Computer Science and Information Systems 20(3):1133–1155 https://doi.org/10.2298/CSIS230127038S

The Proposal of New Ethereum Request for Comments for Supporting Fractional Ownership of Non-Fungible Tokens ⋆

Miroslav Stefanović[1], Đorđe Pržulj[1], Darko Stefanović[1], Sonja Ristić[1], and Darko Čapko[1,2]

[1] University of Novi Sad, Faculty of Technical Sciences
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
{mstef, przulj, darko.stefanovic, sdristic}@uns.ac.rs
[2] Ethernal, Nikolajevska 2, 21000 Novi Sad, Serbia
darko@ethernal.tech

Abstract. During the last couple of years, non-fungible tokens became the most prominent implementation of blockchain technology apart from cryptocurrencies. This is mainly due to their recent association with digital art, but the application of non-fungible tokens has been in the focus of researchers since the appearance of Blockchain 2.0. It was usually tightly coupled with the research on possible applications of blockchain technology in some real-life applications, such as land administration, healthcare, or supply chain management. Since the initial release of the Ethereum blockchain in 2015, until 2022, more than 44 million smart contracts have been created, and out of those that are still active, more than 70% are based on some prominent templates. In the Ethereum blockchain, the creation of non-fungible tokens is usually based on Ethereum Request for Comments 721. In this paper, the authors are proposing the creation of a new standard that would support fractional ownership of non-fungible tokens. Fractional ownership is necessary so non-fungible tokens and blockchain technology could be applied to an even wider number of use cases. This paper also presents an example of a possible implementation of the newly proposed standard in the Solidity programming language.

Keywords: blockchain, smart contract, non-fungible tokens, NFT, Ethereum, ERC

1. Introduction

In 2009, a white paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System" was published and even though it was nowhere directly mentioned by that name, blockchain technology (BT) was born. BT represents the first implementation of distributed ledger technology (DLT). DLT is a solution that, instead of a centralized registry, has a unique registry that is distributed among multiple nodes with decentralized control. These nodes record, share and synchronize data across the network, making the data secure by reaching a consensus on the content of the registry [39]. The first concrete implementation of BT is the Bitcoin blockchain, but over the years various DLT platforms have been implemented [12].

⋆ Based on extended abstract titled "Ethereum Request for Comments for Fractional Ownership of Non-Fungible Tokens" that was presented at XVIII International Symposium - SymOrg2022, Belgrade, Serbia

In BT, transactions are stored in the chain of blocks. Blocks are added in chronological order making the possibility of manipulation or forgery highly unlikely [25]. The system is secured by making data falsification almost impossible because the data is distributed among a large number of interconnected nodes, the so-called blockchain network. Distributed data and the fact that there is no single point of failure make the system resilient, and the fact that the use of the system is public makes it transparent [30]. It is very important to eliminate or at least minimize the possibility of manipulation or forgery within the transactions stored in a blockchain. To achieve this, BT relies on a cryptographic hash function, asymmetric cryptography, and distributed consensus mechanism [55]. The role of the consensus mechanism in the blockchain network is to make it possible for nodes to reach an agreement on the single state of the network [2]. The new block is added to the chain only once nodes perform the same computation, achieving the same result, and reaching a consensus on that result [21]. Some of the advantages of BT are efficiency, security, resilience, and transparency. The fact that it is possible to easily monitor and manage complex data logs with the help of BT makes this solution very effective.

One of the most common classifications divides blockchains into three categories: public permissionless blockchains, consortium/hybrid/public permissioned blockchains, and private blockchains [46]. Public blockchains are blockchains where all transactions are public and anyone can join the network as a node and participate in the process of confirmation of transactions[55][35]. In consortium/hybrid/public permissioned blockchains, transactions are also public, but only a set of predetermined nodes participate in confirmations of transactions. Private blockchains are blockchains where only selected nodes can see and participate in process of confirmations of transactions [28] [45].

The main characteristics of BT are decentralization, persistence, anonymity, and auditability [13][21][35][49][55].

In BT, decentralization is achieved by the possibility for every node to manage and store transactions. Information about transactions is exchanged between all the nodes on the network, thus eliminating the need for a trusted third party [49].

Blocks are added onto a chain by having the content of the previous block, through its hash value, participating in the content of the next block. In that way, each block in the chain participates in the content of its successor block. This implies that in case the content of a previous block, which is already a part of a blockchain, is changed, it would invalidate all the following blocks [13]. In this way, persistency of the transactions recorded within the blockchain is achieved. The new transactions can be added to the blockchain while the possibility of deleting or updating previously registered transactions is highly unlikely [35].

Anonymity is a characteristic of public permissionless and partly of consortium/hybrid/public permissioned blockchains. In these blockchains interested parties could participate as clients by exchanging assets, or even as a node in a public permissionless blockchain, without the need to expose their identity, thus preserving their anonymity.

In [31], Bitcoin is described as a peer-to-peer distributed timestamp server, meaning that all transactions that happen on the blockchain are timestamped. As each transaction is timestamped and since forgery is highly unlikely, as previously mentioned, an interested party can search the blockchain for any previous transactions, thus making the blockchain auditable [13][55].

All mentioned advantages and characteristics related to BT are applicable to smart contracts, too. The term smart contract was mentioned for the first time in 1996 and it was defined as a "set of promises, specified in digital form, including protocols within which the parties perform on these promises" [43]. The idea was based on the possibility for contract clauses to be implemented either through hardware or software in a way that any breach of contract would cause significant expense for a breaching party [43]. In BT, a smart contract is a computer program that is deployed on the blockchain network and that is governed by the same rules that govern transactions [42]. They could be used to automatically verify and execute contract clauses once predetermined conditions have been met [17].

The most common way of representing real-life assets in smart contracts is through tokens, and in the Ethereum blockchain, tokens are usually created in accordance with ERC-20 and ERC-721 standards. ERC-20 token standard represents an interface that will allow the creation of fungible tokens that can be used by other applications, such as wallets and exchanges. Fungible tokens are used for representing interchangeable assets, for example for the creation of new cryptocurrencies. ERC-721 non-fungible token standard represents an interface that will allow the creation of non-fungible tokens (NFT) that can be used in the same manner as ERC-20 tokens but are used to represent unique assets, that can not be interchanged, such as artwork or real-estate. Although these two standards can cover a wide range of use cases, there is a significant number of use cases that can not be fully supported by either of these tokens and those are mainly related to fractional ownership of non-fungible tokens. For example, in previously mentioned applications of land administration and supply chain management, it would surely be necessary for smart contracts to support fractional ownership and use cases in which:

– multiple entities could share ownership of an item,
– entities might have different shares in the ownership, and
– entities having a share of ownership could transfer less than their share of ownership to another entity.

The lack of an adequate standard for supporting fractional ownership of NFTs leads to the creation of smart contracts with different sets of application programming interfaces (APIs). This means that any application that needs to communicate with such a smart contract would need to be tailored to that specific set of APIs. This could especially be problematic if an application needs to communicate with a large number of non-standard smart contracts because it would need to be tailored to each of those smart contracts' specific set of APIs. This represents a major issue and this paper proposes a solution for it.

The solution for this issue is proposed in this paper in a form of a new ERC that will be built upon existing ERC-721 standards and that will propose a solution for the problem of fractional asset ownership and its sharing. Furthermore, the proposed solution defines a standard set of APIs that would make smart contracts implementing this new standard easily interoperable with other applications. This ERC will be proposed in a form of a Unified Modeling Language (UML) class diagram and as a programming interface written in Solidity programming language. The main contribution of the proposed solution is the definition of a novel ERC that could be introduced as a new standard to the existing body of ERC.

