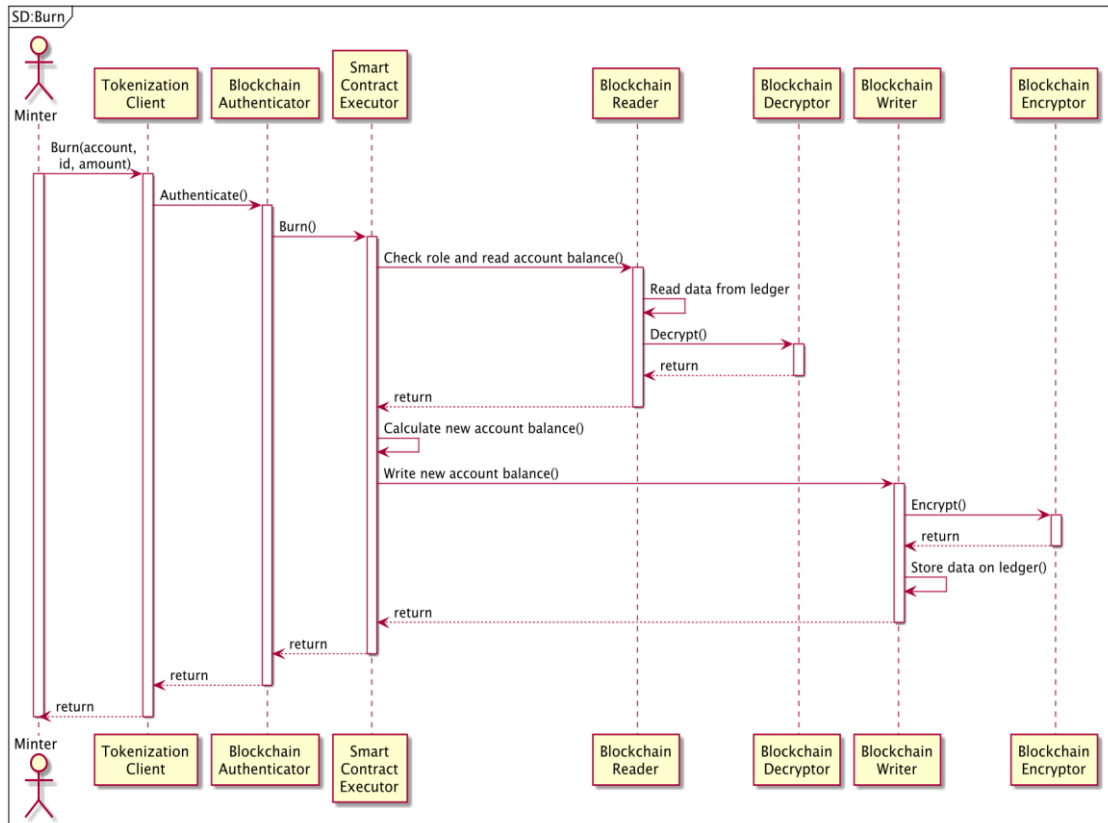


Figure 10 – Burn sequence diagram

3.1.2.10 Burn batch

In the Burn Batch use case, burning of tokens for the given token IDs by an account with the “Burner” role is performed. The tokenization client interacts with the blockchain components, which in turn, verify that the invoking account has a Burner role and check the balance of the account whose tokens will be burned. The balance of the account is decreased through a new transaction in the ledger (Figure 11).

