BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **24**, No 2
Sofia • 2024 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2024-0019

Decentralized Application (dApp) Development and Implementation

Ivan Popchev[1], *Irina Radeva*[2]

[1]*Bulgarian Academy of Sciences, Acad. G. Bonchev Str., Block 2, 1113 Sofia, Bulgaria*
[2]*Institute of Information and Communication Technologies – Bulgarian Academy of Sciences, Acad. G. Bonchev Str., Block 2, 1113 Sofia, Bulgaria*
E-mails: ivan.popchev@iict.bas.bg irina.radeva@iict.bas.bg

***Abstract*:** *This paper focuses on the development and deployment of a dApp* (*decentralized Application*) *for Smart Crop Production Data exchange* (*SCPDx*) *that runs on Antelope blockchain/IPFS infrastructure. The paper emphasizes practical approaches to dApp design and deployment, analyses architectural patterns of dApps, and underlines the role of smart contracts in implementing complex functionality. The paper's contribution is the detailed description of the main smart contracts and the practical knowledge provided on the architecture and implementation of dApps, emphasizing the challenges and solutions in the development process, especially in the context of smart contract implementation. Future developments of the application towards additional data types processing, and design of an interface for leveraging, testing, and evaluating the performance of open source Large Language Models* (*LLMs*) *on specific datasets are commented on.*

***Keywords*:** *dApp* (*decentralized Application*), *Antelope blockchain, IPFS, Decentralization, Smart contracts, Architecture patterns.*

1. Introduction

A decentralized application (dApp) is a software application that runs on a blockchain network and is: 1) *open source* – the source code is available to the public, which means that anyone can inspect, use, copy and modify them; 2) *decentralized and cryptographically secure* – all information on dApp's is cryptographically secure and stored in a public, decentralized blockchain maintained by multiple users (or nodes); 3) *tokenized system* – dApp's can be accessed with a cryptographic token [1]. They can accept cryptocurrencies or generate their own token using a consensus algorithm, such as Proof-of-Work (PoW) or Proof-of-Stake (PoS).

The emergence and widespread of dApps are associated with the development of Web3 technologies. This term was introduced by Gavin Wood [2] and refers to the next generation of Internet built on different decentralized digital technologies such as cryptocurrencies, Non-Fungible Tokens (NFTs), Decentralized Autonomous Organizations (DAOs), decentralized finance [3], supporting technologies such as

Artificial Intelligence (AI), Machine Learning (ML), big data, and others [4, 5]. The evolution of dApps is related to blockchain technology, and the introduction of Smart Contracts (hereinafter referred to as SCs).

The *common element* of dApps can be attributed to SCs, frontend user interface, decentralized storage solution, underlying blockchain, deterministic and isolated execution, and Turing completeness.

A *smart contract* is "A collection of code and data (sometimes referred to as functions and state) that is deployed using cryptographically signed transactions on the blockchain network" [6]. The SCs allow the implementation of embedded business logic and as a consequence the ability to develop complex applications [7], [8], they are executed in the blockchain and serve as backend logic for dApps. The *Frontend user interface* is part of dApp that users interact with. It can be written in any language and is responsible for generating calls to the dApp's backend. *Storage of data* in dApps is typically stored in a *decentralized* manner [9]. The type of data a dApp can store in a blockchain can vary widely depending on specific use cases, but most commonly it is transaction data, smart contract status data, and file metadata. However, it is important to note that storing large amounts of data directly on a blockchain can be expensive and slow due to the limitations of blockchain technology. Therefore, for larger data storage needs, dApps use decentralized file systems such as the InterPlanetary File System (IPFS), Sia, Storj, MaidSafe, Swarm [10]. These systems allow larger amounts of data to be stored in a decentralized manner while leveraging the security and immutability of blockchain.

Blockchain is the underlying technology that drives dApps [11]. The core elements of the blockchain ecosystem ensure the security, performance, and execution of dApps functions. These are SCs, consensus algorithms, and token standards (ERC-20, ERC-721), which define a set of rules that a token must follow within the blockchain. Tokens can represent a variety of assets and are essential for dApps, especially those that involve working with cryptocurrencies or digital assets. Third, interoperability capabilities with other blockchain platforms or external systems. Fourth, digital wallets store users' private keys needed to access their funds and interact with SMs.