Apart from the Introduction and Conclusion, this paper is organized as follows, in Section 2 a short literature review is presented. In Section 3 ERC-20 and ERC-721 standards are presented together with the proposal of the new standard in section 4 and an example of implementation of the proposed standard in section 5.

2. Literature review

Bitcoin blockchain, now often referred to as Blockchain 1.0 [10][53], was introduced with the intention of creating of peer-to-peer electronic cash system that would not suffer from the problem of double spending and will not need a trusted third party to execute transactions [31]. Bitcoin blockchain had some possibilities of application in fields other than cryptocurrency, but true advances came with Blockchain 2.0. Blockchain 2.0 is a term used to represent blockchain that supports smart contracts [34][48]. Smart contracts are programs executed on the blockchain network and their correctness is enforced by the consensus mechanism [4][26]. Smart contracts first appeared on blockchain in 2015 with the Ethereum blockchain network. Ethereum blockchain network was first mentioned in 2014, in a white paper by Vitalik Buterin, where it was announced as a platform for developing Decentralized Applications (DApps) based on smart contracts [8][27].

Since then, the application of BT in fields other than cryptocurrency has come into the focus of scientific research. A literature review conducted in 2017, examining Web of Science, IEEE Xplore, the AIS Electronic Library, ScienceDirect, and SSRN for scholarly journal articles and conference proceedings, looking for conceptual papers or empirical analyses on possible application, use, or implications that BT could have on humans, organizations, and markets, has discovered only 69 papers on these subjects [36]. In 2019, another literature review paper presented the results of research conducted at the beginning of 2018, has shown a significant increase in the number of published papers on this subject. In this research, the main source of scientific papers was Scopus, and 245 papers were identified in fields other than cryptocurrency and finance. Out of those 245 papers, most were in the field of business and industry with 56 papers in total [11]. Research conducted in 2021, which included only journal articles on the subject of security, application, and challenges in BT, that were indexed in Scopus, IEEE Xplore, Google scholar, ScienceDirect, SpringerLink, and Web of Science, showed that this trend continues with 335 analyzed papers [24].

Some of the more prominent examples of possible applications of BT in fields other than cryptocurrency are healthcare [22][29][50], land administration [7][40][42], government [19][23][32], IoT [1][3][52] and supply chain management [5][18][37][38].

According to [51], since the genesis block, the first block of the Ethereum blockchain, over 44 million smart contracts have been deployed on this network. Half of that number has been destroyed, but from the remaining 22 million, around 70% are created based on only 15 templates. These data emphasize the importance of these templates, and in the case of the Ethereum blockchain, templates are usually built in accordance with Ethereum Request for Comments (ERC). In the Ethereum blockchain ERCs represent one of the Ethereum Improvement Proposal (EIP) types that are intended for defining application-level standards and conventions, such as token standards, URI schemes, library/package formats, and name registries.

The introduction of smart contracts into BT has opened the possibility for the development of DApps and Decentralized Autonomous Organizations (DAOs). As the blockchain network represents a distributed system, with no central authority, where decisions are made based on a decentralized consensus mechanism, in the same way, applications that are executed on blockchain networks are also decentralized and represent a special kind of software whose execution is not controlled by a single entity [47]. DAO represents a long-term smart contract for managing certain digital properties that holds all the business rules for one organization and functions without any human intervention [8][9].

Compared to traditional contracts, the following advantages of smart contracts have been recognized.

- Reducing risks – due to the manner in which persistence is achieved in BT, once smart contracts are deployed on a blockchain network, their implementation can not be changed, furthermore, since all transactions are public and since they are being saved on all full nodes they can be audited, thus reducing the risk of malicious behavior.
- Reducing administration and service costs – unlike centralized systems, where there are costs associated with operating trusted third parties, in blockchain networks it is a consensus mechanism that is tasked with confirming transactions, thus reducing the associated costs.
- Improving the efficiency of business processes – the possibility to execute contract clauses automatically, as soon as preconditions are met, can have significant time reduction compared to that required for the process to be executed by a trusted third party [54].

According to [45][54] life cycle of a smart contract consist of the following stages:

- Creation – involved parties, in some cases with help of a solicitor or other legal counsel, draft the initial contract. Software engineers convert this contract into a smart contract. The process of development of smart contracts passes the usual stages of software development, such as design, implementation, and validation.
- Deployment – in this stage, a smart contract is being deployed on the blockchain network. As previously mentioned, once deployed, a smart contract can not be changed and if any change is necessary, then a new smart contract must be deployed.
- Execution – once a smart contract has been deployed, contracted clauses are being monitored. When conditions are met, required functions are executed.
- Completion – once a smart contract has been executed, the state related to all parties in the contract has been updated and the new state has been saved onto the blockchain network.

The first application of smart contracts that achieved public prominence during the last couple of years was NFTs [14]. This prominence came as a result of hype related to digital collectibles. NFTs represent a tokenized item of value where each token owns a unique set of characteristics [33]. In some cases, those tokens can be a part of the same "universe" and still have different values, such as is the case with virtual collectibles Bored Ape Yacht Club, CryptoPunks, and Mutant Ape Yacht Club, or could represent unique digital artwork such as The Merge, The First 5000 Days, and Clock that were sold for almost 92m, 70m, and 53m dollars respectively [20]. Apart from these more prominent examples, significant efforts have been put into research related to the application of NFTs in other fields, and those fields mainly coincide with the previously mentioned field of interest for the general application of BT. NFTs are often classified into six different categories: collectibles, art, metaverse, utility, and others [6]. In some of those categories, having a standard interface that would enable fractional ownership of NFTs would surely be beneficial, while in others, especially those related to real-life goods, it's not just that it would be beneficial, but in fact, it would be necessary.

This paper is based on extended abstract that was presented at SymOrg 2022 - XVIII International Symposium [41]. Presented extended abstract gave a short introduction to the need for a standard that will support fractional ownership of NFTs and provided a draft version of the necessary function that was used as a foundation of this research. The draft UML class diagram presented in this extended abstract was extended, elaborated, and refined for this paper. Based on this new UML class diagram, in this paper, a programming interface written in Solidity programming language is presented. Additionally, the proposed programming interface is accompanied by constraints that an implementation of this interface must satisfy. An example of such implementation of a smart contract, that satisfies all the required constraints, also represents an addition to the previously published extended abstract.

3. Existing ERC standards

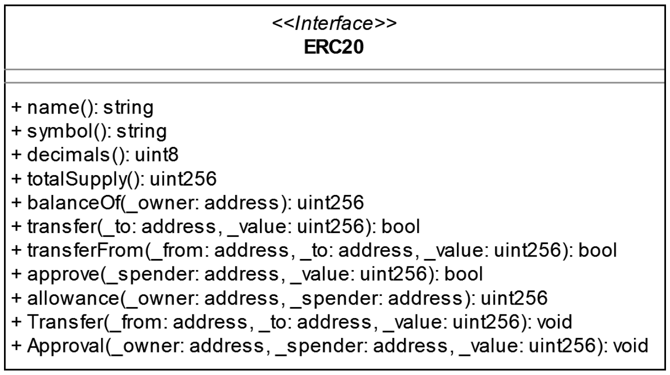
In the Ethereum blockchain network, changes related to core protocol, smart contracts, and client APIs are made based on the Ethereum Improvement Proposal (EIP). EIP represents a standard for specifying potential new capabilities and processes on the Ethereum network. EIPs are divided into several categories, and those categories are:

- Standard track – used for changes that affect almost all segments of the Ethereum network, such as network protocol changes. At the end of 2022, there were 531 EIPs in this category.
- Core – improvements that consensus fork in the consensus mechanism. At the end of 2022, there were 197 EIPs in this category.
- Networking – improvements related to the implementation of devp2p Wire Protocol, RLPx Discovery Protocol and RLPx TCP Transport Protocol. At the end of 2022, there were 14 EIPs in this category.
- Interface – improvements related to API/remote procedure calls (RPC), standards related to method naming, and application binary interface (ABI) of smart contracts. At the end of 2022, there were 46 EIPs in this category.
- ERC – improvements related to standards and conventions at the application level, such as standards for tokens, name registries, uniform resource identifier (URI) schema, and library and package formats. At the end of 2022, there were 274 EIPs in this category, and out of those 274, 46 were in status final, 9 were in the last call, 24 were in review, 73 were in status draft, 117 were stagnant, and 5 were withdrawn.
- Meta – improvements related to the processes surrounding the Ethereum network, but unlike the Standard track, they do not refer to the Ethereum protocol itself. At the end of 2022, there were 20 EIPs in this category.
- Informational – do not represent improvement suggestions, but provide instructions, guidelines, or information to the Ethereum community. At the end of 2022, there were 6 EIPs of this type [16].

Among the 46 ERC standards that are in final status, 7 represent standards related to tokens:

- ERC-20 Token Standard – defines a standard interface that enables the creation of new tokens, which will be used by other applications,
- ERC-721 Non-Fungible Token Standard – defines a standard interface for creating unique (non-fungible) tokens,
- ERC-777 Token Standard – defines the improvement of the ERC-20 Token Standard,
- ERC-1155 Multi Token Standard – defines a standard interface for smart contracts managed by several different tokens,
- ERC-1363 Payable Token – defines the improvement of the ERC-20 Token Standard,
- ERC-3525 Semi-Fungible Token – defines a standard interface for creating tokens that will have part of the features described in ERC-20, and part of the features described in ERC-721, and
- ERC-4626 Tokenized Vaults – defines an enhancement of the ERC-20 Token Standard to provide support for the implementation of tokenized Vaults.

Out of those 7 standards, there are only two basic standards for creating tokens on the Ethereum network, namely ERC-20 and ERC-721, while the remaining five represent improvements of these standards. ERC-20 and ERC-721 are defined in the form of programming interfaces written in the Solidity programming language. In both cases, a set of APIs is defined that should allow tokens created in accordance with these standards to be used by various applications, cryptocurrency wallets, and decentralized exchanges. By implementing either of these two standards, functionality will be implemented that will enable the transfer of tokens by the owner or another authorized entity [15][44]. Both standards will be presented in the form of UML class diagrams and their function calls will be explained. UML class diagram representing the ERC-20 standard is shown in Fig. 1.


```
<<Interface>>
ERC20

+ name(): string
+ symbol(): string
+ decimals(): uint8
+ totalSupply(): uint256
+ balanceOf(_owner: address): uint256
+ transfer(_to: address, _value: uint256): bool
+ transferFrom(_from: address, _to: address, _value: uint256): bool
+ approve(_spender: address, _value: uint256): bool
+ allowance(_owner: address, _spender: address): uint256
+ Transfer(_from: address, _to: address, _value: uint256): void
+ Approval(_owner: address, _spender: address, _value: uint256): void
```

Fig. 1. UML class diagram of ERC-20 standard

Functions *name*(), *symbol*(), *decimals*(), and *totalSupply*() return values representing a name, symbol, decimal value, and total supply of a created token. These values are optional, and while they may improve application usability, other interfaces and smart contracts cannot expect that token name value will exist in every ERC-20 implementation.

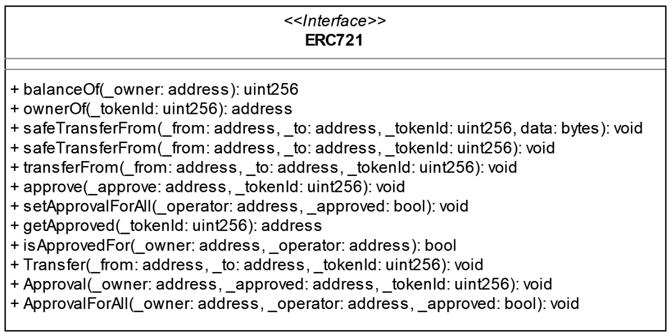
Function *balanceOf*() returns a number representing the amount of tokens owned by that address passed as the argument in the function call. Address type in Solidity programming language represents a 20-byte value of Ethereum address. Depending on the function the address type is used to either represent the current or future owner of a token.

Functions *transfer*() and *transferFrom*() transfer the amount of tokens specified in the function call from either function caller or from the address passed as an argument.

Functions *approve*() and *allowance*() make it possible for an address to be approved to transfer a certain amount of tokens on the behalf of the owner.

Transfer() and *Approval*() events are emitted when corresponding functions are successfully executed [44]. Events are abstractions of the Ethereum logging protocol. In the case of their call, the passed arguments are stored in the transaction log, which is a special data structure on the blockchain. These logs are linked to the smart contract address and are permanently stored on the blockchain.

UML diagram representing the ERC-721 standard is shown in Fig. 2.



```
+ balanceOf(_owner: address): uint256
+ ownerOf(_tokenId: uint256): address
+ safeTransferFrom(_from: address, _to: address, _tokenId: uint256, data: bytes): void
+ safeTransferFrom(_from: address, _to: address, _tokenId: uint256): void
+ transferFrom(_from: address, _to: address, _tokenId: uint256): void
+ approve(_approve: address, _tokenId: uint256): void
+ setApprovalForAll(_operator: address, _approved: bool): void
+ getApproved(_tokenId: uint256): address
+ isApprovedFor(_owner: address, _operator: address): bool
+ Transfer(_from: address, _to: address, _tokenId: uint256): void
+ Approval(_owner: address, _approved: address, _tokenId: uint256): void
+ ApprovalForAll(_owner: address, _operator: address, _approved: bool): void
```

Fig. 2. UML class diagram of ERC-721 standard

Functions *balanceOf*() and *ownerOf*() return the amount of NFTs that an address owns or the address of a token owner, respectively.

Functions *safeTransferFrom*() and *transferFrom*() transfer ownership of a specific token from a previous owner to a new owner. Declarations of two *safeTransferFrom*() functions differ in *data* parameter which could be used to store additional information. Function *transferFrom*() does not perform validity checks related to the new owner.

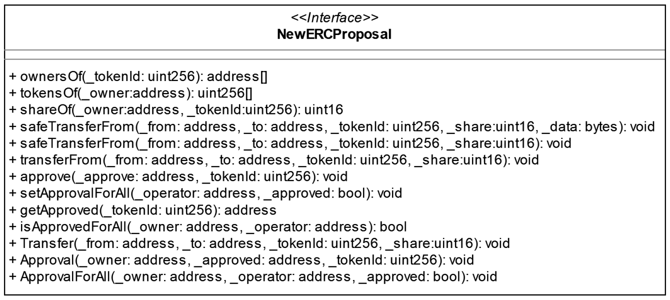
Functions *approve*(), *setApprovalForAll*(), *getApproved*(), and *isApprovedForAll*() are providing a possibility for an entity other than the owner to be approved to transfer the ownership on behalf of the owner and to query information related to those possibilities.

Events *Transfer*(), *Approval*(), and *ApprovalForAll*() are emitted once corresponding functions are successfully executed.