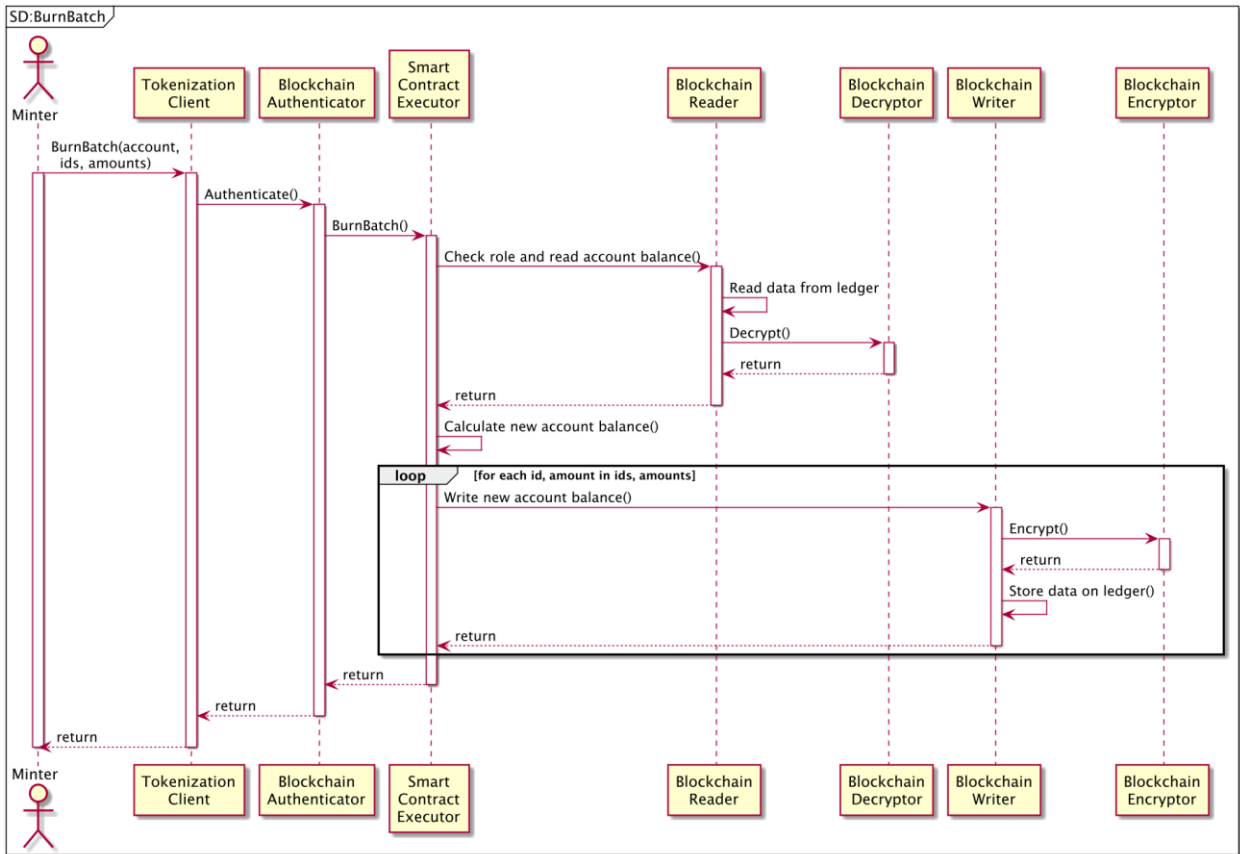


Figure 11 – Burn batch sequence diagram

3.1.2.11 Set URI

In the Set URI use case, the minter account sets a URI for tokens. The tokenization client interacts with the blockchain components and verifies that the invoking account has a Minter role and sets the URI through a new transaction in the ledger (Figure 12).