All operations in a dApp are *deterministic.* In the context of dApps, deterministic operations refer to the fact that dApps perform the same function regardless of the environment in which they are executed. This means that given the same input, a dApp will always produce the same result regardless of where or when it is executed. The deterministic nature of the operations is critical to maintaining consistency in a decentralized network. Because dApps operate in a peer-to-peer network with many nodes, all nodes must agree on the result of any given operation. This is particularly important for the implementation of SCs, which are the basis of many dApps [12].

This paper is a further development on applications and implementation of blockchain technologies conducted under the National Research Program "Smart Crop Production" 2020-2023. The research and development have been presented in several publications. The aspects of digital risk managing associated with disruptive technologies in agriculture, an overview of blockchain definitions, principles, and its

potential in various industrial sectors, and implications of blockchain in risk management, internal audit, and control procedures in agricultural organizations were presented in [13] and [14]. A multi-criteria decision-making framework for selecting blockchain software in a fuzzy environment [15], a blockchain ecosystem as a network of stakeholders and infrastructure comprising technical and logical elements [16], along with smart contract audit procedures tailored to the functionality and complexity of smart contract code [17] were part of the studied issues.

The development stages and a framework for blockchain/distributed file system platform for the exchange and storage of data and information SCPDx (Smart Crop Production Data Exchange), the deployed prototype and its initial testing, and integration of blockchain oracles were presented in [18-20]. As an underlying blockchain and distributed file system have been selected eosio [21] and IPFS (InterPlanetary File System) [22]. The networks were deployed as private. Later, in 2022, due to the hard fork, eosio blockchain software was renamed to Antelope.io [23], which forced the migration of the blockchain network. The functionality extension and User Interface (UI) enhancement continued with the launch of the biometric authentication software hot wallet supporting tokens/NFT (ERC-20, ERC-721) [24], and a web application developed using open-source solutions [25].

The purpose of this paper is to provide an overview of the different architectural patterns of dApps to enhance understanding of how decentralized applications are structured and function within blockchain technology. To describe the architecture and deployment procedure of the SCPDx dApp on Antelope blockchain/IPFS infrastructure, and to describe the design, structure, functionality, and implementation of smart contracts used within this dApp.

Further, the paper is organized as follows. In Section 2, common architectural patterns for dApps are presented. Section 3 describes an SCPDx dApp architecture and explains the software feathers of dApp deployment. Section 4 is an implementation of the smart contracts. Section 5 is the discussion and conclusions.

2. dApps architecture patterns

The key difference between decentralized and traditional web applications is their underlying architecture. A web application is hosted on centralized servers. In contrast, the decentralized application operates on a decentralized network where control and decision-making are dispersed among multiple autonomous entities rather than centralized in a single authority.

Despite the significant number of dApps, the definition, architectures, and classifications of dApps are still quite generally defined [26]. This section presents a detailed and structured overview of different architectural patterns used in the development and operation of decentralized Applications (dApps).

It is necessary to highlight the difference between a decentralized and a distributed network (Fig. 1).

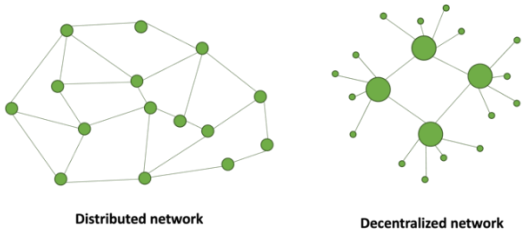

Fig. 1. Distributed and decentralized network configuration

A network configuration in which multiple authority serves as a centralized hub for a subsection of participants is decentralized. If some participants are located behind a centralized hub, its loss hinders communication between them. A network where all participants can communicate with each other without going through a centralized point is distributed. The loss of any participant will not hinder communication. This is also known as a peer-to-peer network [6].

2.1. Peer-to-Peer (P2P) architecture pattern

In a P2P pattern (Fig. 2), each node in the network acts as both a client and a server. This model is fundamental to many dApps, especially those involving file-sharing or communication. The dApp itself does not reside in a single location or node. It is distributed across the network within the participating nodes (peers). Each node contains a copy of the application code, or part of it, and potentially part of the shared data.