Based on the described characteristics of ERC-20 and ERC-721, it is clear that neither of these two standards meets the needs for managing fractional ownership of non-fungible tokens. Mainly in ERC-20, there is no support for NFTs, while in ERC-721, there are no APIs that will enable storing data about fractional ownership or transferring less than full ownership of a token. To achieve this possibility, different solutions have been used over the years, most commonly creating a combination of two existing standards, but having different implementations makes it hard for smart contracts to communicate with each other because there are no common APIs. Therefore, defining a new standard, with the intention to provide a common set of APIs for managing fractional ownership of non-fungible tokens would be beneficiary. The proposed standard is built upon existing ERC-20 and ERC-721 standards.

4. New ERC standards proposal

UML class diagram representing the proposed standard is shown in Fig. 3.



```
<<Interface>>
NewERCProposal

+ ownersOf(_tokenId: uint256): address[]
+ tokensOf(_owner:address): uint256[]
+ shareOf(_owner:address, _tokenId:uint256): uint16
+ safeTransferFrom(_from: address, _to: address, _tokenId: uint256, _share:uint16, _data: bytes): void
+ safeTransferFrom(_from: address, _to: address, _tokenId: uint256, _share:uint16): void
+ transferFrom(_from: address, _to: address, _tokenId: uint256, _share:uint16): void
+ approve(_approve: address, _tokenId: uint256): void
+ setApprovalForAll(_operator: address, _approved: bool): void
+ getApproved(_tokenId: uint256): address
+ isApprovedForAll(_owner: address, _operator: address): bool
+ Transfer(_from: address, _to: address, _tokenId: uint256, _share:uint16): void
+ Approval(_owner: address, _approved: address, _tokenId: uint256): void
+ ApprovalForAll(_owner: address, _operator: address, _approved: bool): void
```

Fig. 3. UML class diagram of new ERC standard

Function *ownersOf(_tokenId: uint256): address*[] – unlike the function *ownerOf*() from ERC-721 where function call would return the only owner of a token, in case of the proposed function *ownersOf*(), for a token identifier, passed as a *_tokenId* argument of type *uint256*, the function returns the array of data type *address* representing the addresses of the owners of the token.

Function *tokensOf(_owner: address): uint256*[] – this function is based on function *balanceOf*() from ERC-721, but instead of providing a count of owned tokens, a call to proposed *tokensOf*() function for the owner who is identified by argument passed as a *_owner* argument of type *address*, the function returns the array of data type *uint256* representing identifiers of all the tokens that that specific owner owns. In case the function is called with an argument representing a zero address, an exception should be thrown.

Function *shareOf(_owner: address, _tokenId: uint256): uint16* – for the owner that is identified as *_owner* argument of data type *address*, the function will return the share of ownership as data type *uint16*, representing the share that that specific owner has in the token that identified as *_tokenId* argument of data type *uint256* that is passed in the function call. In case the function is called with an argument representing a zero address, an exception should be thrown.

Function *safeTransferFrom(_from: address, _to: address, _tokenId: uint256, _share:uint16, _data: bytes): void* – Similarly to *safeTransferFrom*() function ERC-721 this function for the address of the current owner, which is passed as the *_from* argument, to the address that is passed as *_to* argument, the ownership of the token whose identifier is passed as the *_tokenId* argument, is being transferred, with additional argument *_share*, representing the share of ownership that is being transferred. The function keeps the additional parameter *_data* for the same purpose as is the case in ERC-721. The function does not return data. An exception should be thrown if the address that calls the function is not the owner of the token, if the address is not approved for the transfer of ownership of a specific token, if *_tokenId* is not a valid identifier if the *_to* argument is not a valid address or if transfer share specified in *_share* argument is bigger than the share the owner has in the specific token.

Function *safeTransfer(_from: address, _to: address, _tokenId: uint256), _share: uint16): void* – the function works as in the previous case with the difference that the function does not have an input parameter data, and the value data is set to an empty string ("").

Function *transferFrom(_from: address, _to: address, _tokenId: uint256, _share: uint16): void* – from the address of the current owner, passed as the *_from* argument, to the address passed as the *_to* argument, ownership of the token whose identifier is passed as *_tokenId* is transferred in a share that is specified with argument *_share*. In this case, the function is not expected to check whether the address passed as the *_to* argument is valid, but the check should be done by the client calling the function. An exception will occur in the function if the address calling the function is not the owner of the token, if it is not approved for the transfer of ownership of the specific token, if *_tokenId* is not a valid identifier or if transfer share specified in *_share* argument is bigger than the share the owner has in the specific token.

Functions *approve*(), *setApprovalForAll*(), *getApproved*(), and *isApprovedForAll*() declare the same behavior as already presented in ERC-721, so they will not be presented again.

Event *Transfer(_from: address, _to: address, _tokenId: uint256, _share: uint16)* – an event that must be triggered in the case of a transaction and that will broadcast that ownership has been transferred from the address passed as the *_from* argument to the address passed as the *_to* argument over the token with the identifier passed in the *_tokenId* argument in a share passed as *_share* argument. The requirements set for *Transfer*() event in ERC-721 are valid in the case of this newly proposed *Transfer*() event.

Events *Approval*() and *ApprovalForAll*() declare the same behavior as already presented in ERC-721, so they will not be presented again.

In the following section, a simple implementation of the newly proposed ERC standard in the Solidity programming language will be presented and discussed.

5. Example of implementation of the proposed ERC standard

In this section, one implementation of the proposed ERC standard, written in the Solidity programming language, is presented. In Listing 1, the code representing the declaration of the new programming interface is shown, to be followed by examples of implementations of declared functions in Listings 2 through 6. Helper functions are presented in Listings 7 through 13, while error definitions are shown in Listing 14. The code is divided into several listings to make it easier to comment. In the presented listings, three dots replace the part of the smart contract code that is not relevant to the implementation currently being presented.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
```

```

interface NewERCProposal {
    function ownersOf(uint256 _tokenId) external view
                        returns (address[] memory);
    function tokensOf(address _owner) external view
                        returns (uint256[] memory);
    function shareOf(address _owner, uint256 _tokenId) external
        view
                        returns (uint16);
    function safeTransferFrom(address _from, address _to,
                        uint256 _tokenId, uint16 _share,
                        bytes memory _data) external;
    function safeTransferFrom(address _from, address _to,
                        uint256 _tokenId, uint16 _share) external;
    function transferFrom(address _from, address _to,
                        uint256 _tokenId, uint16 _share) external;
    function approve(address _approve, uint256 _tokenId) external;
    function setApprovalForAll(address _owner, address _operator)
                        external;
    function getApproved(uint256 _tokenId) external view
                        returns (address);
    function isApprovedForAll(address _owner, address _operator)
                        external view returns (bool);

    event Transfer(address indexed _from, address indexed _to,
                        uint256 indexed _tokenId, uint16 _share);
    event Approval(address indexed _owner,
                        address indexed _approved,
                        uint256 indexed tokenId);
    event ApprovalForAll(address indexed _owner,
                        address indexed _operator,
                        bool indexed _approved);
}
...
```

Listing 1. Interface declaration

In Listing 1 the functions are declared using the *function* reserved word, while events are declared using the *event* reserved word. In addition to these reserved words, the following reserved words are also used in the declaration of functions and methods: *external*, which represents one of the four function visibility specifiers in Solidity, *view*, which indicates that the function is not allowed to change the state of the blockchain, *memory*, which indicates that the argument passed to the function call will be saved only temporarily, during the execution of the function, and it will be deleted afterward, and *indexed*, which is used in events and indicates that arguments marked as *indexed* will be saved as a so-called topic and that events can be searched for by these values. In Listing 2, smart contract and state variables definitions are presented.