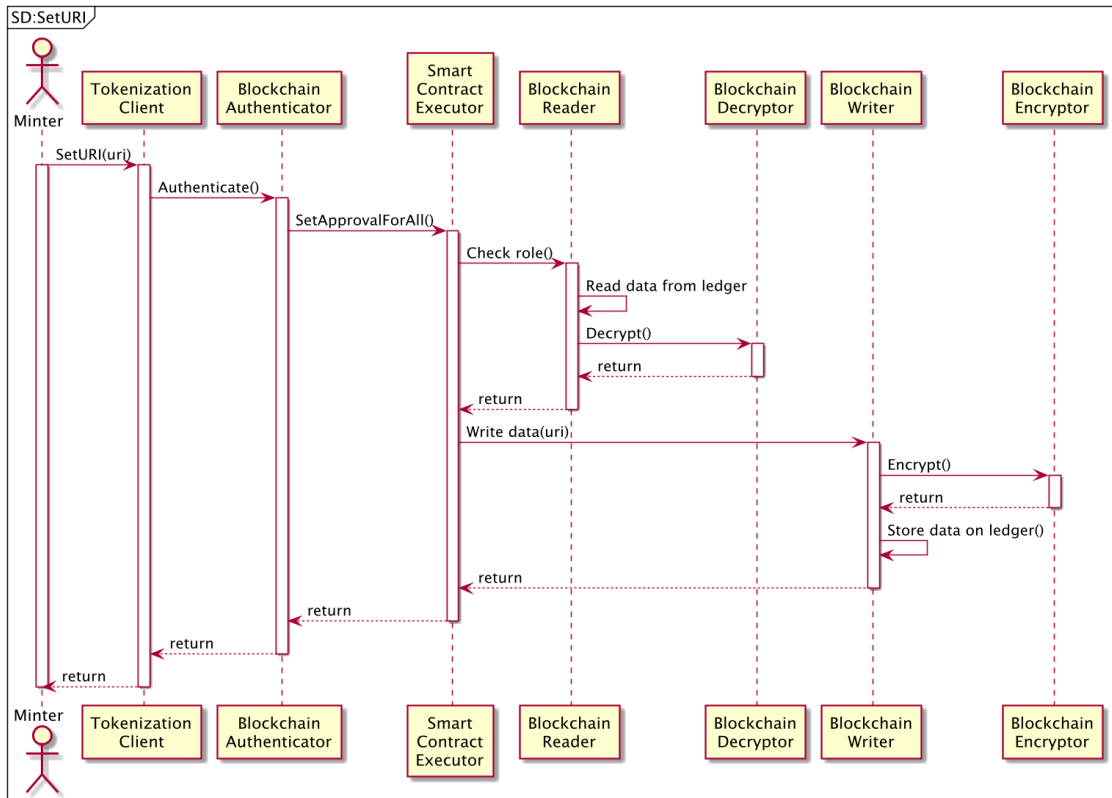


Figure 12 – Set URI sequence diagram

3.1.2.12 URI

In the URI use case, an authorized account in the blockchain network queries the URI for tokens. Upon the authentication of the user and their role as a reader, the tokenization client interacts with blockchain components in order to check the URI by querying and reading the query result upon decrypting them (Figure 13).