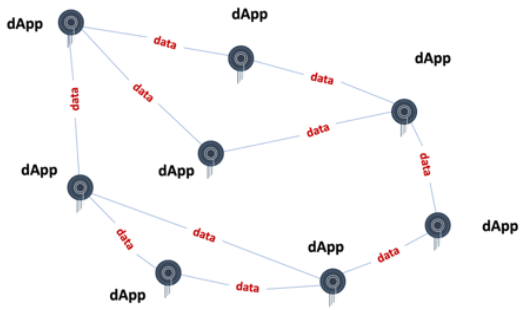

Fig. 2. Peer-to-Peer (P2P) architecture

When a user interacts with the dApp, the required code is executed on its local node (device). This may involve retrieving data from or storing data on their node or communicating with other nodes on the network to perform transactions or access shared data. The state of the dApp (such as user data, transaction records, and application states) is synchronized across the network. This synchronization is typically achieved through the consensus mechanism of the blockchain, ensuring that all nodes have a consistent view of the data.

If the dApp is built on a blockchain it uses SCs. These contracts are executed on the blockchain, affecting all nodes in the network. Nodes communicate directly with each other to share data, confirm transactions or update the dApp's status.

P2P is preferable for use cases such as file sharing (BitTorrent, IPFS), and decentralized messaging dApps (Status, Signal).

2.2. Client-server with decentralized backend

In a client-server architecture with a decentralized backend, the dApp components are distributed between the client-side interfaces and the decentralized (often blockchain-based) backend (Fig. 3). The frontend (client) interacts with the user, while backend operations, typically involving SCs, are executed in a decentralized network.

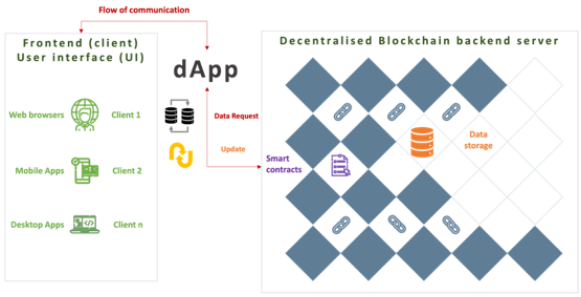

Fig. 3. Client-server with decentralized backend

Users interact with the dApp through a UI, typically deployed on traditional devices such as web browsers, and mobile or desktop applications. Some of the application logic (frontend logic – user interactions and data presentation) resides on the client side. Client-side interfaces communicate with the decentralized backend to retrieve data, transactions, and other operations that need to be recorded in the blockchain.

The core business logic of a dApp is often implemented in SCs on a blockchain platform. The blockchain serves as a decentralized database that stores data (transaction records, user balances, or any other status information) needed by the dApp.

In client-server interaction, the client side sends requests to the blockchain to retrieve or update data. For example, a user action in the UI can trigger a transaction that is executed as a smart contract in the blockchain. In some cases, the client side may receive real-time updates from the blockchain, which are then reflected in the UI.

A client-server architecture with a decentralized backend is often used in decentralized storage services (Storj, Filecoin) and decentralized streaming platforms dApps (Livepeer, Theta).

2.3. Off-chain with on-chain settlement

The dApp is structured to operate across two layers: the off-chain layer for processing and the on-chain layer for final settlement and record-keeping (Fig. 4).

Most of the dApp's processing, including computations and temporary data storage, occurs on the off-chain layer. This can be on centralized servers or distributed networks, depending on the dApp's design. The off-chain layer is used to enhance efficiency and scalability, as it can handle transactions or computations more quickly and cost-effectively than the on-chain layer. The UI, which interacts with the end-users, typically operates at this layer, facilitating a smoother user experience.