```
...
contract FractionalOwnership is NewERCProposal {
    address creator;
    uint16 maximumShare;
    mapping(uint256 => address[]) owners;
    mapping(address => uint256[]) tokens;
    mapping(uint256 => mapping(address => uint16)) share;
    mapping(uint256 => address) approved;
    mapping(address => address) approvedForAll;
```

```
44
45      constructor(uint16 _maximumShare) {
46          creator = msg.sender;
47          maximumShare = _maximumShare;
48      }
49  ...
```

Listing 2. Smart contract and state variables definitions

Listing 2 begins with the reserved word *contract*, followed by the name of the smart contract *FractionalOwnership*, and in line 36 it is declared that the smart contract implements the *NewERCProposal* interface. In listing 2, the following state variables are declared:

- *creator* – variable representing the address that was used to deploy the smart contract, and that will be used in the *mint*() function that will be presented in Listing 7;
- *maximumShare* – variable of type *uint16* that represents the maximum share in the ownership of a token. The value should be large enough to cover all the necessary use cases in the specific application;
- *owners* – variable of type *mapping*, mappings in the Solidity programming language represent the so-called key/value structure and in this case, it represents the relationship between the *uint256* value, which represents a token, and an array of *address* data type, which represent the owners of the token, is mapped;
- *tokens* — maps the relationship between the *address*, which represents the owner, and a series of *uint256* values, which represent tokens over which the address has ownership shares;
- *share* — maps the relationship between the *uint256* value, which represents the token, and the mapping that maps the relationship between the *address*, representing the owner, and the *uint16* value, representing the ownership share;
- *approved* - maps the relationship between the *address*, which represents the approved address for managing a token identified by *uint256* value;
- *approvedForAll* - maps the relationship in which key *address* grants the rights to value *address* to manage all of key address's tokens.

In addition to state variables, a constructor is declared and implemented in Listing 2. In smart contracts, the constructor is a function that is called only once, when the smart contract is placed on the blockchain network. In this particular case, the implementation of the smart contract is such that in line 46 the smart contract first queries which address sent the request for its creation and sets that value in the *creator* state variable and then in line 47, it sets the passed *uint16* argument in the *maximumShare* state variable. In Listing 3 implementation of *ownersOf*() function will be presented.

```
58  ...
59      function ownersOf(uint256 _tokenId) override external view
60                          returns (address[] memory){
61          if(isOwnerZeroAddress(_tokenId)) {
62              revert zeroAddress({
63                  _owner: address(0),
64                  _message:
65                      bytes("Zero address can not be queried.")
66              });
67          }
68          return owners[_tokenId];
69      }
70  ...
```

Listing 3. Example of implementation of *ownersOf*() function

The reserved word *override* in the *ownersOf*() function declaration in Listing 4 indicates that it is an implementation of the function declared in *NewERCProposal* interface. In lines 61 to 67, a check is made to see if the argument passed in the function call is bound to the zero address. This is done by calling *isOwnerZeroAddress*() function that will be presented Listing 10. In the case that this function call returns *true*, *revert*() function is called. The *revert*() function, will revert any changes that might have happened during the execution of the initial function call and throw *zeroAddress* error. The error *zeroAddress* accepts as parameters a zero address and the message that zero addresses cannot be queried. In case the call to the *isOwnerZeroAddress* function returns *false*, the function will perform a query on the *owners* state variable by passing the *_tokenId* argument and get a list of addresses representing all owners of a token. In Listing 4, implementations of *tokensOf*() and *shareOf*() functions will be presented.

```
70  ...
71      function tokensOf(address _owner) override external view
72                          returns (uint256[] memory) {
73          if(_owner == address(0)) {
74              revert zeroAddress({
75                  _owner: address(0),
76                  _message:
77                      bytes("Zero address can not be queried.")
78              });
79          }
80          return tokens[_owner];
81      }
82
83      function shareOf(address _owner, uint256 _tokenId) override
84                          public view returns (uint16){
85          if(_owner == address(0)) {
86              revert zeroAddress({
87                  _owner: address(0),
88                  _message:
89                      bytes("Zero address can not be queried.")
90              });
91          }
92          return share[_tokenId][_owner];
93      }
94  ...
```

Listing 4. Example of implementation of *tokensOf*() and *shareOf*() functions

The role of the *tokensOf*() function is to make it possible to find out all the tokens associated with a specific address. In lines 73 to 79 requirements related to zero address are checked and in case those requirements are met in line 80 array of tokens owned by the address for which the function is called is returned.

The *shareOf*() for the arguments representing the owner and a token, if requirements related to zero address are met, as shown in Lines 85 to 91, will in line 92 return the share of ownership that *_owner* had over *_tokenId*. In Listing 4, implementations of *safeTransferFrom*() and *transferFrom*() functions will be presented.

```
 94  ...
 95      function safeTransferFrom(address _from, address _to,
 96                          uint256 _tokenId, uint16 _share,
 97                          bytes memory _data) override public {
 98          if(_to == address(0)) {
 99              revert zeroAddress({
100                  _owner: address(0),
101                  _message:
102                      bytes("Tokens can not be sent to zero address."
                             )
103              });
104          }
105          transferFrom(_from, _to, _tokenId, _share);
106      }
107
108      function safeTransferFrom(address _from, address _to,
109                          uint256 _tokenId, uint16 _share)
110                          override external {
111          safeTransferFrom(_from, _to, _tokenId, _share, "");
112      }
113
114      function transferFrom(address _from, address _to,
115                          uint256 _tokenId, uint16 _share)
116                          override public {
117          checkIfTransferIsPermited(_from, _tokenId, _share);
118
119          share[_tokenId][_from] -= _share;
120          share[_tokenId][_to] += _share;
121          addToTokensIfNewToken(_tokenId, _to);
122          removeFromOwnersIfNoShare(_tokenId, _from);
123          addToOwnersIfNewOwner(_to, _tokenId);
124          removeFromTokensIfNoShare(_from, _tokenId);
125          emit Transfer(_from, _to, _tokenId, _share);
126      }
127  ...
```

Listing 5. Example of implementation of *safeTransferFrom*() and *transferFrom*() functions

In Listing 5, starting from line 95, *safeTransferFrom*() function is implemented. Firstly in lines 98 through 104 requirements related to zero address are checked, to be followed by a call to *transferFrom*() function.

In lines 108 through 111, function implementation of *safeTranferFrom*() function is shown for the call that does not have *_date* parameter, or as required, call that has an empty string for *_date*.

Implementation of *transferFrom*() function is shown in lines 113 through 124. Firstly, function *checkIfTransferIsPermited*() is called to check if necessary conditions for transfer have been met, this function will be presented in Listing 8. If all conditions are met, in lines 119 and 120 the share of ownership will be reduced and increased for old and new owners, to be followed by calls to function *addToTokensIfNewToken*(), *removeFromOwnersIfNoShare*(), *addToOwnersIfNewOwner*(), and *removeFromTokensIfNoShare*() in lines 121 through 124, for adding/removing tokens/owners from *owners* and *tokens* state variables. These functions will be presented in Listing 12 and 13. In line 125 required *Transfer*() event is being emitted. In Listing 6, implementations of functions *approve*(), *setApprovalForAll*(), *isApprovedForAll*(), and *getApproved*() are presented.