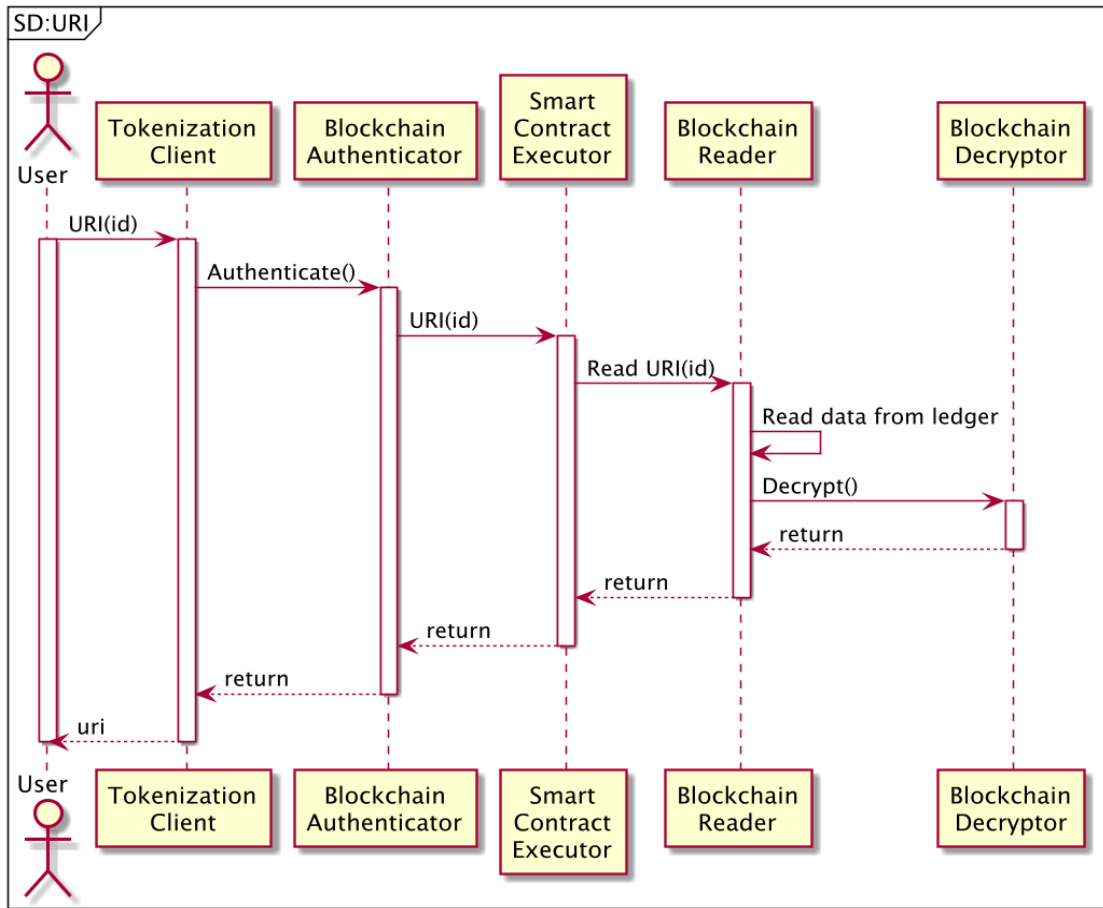


Figure 13 – URI sequence diagram

3.1.2.13 Batch transfer from multi recipient

In Batch Transfer From Multi Recipient use case, the transfer of tokens of multiple token IDs from one account to other accounts is handled. The tokenization client interacts with the blockchain components in order to retrieve the accounts balance from the ledger, decrypt the query result and update the balance of all accounts, decreasing the amount of tokens in the origin account and increasing the amount of tokens in the recipient accounts through a new transaction in the ledger (Figure 14).