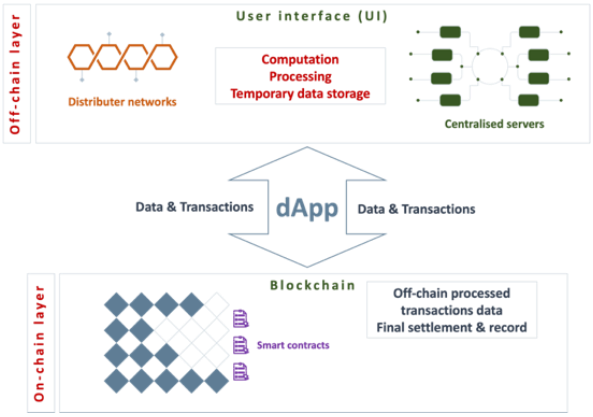

Fig. 4. Off-chain with on-chain settlement

Critical transactions and final states are recorded on the on-chain layer. This includes the conclusive data of transactions processed off-chain. The SCs might be used for managing the rules of settlement and for integrating off-chain data with the blockchain. The on-chain layer ensures security, transparency, and immutability of records, which are key features of blockchain technology.

Data and transactions are periodically transferred from the off-chain layer to the on-chain layer for final settlement. The on-chain layer may also play a role in verifying transactions or data processed off-chain, ensuring the integrity and finality of the dApp's operations. Operational efficiency in this case is achieved by off-chain transaction processing. The security and reliability of blockchain technology are exploited through on-chain settlement. This approach is often used to enhance scalability and reduce costs.

This architecture pattern is applied for scalable payment channels (Lightning Network (for Bitcoin), Raiden Network (for Ethereum)), and decentralized exchange (DEX) dApps (0x, IDEX).

2.4. Hybrid decentralized architecture

Here, the dApp is spread over a blockchain (decentralized component) and traditional servers (centralized component) (Fig. 5). The goal is to balance the benefits of decentralization, such as security and transparency, with the efficiency and scalability of centralized systems. This architecture is particularly useful for complex dApps requiring high performance without compromising the decentralized design ecosystem.

The decentralized components are various SCs, and critical data (e.g. transaction records or user assets) stored on the blockchain. The nodes in the blockchain are involved in maintaining the decentralized aspects of the dApp (transaction validation or consensus building).

Centralized components such as the UI are hosted on centralized servers or cloud services. This includes the visual components and some of the front-end application logic. Parts of application processing and temporary data storage can be handled by centralized servers (computation or data processing that does not require decentralization). Some services, such as notifications or API integration, may be

managed by centralized systems to provide functionality that is not feasible or efficient in a blockchain.

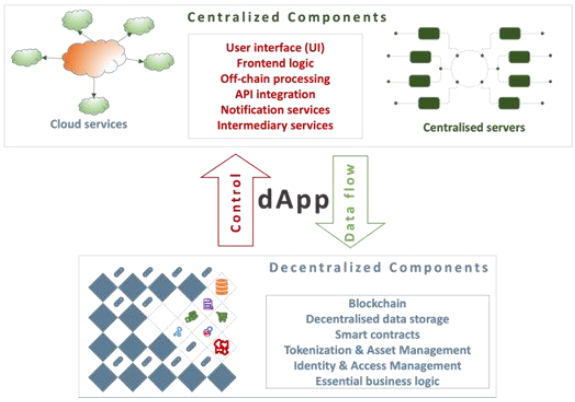

Fig. 5. Hybrid decentralized architecture

The dApp provides interoperability, consistent data flow, and synchronization between the centralized and decentralized components of the architecture. For example, a user action in the UI can trigger a transaction in the blockchain. Certain transactions can start in the centralized system and then be settled or recorded in the blockchain, combining the speed of centralized processing with the security of a decentralized ledger.

This pattern is preferable for social media platforms with decentralized moderation (Minds, Steemit), and supply chain tracking with centralized reporting (VeChain, Hyperledger).

2.5. Fully decentralized (on-chain) architecture

Here, the entire application is hosted and runs on the blockchain. This means all components of the dApp, including UI, business logic, and data storage, are contained within the blockchain network (Fig. 6). The core logic of the dApp is implemented in SCs. All data related to the dApp, such as transaction history, and application state, is stored on the blockchain.