```
127  ...
128      function approve(address _approve, uint256 _tokenId)
129                          override external{
130          require(isInOwners(msg.sender, _tokenId),
131                  "Caller is not the owner.");
132          if(_approve == address(0)) {
133              revert zeroAddress({
134                  _owner: address(0),
135                  _message: bytes
136                  ("Zero address can not be approved")
137              });
138          }
139          approved[_tokenId] = _approve;
140          emit Approval(msg.sender, _approve, _tokenId);
141      }
142
143      function setApprovalForAll(address _owner, address _operator)
144                          override external {
145          approvedForAll[_owner] = _operator;
146      }
147
148      function isApprovedForAll(address _owner, address _operator)
149                          override external view returns (bool){
150          return approvedForAll[_owner] == _operator;
151      }
152
153      function getApproved(uint256 _tokenId) override external view
154                          returns (address){
155          return approved[_tokenId];
156      }
157  ...
```

Listing 6. Example of implementation of *approve*(), *setApprovalForAll*(), *isApprovedForAll*(), and *getApproved*() functions

In Listing 6, starting with line 128 *approve*() function is implemented. In lines 129 and 130 requirement that the caller has a share in the ownership of a token is checked. The requirement that zero address can not be approved is checked in lines 131 through 137. If all requirements are met in line 138 state variable *approved* is updated with the new approval and in line 139 *Approval*() event is emitted.

In lines 142 through 145 function *setApprovalForAll*() is implemented by mapping *_owner* and *_operator* in *approvedForAll* state variable in line 144.

Functions *isApprovedForAll*() and *getApproved*() are implemented in lines 147 through 150 and 152 through 155 respectively returning results of calls to *approvedForAll* and *approved* state variables.

```
49  ...
50      function mint(uint _tokenId) external {
51          require(msg.sender == creator,
52                  "Sender not creator address.");
53          addToOwnersIfNewOwner(creator, _tokenId);
54          addToTokensIfNewToken(_tokenId, creator);
55          share[_tokenId][creator] = maximumShare;
56          emit Transfer(address(0), creator, _tokenId, maximumShare);
57      }
58  ...
```

Listing 7. Example of implementation of *mint*() function

The *mint*() is not declared in *NewERCProposal* interface, but it represents a common solution for the initial creation of tokens and usually, it is only available for the address that initially deployed the contract and that is why that address is preserved in the *creator* state variable. The function accepts the parameters *_tokenId*, which represents the identifier of the token to be created. In lines 52 and 53, the requirement that the function call came from the *creator* address is checked, and if this requirement is met in lines 54 through 56 functions required for the creation of new token are called in a similar way as it was the case with *transferFrom*() function. In line 57, *Transfer*() event is emitted. Helper function checkIfTransferIsPermited is presented in Listing 8.

The remaining, helper functions will be presented in the following listings: in Listing 8 *checkIfTransferIsPermited*() function, functions *isInOwners*() and *isInTokens*() are presented in Listing 9, to be followed by the implementation of *isOwnerZeroAddress*() functions in Listing 10. Listing 11 presents implementations of functions *getIndexOfOwner*() and getIndexOfToken() functions, while in Listing 12 *addToOwnersIfNewOwner*() and *removeFromOwnersIfNoShare*() are presented. Listing 13 holds implementations of functions *addToTokensIfNewToken*() and *removeFromTokensIfNoShare*(), to be concluded with error declarations in Listing 14.

```
...
    function checkIfTransferIsPermited(address _from,
                        uint256 _tokenId, uint16 _share)
                        internal view{
         if(_from == address(0)) {
            revert zeroAddress({
                _owner: address(0),
                _message: bytes
                 ("Transfers from zero address are not allowed.")
            });
        }

        if (!isInOwners(msg.sender, _tokenId)
            && !(msg.sender == approved[_tokenId]
            && !(msg.sender == approvedForAll[_from]))) {
            revert notOwnerOrApproved({
                _tokenId: _tokenId,
                _from: _from
            });
        }

        if (shareOf(_from, _tokenId) < _share) {
            revert notOwningBigEnoughShare({
                _tokenId: _tokenId,
                _from: _from,
                _owningShare: shareOf(_from, _tokenId),
                _transferingShare: _share
            });
        }
    }
```

```
...
```

Listing 8. Example of implementation of *checkIfTransferIsPermited*() function

```
...
    function isInOwners(address _address, uint256 _tokenId)
                        internal view returns (bool) {
        address[] memory allOwners = owners[_tokenId];
        for (uint i=0; i < allOwners.length; i++) {
            if (allOwners[i] == _address ) {
                return true;
            }
        }
        return false;
    }

    function isInTokens(uint256 _tokenId, address _address)
                        internal view returns (bool) {
        uint256[] memory allOwned = tokens[_address];
        for (uint i=0; i < allOwned.length; i++) {
            if (allOwned[i] == _tokenId ) {
                return true;
            }
        }
        return false;
    }
...
```

Listing 9. Example of implementation of *isInOwners*() and *isInTokens*() functions

```
...
    function isOwnerZeroAddress(uint256 _tokenId)
                        internal view returns (bool) {
        address[] memory allOwners = owners[_tokenId];
        for (uint i=0; i < allOwners.length; i++) {
            if (allOwners[i] == address(0) ) {
                return true;
            }
        }
        return false;
    }
...
```

Listing 10. Example of implementation of *isOwnerZeroAddress*() function

```
...
    function getIndexOfOwner(uint256 _tokenId,
                        address _owner)
                        internal view returns (int){
        for(uint i = 0;
            i < owners[_tokenId].length; i++){
            if(_owner == owners[_tokenId][i])
                return int(i);
        }
        return -1;
    }

    function getIndexOfToken(uint256 _tokenId,
                        address _owner)
                        internal view returns (int){
        for(uint i = 0; i < tokens[_owner].length; i++){
            if(_tokenId == tokens[_owner][i])
                return int(i);
        }
        return -1;
    }
...
```

Listing 11. Example of implementation of *getIndexOfOwner*() and getIndexOfToken() functions

```
...
    function addToOwnersIfNewOwner(address _owner, uint256 _tokenId
        )
                        internal {
        if (!isInOwners(_owner, _tokenId)) {
            owners[_tokenId].push(_owner);
        }
    }

    function removeFromOwnersIfNoShare(uint256 _tokenId,
                        address _from) internal {
        if (shareOf(_from, _tokenId) == 0) {
            int i = getIndexOfOwner(_tokenId, _from);
            if (i != -1) {
                owners[_tokenId][uint(i)] =
                    owners[_tokenId][owners[_tokenId].length - 1];
                owners[_tokenId].pop();
            }
        }
    }
...
```

Listing 12. Example of implementation of *ownersOf*() function

```
..
    function addToTokensIfNewToken(uint256 _tokenId, address _owner
        )
                        internal {
        if (!isInTokens(_tokenId, _owner)) {
            tokens[_owner].push(_tokenId);
        }
    }

    function removeFromTokensIfNoShare(address _owner,
                        uint256 _tokenId) internal {
        if (shareOf(_owner, _tokenId) == 0) {
        int i = getIndexOfToken(_tokenId, _owner);
            if (i != -1) {
                tokens[_owner][uint(i)] =
            tokens[_owner][tokens[_owner].length - 1];
                tokens[_owner].pop();
            }
        }
    }
...
```

Listing 13. Example of implementation of *ownersOf*() function