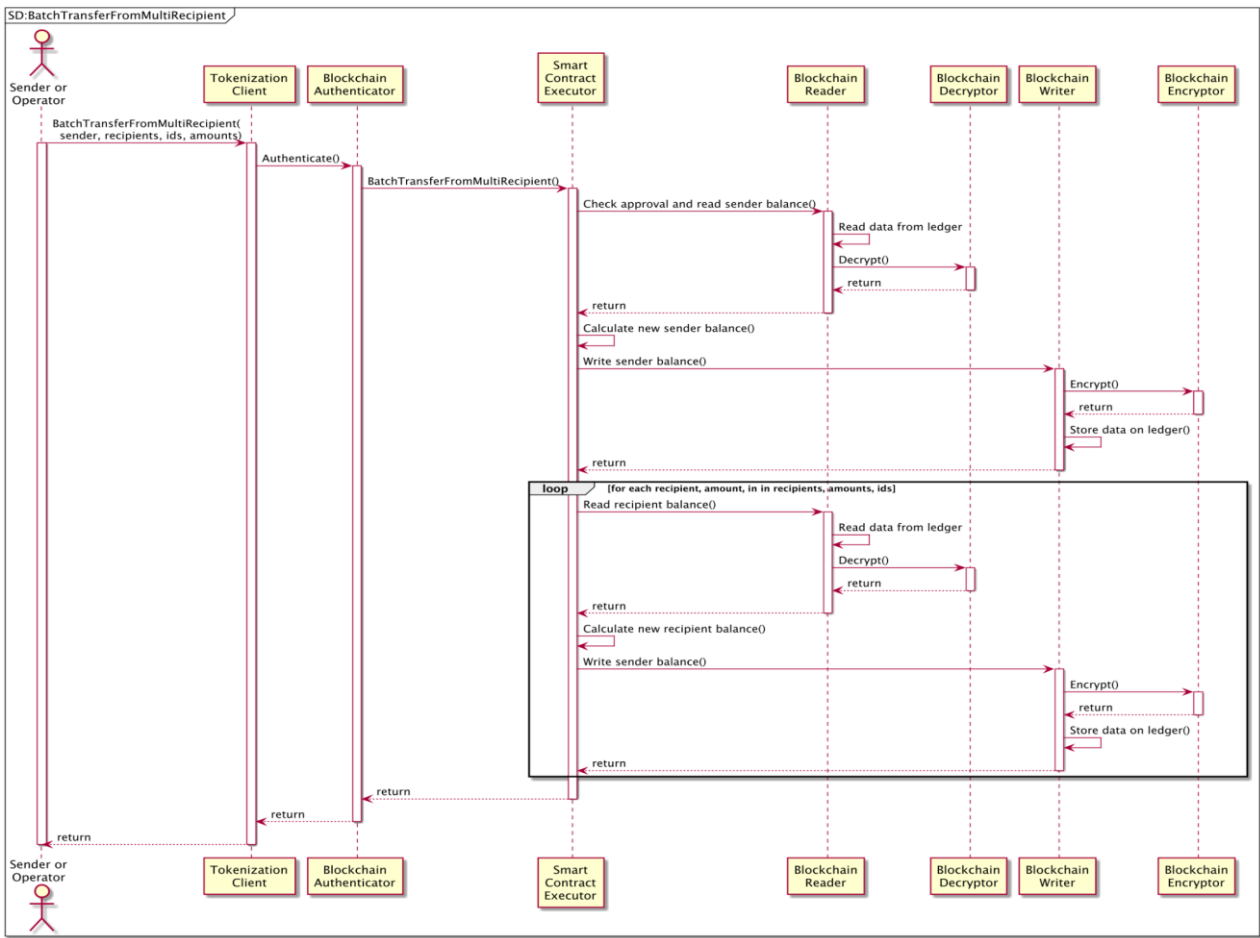


Figure 14 – Batch transfer from multi recipient sequence diagram

3.1.2.14 Broadcast Token Existence

In Broadcast Token Existence use case, a minter emits a `TransferSingle` event to inform listening parties about the existence of a new token. The tokenization client interacts with the blockchain components and verifies that the invoking account has a Minter role. A `TransferSingle` event is emitted through a new transaction in the ledger (Figure 15).