In a fully decentralized architecture, even the UI components can be hosted on-chain, though this is less common due to practical limitations (like blockchain storage constraints and efficiency). More commonly, the UI is hosted off-chain (like on a user's device or a traditional server) but interacts entirely with the on-chain components.

Most use cases in this case are decentralized finance (DeFi) platforms (Uniswap, MakerDAO), decentralized autonomous organizations (DAOs) (Aragon, DAOstack), supply chain management dApps (VeChain, IBM Food Trust), decentralized identity verification systems (uPort, Civic) and decentralized voting systems (Horizon State, Voatz).

The following clarification should be made: the position of a dApp in a P2P and a fully decentralized architecture differs slightly. In P2P, the dApp runs on each participant's device (e.g., computer or smartphone). Each instance of the dApp communicates directly with other instances running on other users' devices. In a fully

decentralized architecture, the application not only runs on individual devices but also integrates deeply into the underlying decentralized network (such as a blockchain). It uses the decentralized network infrastructure for all its operations, including data storage, processing, and decision-making.

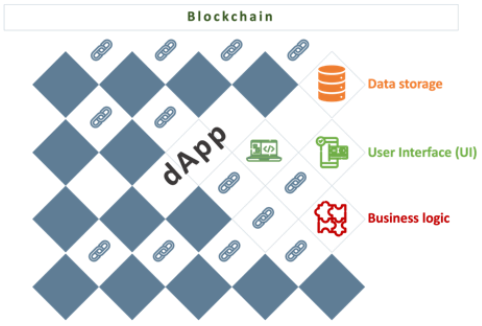

Fig. 6. Fully decentralized (on-chain) architecture

Each of these architectural models has its unique advantages and challenges. The choice of architecture depends on the specific requirements and goals of the dApp – the need for scalability, security, decentralization, or otherwise.

3. SCPDx dApp architecture and deployment

This section presents an overview of the SCPDx dApp with a focus on its architecture, functionalities, and deployment procedure. The dApp is designed for data and information storage and exchange. It is hosted and deployed on private Antelope blockchain and IPFS networks. In this regard, it is necessary to highlight the differences in how a dApp works on private and public infrastructure. Private blockchains are typically deployed on a smaller number of nodes, which in general could result in more efficient operations, faster transactions, and higher throughput as the data load requiring verification is less. The control access rights are regulated by administrators, which could enhance security, but at the expense of a certain level of centralization. Private platforms are well suited for dApps that require strict data access. On the other hand, public blockchains are very widespread and have more support from the open-source community.

3.1. dApp architecture

The goal in the development of SCPDx dApp was a maximum decentralization by hosting on an IPFS environment and implementing the business logic on the blockchain network. This has been achieved through a three-layer architecture: a Presentation Layer (PL), a Business Logic Layer (BLL), and a Data Access Layer (DAL).

PL enables the browser to visualize the individual components of the dApp and provide user access, along with the means to enter information and get data from available sources (available files list, add new files or edit, etc.). Also, in this layer is stored the state management of the dApp, which includes the actual data stored in the

blockchain and in particular in the multi-index tables of the SCs. A standard Anchor blockchain wallet is used to authenticate users as well as to validate users' actions in the dApp [27].

The BLL defines the objects (e.g., the list of transactions in the blockchain, the list of files available in the platform, etc.) as well as the processing methods. It also manages data received from external sources (IPFS) by performing upload/download/edit operations/handling hash/ipns/file metadata, etc. This functionality is mainly implemented based on various smart contract actions.

Access to on-chain data (SCs' multi-index tables), and to off-chain data (IPFS network) is performed at the DAL. The IPFS and blockchain *cannot communicate directly*, and their interaction is done through BL and PL. The blockchain serves as an event log rather than a database, while IPFS functions as a distributed file system. Each technology performs unique functions and is not interchangeable.

In this implementation, the use of a backend server is completely eliminated. This consists of implementing all access and data exchange operations in the PL (frontend part). This approach shortens the assessment time and avoids a single point of failure in dApp operation.

The core logic of the dApp is implemented in four SCs, deployed on the blockchain, and executed autonomously. All data related to user information, transaction history, and application state, is stored on-chain.