```
...
    error zeroAddress(address _owner, bytes _message);
    error notApproved(address _from);
    error notOwnerOrApproved(uint256 _tokenId, address _from);
    error notOwningBigEnoughShare(uint256 _tokenId, address _from,
                         uint16 _owningShare,
                         uint16 _transferingShare);
    error documentHashMustBeProvided(address _from, address _to,
                         uint256 _tokenId,
                         uint16 _share,
                         bytes _documentHash);
}
```

Listing 14. Example of implementation of *ownersOf*() function

6. Conclusion

BT has for some time been described as a technology that could be used in fields other than cryptocurrency and fintech. Various possible applications that would benefit from the advantages and characteristics of BT have been identified. The most common applications of BT in those filed are tied to smart contracts and either fungible or NFTs. NFTs are recognized as a possible way to represent unique items from the real world in a blockchain network. What was identified as possible improvements related to NFTs was a standard set of APIs for the representation of fractional ownership of NFTs. In this paper, a proposal for a new ERC is made in a form of a UML class diagram and an interface written in Solidity programming language. Also, the implementation of the proposed interface is presented in a form of a smart contract, together with all the required constraints. The adoption of such a new ERC would define a standard set of APIs for exchanging function calls that would solve the issue that was usually resolved by combining multiple standards.

Future research could be directed into the possible optimization of smart contracts implementing the proposed new ERC. The costs associated with running smart contracts on the Ethereum network are related to the gas spent during the prices of execution of a transaction. Limiting those costs should be of concern. This concern is of especially big importance in use cases where there are a significant number of NFTs managed by a single smart contract. Preliminary research on the proposed implementation shows that the amount of gas spent related to the transaction could vary between 140.000 and 2.800.000 gas. Another possible research might be related to cost comparison between the application of a single smart contract that will manage multiple NFTs, or multiple smart contracts, each representing a single NFT.

References

1. Abdelmaboud, A., Ahmed, A., Abaker, M., Eisa, T., Albasheer, H., Ghorashi, S., Karim, F.: Blockchain for iot applications: Taxonomy, platforms, recent advances, challenges and future research directions. Electronics 11(4) (2022)

2. Aggarwal, S., Kumar, N., Chelliah, P.: Cryptographic consensus mechanisms. Advances in Computers 121(4), 211–226 (2021)
3. Alamri, M., Jhanjhi, N., Humayun, M.: Blockchain for internet of things (iot) research issues challenges & future directions: A review. International Journal of Computer Science and Network Security 19(5), 248–258 (2019)
4. Alotaibi, L., Alshamrani, S.: Smart contract: Security and privacy. Computer Systems Science and Engineering 38(1), 93–101 (2021)
5. Azzi, R., Chamoun, R., Sokhn, M.: The power of a blockchain-based supply chain. Computers & Industrial Engineering 135 (2019)
6. Bao, H., Roubaud, D.: Non-fungible token: A systematic review and research agenda. Journal of Risk and Financial Management 15(215) (2022)
7. Bennett, R., Miller, T., Pickering, M., Kara, A.: Hybrid approaches for smart contracts in land administration: Lessons from three blockchain proofs-of-concept. Land 10(2) (2021)
8. Buterin, V.: Ethereum white paper - a next generation smart contract & decentralized application platform. `https://blockchainlab.com/pdf/Ethereum\_white\_paper\-a\_next\_generation\_smart\_contract\_and\_decentralized\_application\_platform\-vitalik\-buterin.pdf` (2015), accessed: 2022-12-04
9. Cai, W., Wang, Z., Ernst, J., Hong, Z., Feng, C., Leung, V.: Decentralized applications: The blockchain-empowered software system. IEEE Access 6, 53019–53033 (2018)
10. Cao, X., Zhang, J., Wu, X., Liu, B.: A survey on security in consensus and smart contracts. Peer-to-Peer Networking and Applications 15, 1008–1028 (2022)
11. Casino, F., Dasaklisb, T., Patsakis, C.: A systematic literature review of blockchain-based applications: Current status, classification and open issues. Telematics and Informatics 36, 55–81 (2019)
12. Chowdhury, M., Ferdous, M., Biswas, K., Chowdhury, N., Kayes, A., Alazab, M., Watters, P.: A comparative analysis of distributed ledger technology platforms. IEEE Access 7, 167930–167943 (2019)
13. Dai, H., Zheng, Y., Zhang, Y.: Blockchain for internet of things: A survey. IEEE Internet of Things Journal 6(5), 8076–8094 (2019)
14. Dowling, M.: Is non-fungible token pricing driven by cryptocurrencies? Finance Research Letters 44 (2022)
15. Entriken, W., Shirley, D., Evans, J., Sachs, N.: Eip-721: Non-fungible token standard. `https://eips.ethereum.org/EIPS/eip-721` (2018), accessed: 2022-10-23
16. Ethereum: Ethereum improvement proposals. `https://eips.ethereum.org/` (2022), accessed: 2022-12-04
17. Giancaspro, M.: Is a 'smart contract' really a smart idea? insights from a legal perspective. Computer Law & Security Review 33(6), 825–835 (2017)
18. Guido, R., Mirabelli, G., Palermo, E., Solina, V.: A framework for food traceability: Case study–italian extra-virgin olive oil supply chain. International Journal of Industrial Engineering and Management 11(1), 50–60 (2020)
19. Hassija, V., Chamola, V., Krishna, D., Kumar, N., Guizani, M.: A blockchain and edge-computing-based secure framework for government tender allocation. IEEE Internet of Things Journal 8(4), 2409–2418 (2021)
20. Hood, D.: The most expensive nfts ever sold. `https://www.business2community.com/nft/most-expensive-nft` (2022), accessed: 2022-11-23
21. Huo, R., Zeng, Z., Wang, J., Shang, W., Chen, T., Huang, S., Wang, R., Yu, Y., Liu, Y.: A comprehensive survey on blockchain in industrial internet of things: Motivations, research progresses, and future challenges. IEEE Communications Survey & Tutorials 24(1), 88–122 (2022)
22. Hussien, H., Yasin, S., Yan, J., Udzir, N., Ninggal, M., Salman, S.: Blockchain technology in the healthcare industry: Trends and opportunities. Journal of Industrial Information Integration 22 (2021)

23. Kassen, M.: Blockchain and e-government innovation: Automation of public information processes. Information Systems 103 (2022)
24. Le, T., Hsu, C.: A systematic literature review of blockchain technology: Security properties, applications and challenges. Journal of Internet Technology 22(4), 789–801 (2021)
25. Lu, y.: The blockchain: State-of-the-art and research challenges. Journal of Industrial Information Integration 15, 80–90 (2019)
26. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. Association for Computing Machinery, New York, NY, United States (2016)
27. Majeed, U., Khan, L., Yaqoob, I., Kazmi, S., Salah, K., Hong, C.: Blockchain for iot-based smart cities: Recent advances, requirements, and future challenges. Journal of Network and Computer Applications 181 (2021)
28. Marjanović, J., Dalčeković, N., Sladić, G.: Blockchain-based model for tracking compliance with security requirements. Computer Science and Information Systems 20(1), 359–380 (2023)
29. Matulevicius, R., Iqbal, M., Elhadjamor, E., Ghannouchi, S., Bakhtina, M., Ghannouchi, S.: Ontological representation of healthcare application security using blockchain technology. INFORMATICA 33(2), 365–397 (2022)
30. Mohsin, A., Zaidan, A., Zaidan, B., Albahri, A., Albahri, M., Alsalem, M., Mohammed, K.: Ontological representation of healthcare application security using blockchain technology. INFORMATICA 33(2), 365–397 (2022)
31. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf` (2009), accessed: 2022-11-28
32. Ning, X., Ramirez, R., Khuntia, J.: Blockchain-enabled government efficiency and impartiality: using blockchain for targeted poverty alleviation in a city in china. Information Technology for Development 27(3), 599–616 (2021)
33. Park, A., Kietzmann, J., Pitt, L., Dabirian, A.: The evolution of nonfungible tokens: Complexity and novelty of nft use-cases. IT Professional 24(1), 9–14 (2022)