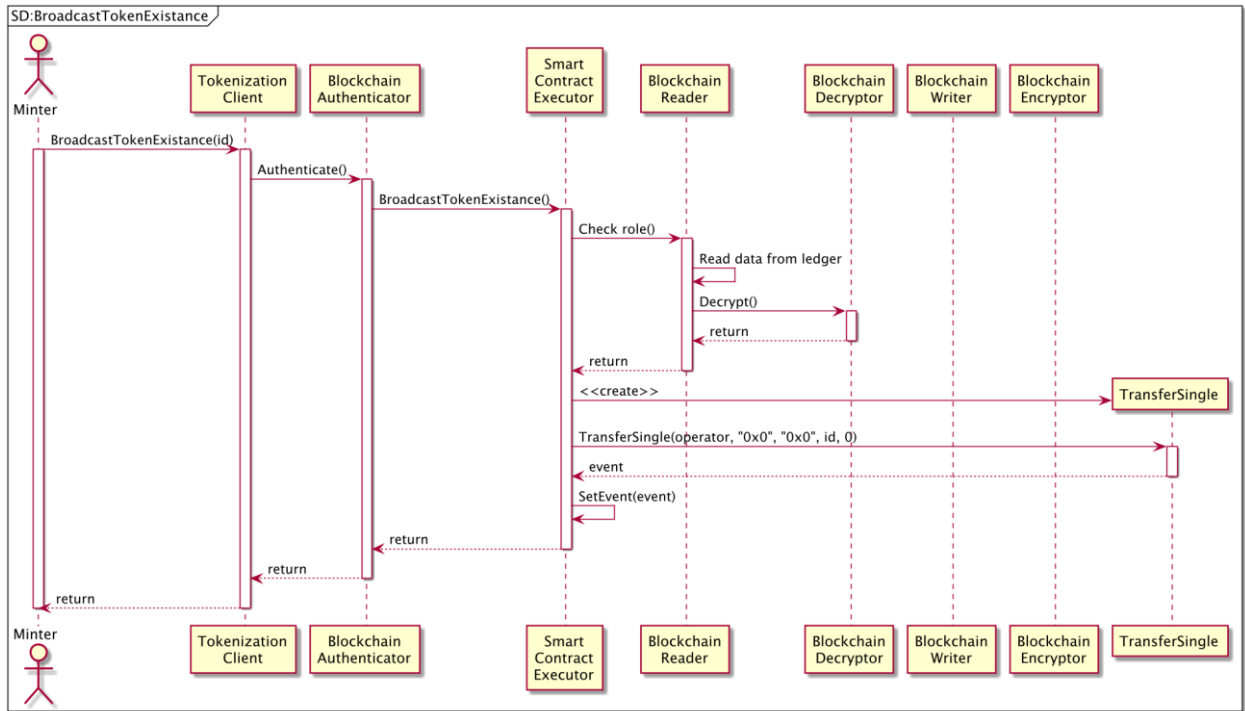


Figure 15 – Broadcast token existence sequence diagram

3.1.2.15 Client account ID

In the Client Account ID use case, an owner account in the blockchain network queries its own client account id. Upon the authentication of the user and their role as a reader, the tokenization client interacts with blockchain components in order to check the client ID of the owner account by querying and reading the query result upon decrypting them (Figure 16).