The supported dApp's functionalities are: 1) joining request procedure (PL and DAL); 2) users' login digital identity mechanism (PL and BLL); 3) file (upload, download and edit, and related Digital Rights Management (DRM)) mechanism (PL, BLL and DAL); 4) tokenization: assigning file's value in SYS or METEO native tokens (if applicable) (BLL); 5) blockchain/IPFS endpoint testing mechanism: automatic selection of blockchain/IPFS active endpoints (PL); 6) activity logging (PL, BLL and DAL).

3.2. dApp deployment

The general concept of developing and deploying a dApp on IPFS is described in [28]. Here is presented an implementation adapted to current ReactJS.

Normally, all resources and files in a web application are accessed via a URL (Uniform Resource Locator). When using IPFS, the distinctive feature is that the files and components of the application are not addressed by URL (i.e., not where they are located), but according to the fact that they are context-aware. This addressing is called a URI (Uniform Resource Identifier), i.e., a corresponding hash is generated for each file.

The dApp, developed in ReactJS, employs *BrowserRouter* and *HashRouter* for component navigation, each facilitating access to resources via distinct addressing methods. *BrowserRouter* requires server-side request handling and configuration, granting the server full control. In contrast, *HashRouter* manages requests client-side, obviating server configuration and simplifying setup for servers with limited control, like static file servers or IPFS.

In this dApp, an IPFS network replaces a web server, necessitating *HashRouter* due to its similarity to static or limited-configuration servers. The dApp is compiled

using the standard *npm package*, producing a 'build' directory that is added to IPFS, generating a unique hash for directory items. This hash serves as the dApp's address. Updates are managed via IPNS (InterPlanetary Name System), involving key creation and publishing the directory's hash under this key to obtain an IPNS identifier. New versions are uploaded using IPNS and the key, enabling dApp access through this addressing method.

4. An implementation of the smart contracts

DApp uses independent SCs in C^{++}, that implement dApp's functionalities: *"ipfsstore"* for file actions and DRM, tokenization, and activity logging; *"iotoracle"* for file actions, *"transferstat"* for tokenization, and *"gorgo"* for endpoints testing mechanism. The smart contract communication uses the WharfKit library, which simplifies interactions with the blockchain in user-friendly interfaces. The implementation is aligned with the standard EOSIO/Antelope protocol. Fig. 7 shows the interaction between the Antelope/IPFS infrastructure, dApps, blockchain oracles, and external IoT networks, all facilitated by SCs.

4.1. Smart contract "ipfsstore"

This contract interfaces with IPFS to manage file metadata during uploads, downloads, and user edits, primarily aiding in data sharing, digital content distribution, and archiving.

Contract employs:

- singleton structure **"myvar"** for contract settings and multi-index tables **"editstore"** for managing file data and **"activity"** for tracking activities in the platform.
- mechanisms for adding, retrieving, editing, and managing file data.
- modifiers and events handle permissions and state changes.

Functionality and logic:

- includes a flow for file upload, download, edit, and retrieval, with interactions between the **"activity"** and **"editstore"** tables.
- rules for data management, such as file ownership, availability, and edit tracking.
- reads from and writes data to the blockchain, updating file metadata and tracking changes.

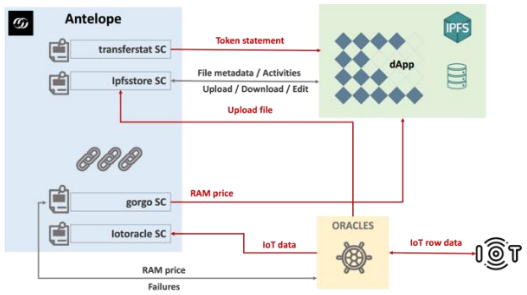

Fig. 7. Interactions enabled by smart contracts

On the EOSIO/Antelope blockchain, CPU and NET are key manageable resources impacting data processing efficiency.