34. Peng, K., Li, M., Huang, H., Wang, C., Wan, S., Choo, K.: Security challenges and opportunities for smart contracts in internet of things: A survey. IEEE Internet of Things Journal 8(15), 12004–12020 (2021)
35. Politou, E., Casino, F., Alepis, E., Patsakis, C.: Blockchain mutability: Challenges and proposed solutions. IEEE Transactions on Emerging Topics in Computing 9, 1082–1986 (2021)
36. Risius, M., Spohrer, K.: A blockchain research framework: What we (don't) know, where we go from here, and how we will get there. Business & Information Systems Engineering 59(6), 385–409 (2017)
37. Saberi, S., Kouhizadeh, M., Sarkis, J., Shen, L.: Blockchain technology and its relationships to sustainable supply chain management. International Journal of Production Research 57(7), 2117–2135 (2019)
38. Saini, H., Dash, S., Kumar Pani, S., Jos e Sousa, M., Rocha, A.: Blockchain-based raw material shipping with poc in hyperledger composer. Computer Science and Information Systems 19(3), 1075–1092 (2022)
39. Shahaab, A., Lidgey, B., Hewage, C., Khan, I.: Applicability and appropriateness of distributed ledgers consensus protocols in public and private sectors: A systematic review. IEEE Access 7, 43622–43636 (2019)
40. Sladić, G., Milosavljević, B., Nikolić, S., Sladić, D., Radulović, A.: A blockchain solution for securing real property transactions: A case study for serbia. ISPRS International Journal of Geo-Information 10(1) (2021)
41. Stefanovic, M., Pržulj, D., Stefanović, D.: Making smart contracts smarter. In: Book of abstracts of Symorg 2022. pp. 10–12. Faculty of Organizational Sciences, University of Belgrade, Belgrade, Serbia (2022)
42. Stefanović, M., Pržulj, D., Ristić, S., Stefanović, D., Nikolić, D.: Smart contract application for managing land administration system transactions. IEEE Access 10, 2169–3536 (2022)

43. Szabo, N.: Smart contracts: Building blocks for digital markets. Extropy, 16, 50–53, 61–63 (1996)
44. Vogelsteller, F., Buterin, V.: Eip-20: Epc-20 token standard. `https://eips.ethereum.org/EIPS/eip-20` (2015), accessed: 2022-10-23
45. Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., Wang, F.: Blockchain-enabled smart contracts: Architecture, applications, and future trends. IEEE Transactions on Systems, Man, and Cybernetics: Systems 49(11), 2266–2277 (2019)
46. Woznica, A., Kedziora, M.: Performance and scalability evaluation of a permissioned blockchain based on the hyperledger fabric, sawtooth and iroha. Computer Science and Information Systems 19(2), 659–678 (2022)
47. Wu, K., Ma, Y., Huang, G., Liu, X.: A first look at blockchain-based decentralized applications. Software: Practice and Experience 51, 2033–2050 (2021)
48. Xuan, S., Zheng, L., Chung, I., Wnag, W., Man, D., Du, X., Yang, W., Guizani, M.: An incentive mechanism for data sharing based on blockchain with smart contracts. Computers and Electrical Engineering 83 (2020)
49. Yang, L.: The blockchain: State-of-the-art and research challenges. Journal of Industrial Information Integration 15, 80–90 (2019)
50. Yaqoob, I. and Salah, K., Jayaraman, R., Al-Hammadi, Y.: Blockchain for healthcare data management: opportunities, challenges, and future recommendations. Neural Computing & Applications 34(44), 11475–11490 (2022)
51. Young, M.: Over 44 million contracts deployed to ethereum since genesis: Research. `https://cryptopotato.com/over-44-million-contracts-deployed-to-ethereum-since-genesis-research/` (2022), accessed: 2022-11-16
52. Zafar, S. and Bhatti, K., Shabbir, M., Hashmat, F., Akbar, A.: Blockchain for healthcare data management: opportunities, challenges, and future recommendations. Annals f Telecommunications 77(1-2), 13–32 (2021)
53. Zhao, J., Fan, S., Yan, J.: Overview of business innovations and research opportunities in blockchain and introduction to the special issue. Finacial Inovation 2(28) (2016)
54. Zheng, Z., Xie, S., Dai, H., Chen, X., Chen, J., Weng, J., Imran, M.: An overview on smart contracts: Challenges, advances and platforms. Future Generation Computer Systems 105, 475–491 (2020)
55. Zheng, Z., Xie, S., Dai, H., Chen, X., Wang, H.: Blockchain challenges and opportunities: A survey. International Journal of Web and Grid Services 14(2), 352–375 (2018)

Miroslav Stefanović received his B.S. degree in information management in 2014 and the M.S. degree in information systems engineering in 2016 from the University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia. He received the Ph.D. degree in industrial engineering and engineering management from the same institution in 2023. From 2015 to 2016, hewas a Teaching Associate and since 2016, he is a Teaching Assistant at the University of Novi Sad, Faculty of Technical Sciences, Department of Industrial Engineering and Engineering Management, Chair for Information and Communication Systems. His research interests include blockchain technologies, especially the implementation of blockchain technology in fields other than cryptocurrency, mainly e-government and land administration systems.

Đorđe Pržulj received his B.S. degree in mechanical engineering in 1999 and the Mr degree in industrial engineering and engineering management in 2004 from the University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia. He received the Ph.D.

degree in industrial engineering and engineering management from the same institution in 2013. From 1999 to 2013 he was a Teaching Assistant and from 2013 to 2018 Assistant Professor at the University of Novi Sad, Faculty of Technical Sciences, Department of Industrial Engineering and Engineering Management, Chair for Information and Communication Systems. Since 2018 he has been an Associate Professor at the same institution. His research interests include land administration systems, service oriented architecture, microservices, ontologies, domain specific languages, and blockchain applications in information systems. Đorđe Pržulj has published in several international information systems journals.

Sonja Ristić works as a full professor at the University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia. She received two bachelor's degrees with honors from the University of Novi Sad, one in Mathematics, Faculty of Science in 1983, and the other in Economics from Faculty of Economics, in 1989. She received her Mr (2 years) and Ph.D. degrees in Informatics, both from the University of Novi Sad, Faculty of Economics, in 1994 and 2003. From 1984 until 1990 she worked with the Novi Sad Cable Company NOVKABEL–Factory of Electronic Computers. From 1990 till 2006 she was with Novi Sad School of Business, and since 2006 she has been with the University of Novi Sad, Faculty of Technical Sciences. Her research interests include database systems, software engineering, model-driven software engineering and domain specific languages. She is the author or co-author of over 100 papers, and 10 industry projects and software solutions in the area.

Darko Stefanović received his B.S. degree in mechanical engineering in 1999 and the Mr degree in industrial engineering and engineering management in 2005 from the University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia. He received the Ph.D. degree in industrial engineering and engineering management from the same institution in 2012. From 2001 to 2012, he was a Teaching Assistant and from 2012 to 2017, Assistant Professor at the University of Novi Sad, Faculty of Technical Sciences, Department of Industrial Engineering and Engineering Management, Chair for Information and Communication Systems. Since 2017 he is Associate Professor at the same institution. He is also a vice-dean for Science and International Cooperation at the University of Novi Sad, Faculty of Technical Sciences, and head of Chair of Information and Communication Systems. His research interest includes ERP systems, e-learning systems, e-government systems, data mining, and business process mining in production planning. Darko Stefanovic has published in several international information systems journals. s

Darko Čapko received the Ph.D. degree from the University of Novi Sad, in 2012. He is a Full Professor at Faculty of Technical Sciences, University of Novi Sad and CSO at Ethernal, Novi Sad. He has published over 80 articles and participated in more than 20 projects. His research interests are related to distributed algorithms, cryptography, blockchain and artificial intelligence.

Received: January 27, 2023; Accepted: May 04, 2023.