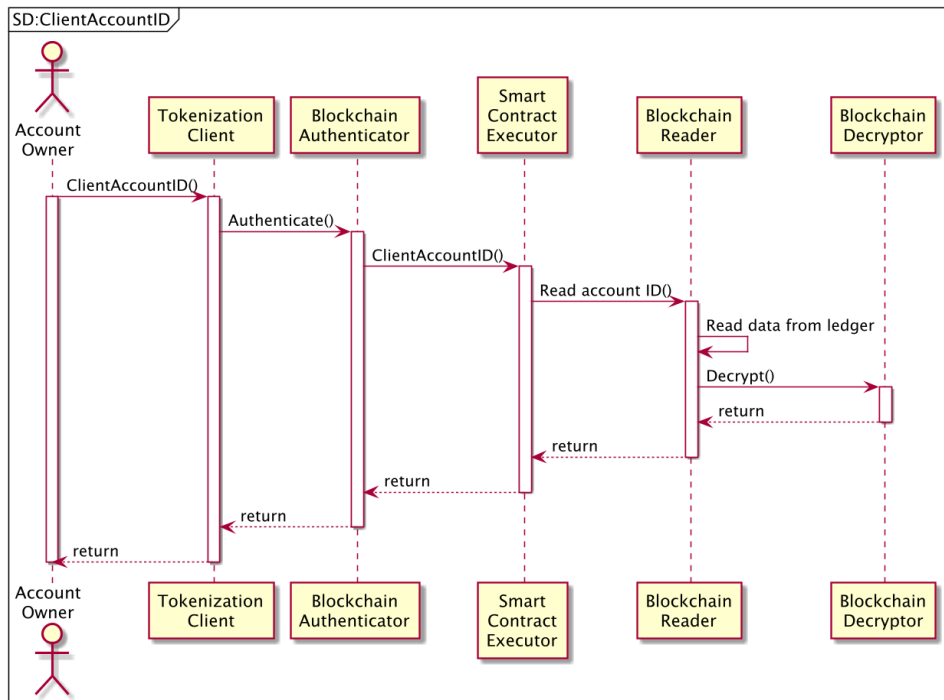


Figure 16 – Client account ID sequence diagram

3.2 Demonstrator

Our demonstrator aims to illustrate different versions of the transfer functions available in ERC 1155 and in what contexts they can be used. We assume that person P1 from organization1, Org1, deploys an ERC 1155 contract and is able to mint tokens. There are also four more persons P2, P3, P4, and P5 from another organization, Org2, who are customers and purchase tokens and hence have various tokens transferred to them. The steps involved in these transactions are as follows:

- *Step1:* Person P1 from the organization calls the MintBatch function in order to create 100 token1s, 200 token2s, 300 token3s, 150 token4s, 100 token5s, 100 token6s.
- *Step2:* Person P1 calls TransferFrom in order to send person P2 six token3s.
- *Step3:* Person P1 calls BatchTransferFrom in order to send person P2 six token3s, three token4s and one token2s.
- *Step4:* Person P1 calls BatchTransferFromMultiReceptient in order to send:
 - six token5s to person P3,
 - six token3s to person P4,
 - three token4s to person P2,
 - two token2s to person P5, and
 - three token6s to person P2.

Steps 2,3, and 4 are depicted in Figure 17 together with the functions called and their parameters. The video animates the three different kinds of token transfers in steps 2,3, and 4.

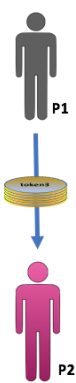
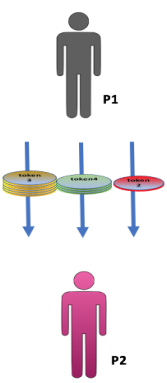
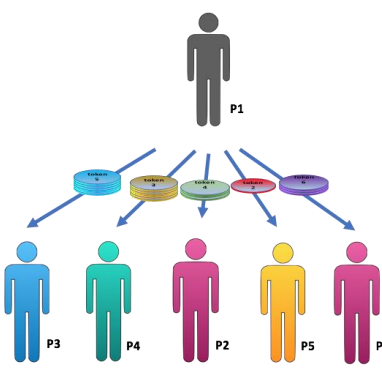
Context			
Function called	TransferFrom	BatchTransferFrom	BatchTransferFromMultiReceptient
example function parameters			
from	"P1"	"P1"	"P1"
to	"P2"	"P2"	["P3","P4","P2","P5","P2"]
id	3	[3,4,2]	[5,3,4,2,6]
values	6	[6,3,1]	[6,6,3,2,3]

Figure 17 – Different types of transfers and their parameters. Note that P1-P5 are client account ids in base64-encoded format

4 Conclusions

One of the most natural ways to extend our initial work on token standards implementations (reported in D4.10) was the development of an extended and more mature standard that includes also NFTs. This was strengthened during the workshop Blockchain Applications in Digital Finance (beyond cryptocurrencies) held in March 2021 in which such a requirement arose.

Tokenization is a disruptive technology as it allows ownership of any asset (e.g., services, bonds, national digital currencies, company shares, product ownerships, tickets, data, cloud machine hours, and licenses) to be represented digitally. Digitally represented assets, also known as virtual assets, can be easily globally marketed to end users, businesses, or agencies with very low transaction costs. Furthermore, these virtual assets can be programmed to be subject to business rules and regulations about their usage. In D4.10 we detailed some of the most prominent use cases for token utilization in FinTech. In finance these include: Cryptocurrency/cross boarder payments; fundraising; letter of credit; credit risk scoring; debt issuance; digital Coin; fiat-backed token; physical asset-backed token; and securities tokens. Examples of potential use cases of tokens in the insurance sector include claim processing; multinational insurance; reinsurance/risk assessment; fraud detection; crowdfunded insurance; and cryptocurrency backed loans.