The **“ipfsstore”** smart contract uses multi-index tables with a specific data architecture:

```
id: A unique identifier for each record in the table.
fileid: An identifier for the file associated with the record.
filename: The name of the file.
hash: The IPFS hash of the file.
ipns: The IPNS name of the file.
key: A key associated with the file.
size: The size of the file.
created by: The account name of the user who created the record.
created at: The timestamp when the record was created.
price: The price of the file.
is available: A boolean indicating whether the file is available.
description: A description of the file.
```

The table indices:

```
primary key: The primary index of the table, based on the id field.
by file: An index based on the fileid field.
by_creator: An index based on the created_by field.
third_key: An index based on the created_at field.
available key: An index based on the is available field.
new index: A composite index based on the created by and id fields.
```

Each table record represents an IPFS-stored file, detailing its characteristics and availability, with indices enabling efficient field-based querying.

The **“activity”** multi-index table data architecture:

```
uid: A unique identifier for each record in the table.
actor: The account name of the user who acted.
timestamp: The timestamp when the action was performed.
act The type of action performed. This can be one of the following: 1:
Add file; 2: Get file; 3: Edit file; 4: Delete file; 5: Restore file;
object_id: An identifier for the object associated with the action.
file_id: An identifier for the file associated with the action.
```

The multi-index table indices:

```
primary key: The primary index of the table, based on the uid field.
time key: An index based on the timestamp field.
action key: An index based on the act field.
object key: An index based on the object id field.
actor_key: A composite index based on the actor and uid fields.
file_key: A composite index based on the actor and uid fields.
```

The table’s records detail user actions, associated objects, and files, with indices facilitating efficient, field-based querying. These are key excerpts showcasing standard smart contract elements.

Multi-index table definition:

```
    ```cpp
    uint64 t id;
    std::string filename;
    std::string hash;
    uint64_t size;
    name created by;
    eosio::time point sec created at;
    eosio::asset price;
    bool is available;
    string description;
    uint64_t primary_key() const { return id; }
```
```

```
        uint64_t third_key() const { return created_at.sec_since_epoch(); }
        uint64_t available_key() const {
            uint64 t ret = 1;
            if (!is available) { ret = 0; ...
        ```
```

The **“store”** multi-index table structure is defined, incorporating fields for file data (e.g., “filename”, “hash”, “size”) and custom key accessors (**“primary_key”**, **“third_key”**, **“available_key”**) for indexing and retrieval.

**Multi-index table type definition:**

```
        ```cpp
        name("ipfsregister"),
        editstore,
        eosio::indexed by<
            name("id"),
            eosio::const mem fun<
                editstore,
                uint64_t,
                &editstore::primary key
            >
        >,
        ```
```

The code defines the **“ipfsregister”** multi-index table based on the **“editstore”** structure with indexed keys (**“id”**, **“timestamp”**, **“available”**) for querying and data management.

All SCs in this section adhere to a consistent approach in defining multi-index tables.

Function examples:

```
        ```cpp
        clearallsts(name user) {
            check(has auth(get self()), "You have no authority to clear
documents table!");
            ledger_table activities(get_self(), get_self().value);
            for (auto itr = activities.begin(); itr != activities.end(); )
{
                itr = activities.erase(itr);
                print(".");
        }
        ```
```

The **“clearallsts”** action in the contract is designed to clear all entries in a specified table. *It is used during the smart contract development and testing phase*. It includes a *security check* (**“has_auth”**) to ensure that only authorized users can perform this action, demonstrating how EOSIO/Antelope SCs *handle permissions and table manipulations*.

## 4.2. Smart contract “gorgo”

The contract monitors blockchain network nodes, maintaining their health and efficiency to ensure correct functioning within the network.

Contract Structure:

- defines a **“TABLE activity”** for storing node-related data, including fields like **“id”**, **“chain”**, **“endpoint”**, **“checked_at”**, and **“status”**.
- key actions include **“add”** (for adding new node records), **“clearallact”** (for clearing all activity records), and **“getstat”** (for retrieving the status of nodes).
```

- uses “**require_auth”** for permission checks, though specific modifiers and events are not detailed in the snippet.

Functionality and logic:

- allow for adding new node status records, clearing all records, and querying the status of nodes.
- maintain a record of nodes’ statuses in the network, useful for monitoring and maintenance purposes.
- update the blockchain state with new node information and status changes.

The smart contract interacts with other network management tools or contracts and depends on external inputs for node status information.