Whereas tokenization has been practiced in public blockchains for unregulated initial coin offerings during the last six years, its applications at the regulated institutional and enterprise level is just starting. There is an increasing interest from businesses to tokenize their assets since this will enable them to reach out to external markets easily and automate the assets trading. In particular, since (permissioned) Hyperledger Fabric is the choice of blockchain at the enterprise level, standardized ERC-20 and ERC-1155 contributed by INFINITECH project can accelerate tokenization adoption at the enterprise level. To this end, the INFINITECH marketplace can make use of ERC-20 and ERC-1155 token implementations as a way to represent its assets as virtual assets and market them to the outside world. Tokenization of INFINITECH assets in the marketplace could also act as a demonstrated model for other businesses. Finally, since blockchains store transactions, ERC-20 and ERC-1155 token transfer transaction data can also be retrieved through standard interfaces and analysed using big data analysis techniques for providing business intelligence, and hence, also contributing to BDVA efforts.

The purpose of the deliverable at hand titled D4.11 “Blockchain Tokenization and Smart Contracts – II” was to report the outcomes of the work performed within the context of T4.4 “Tokenization and Smart Contracts Finance and Insurance Services” in WP4 from M15 to M22 of the project with regard to extending the scope of tokens-implementation on top of Fabric to include the ERC 1155 standard for both fungible and non-fungible tokens. Although the deliverable is of type “R” (only Report), we provide a full demonstrator of the token workflows along with a recording of the work using an illustrative example. D4.11 is also part of Milestone 12 (MS12) “Second Version of Interoperability and Data Exchange Enablers” at M22.

The main artifacts from the ERC 1155 standard implementation are:

- The implementation of the six standard functions and the additional nine functions which provide a full set of functionalities and ensure avoidance of key collisions while preserving high rate of throughput.
- Code available at the GitLab repository of the project at: <https://gitlab.infinittech-h2020.eu/blockchain/erc1155-tokenization>. The code will be released to open source by the end of the project.
- A demonstrator (movie) showing all flows available at the INFINITECH marketplace: <https://marketplace.infinittech-h2020.eu/infinittech/erc1155-token-smart-contract-for-hyperledger>

Another future direction recognized during the writing of D4.10 was the collaboration with FBK towards a framework for federated machine learning with privacy-preserving and execution guarantees. This is the second parallel thread taken as an outcome of D4.10 which has been developed from M15 to M22 of the project as a collaborative effort between FBK and IBM and is reported separately in D4.14 “Encrypted Data Querying and Personal Data Market – II”.

5 Appendix A: Literature

- [1] “Hyperledger Fabric – Hyperledger,” 2020. [Online]. Available: <https://www.hyperledger.org/use/fabric>. [Accessed 10-July-2021].
- [2] F. Vogelsteller and V. Buterin, “ERC-20 token standard,” 2015. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-20> [Accessed 10-July-2021].
- [3] W. Radomski, A. Cooke, P. Castonguay, J. Therien, E. Binet, and R. Sandford, “ERC-1155 multi token standard,” 2015. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-1155> [Accessed 10-July-2021].
- [4] N. Gaur, L. Desrosiers, V. Ramakrishna, P. Novotny, SA. Baset, and A. O'Dowd, Hands-On Blockchain with Hyperledger, 2018.
- [5] W. Entriken, D. Shirley, E. Evans, and N. Sachs, “EIP-721: ERC-721 non-fungible token standard,” 2018. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-721>. [Accessed 10-July-2021].