The “activity” multi-index table data architecture:

```
    id: A unique identifier for each record in the table.
    chain: The name of the blockchain network.
    endpoint: The endpoint URL of the blockchain network.
    checked_at: The timestamp when the endpoint was last checked.
    status: A boolean indicating the status of the endpoint (true if the
endpoint is active, false otherwise)
```

The multi-index table indices:

```
    primary key: The primary index of the table, based on the id field.
    third key: An index based on the checked at field.
    status_key: An index based on the status field.
```

Each table record represents a blockchain network endpoint, detailing the last check and its status, with indices enabling efficient, field-based querying.

Multi-index table definition:

```
        ```cpp
        TABLE activity {
            uint64_t id;
            name chain;
            std::string endpoint;
            eosio::time point sec checked at;
            bool status;
            uint64 t primary key() const { return id; }
            uint64_t        third_key()        const        {        return
checked_at.sec_since_epoch(); }
            uint64 t status key() const {
                uint64 t ret = 1;
                if (!status) { ret = 0; };
                return ret;
            }
        };
        ```
```

This segment outlines the **“activity”** multi-index table, including fields for node data (e.g., **“id”**, **“chain”**, **“endpoint”**), with the **“primary_key”** method crucial for indexing.

Action definition:

```
    ```cpp
        ACTION add (name creator, name chain, string endpoint, bool status)
{
            require auth(creator);
            activity table activity(get self(), get self().value);
            uint64 t newid = activity.available primary key();
            eosio::time point sec                timestamp               =
time_point(current_time_point());
```
```

```
            activity.emplace( get_self(), [&](auto &e) {
                e.id = newid;
                e.chain = chain;
                e.endpoint = endpoint;
                e.checked at = timestamp;
                e.status = status;
            });
        }
        ```
```

The **“add”** action permits users to insert new records into the **“activity_table”**, showcasing authorization checks (**“require_auth”**), interaction with a multi-index table, and the use of EOSIO's time functions.

**Read action with return value:**

```
     ```cpp
        [[eosio::action]] uint64_t getstat (name actor) {
            require_auth(actor);
            activity table documentTable(get self(), get self().value);
            uint64 t    size    =    std::distance(documentTable.cbegin(),
documentTable.cend());
            print("Size: ", size);
            return size;
        }
        ```
```

The **“getstat”** action retrieves and returns the size of the **“activity”** table, demonstrating how to read data from a table, return a value, and utilize the **“print”** function for debugging or logging.

A typical *use case* involves adding a new blockchain node to the network and routinely updating its status in the contract for network health monitoring.

### 4.3. Smart contract *“iotoracle”*

The contract is tailored for managing IoT device data within a blockchain environment, specifically focusing on recording and handling sensor data.

The contract defines two tables:

- **“t_reg”**: a table to store individual IoT data readings, including fields like **“id”**, **“iot_id”**, **“value”**, **“reg_at”**, and **“type”**.
- **“m_reg”**: a table for more complex or aggregated IoT data, with various fields for different types of measurements and time stamps.

The full contract includes actions to add, update, and query IoT data records, although specific functions are not visible in the provided snippet.

Functionality and logic:

- allows to add of new IoT data records, updating existing records, and query data based on different criteria like time or IoT device ID.
- Manages and authenticates IoT data entries, ensuring accurate and timely recording of IoT measurements.
- updates the blockchain state with IoT data, and handles data transformations or aggregations.

The contract can engage with other contracts within the broader IoT ecosystem and interface with external systems or oracles to obtain real-time IoT data.
```

The “m_reg” multi-index table data architecture:

```
    id: A unique identifier for each record in the table.
    iot_id: An identifier for the IoT device associated with the record.
    st0: A vector of double values representing sensor data from soil
temperature 0 depth.
    st10: A vector of double values representing sensor data from soil
temperature 10 depth.
    st20: A vector of double values representing sensor data from soil
temperature 20 depth.
    st30: A vector of double values representing sensor data from soil
temperature 30 depth.
    mc0: A vector of double values representing mc data from state 0 depth.
    mc10: A vector of double values representing mc data from state 10
depth.
    mc20: A vector of double values representing mc data from state 20
depth.
    mc30: A vector of double values representing mc data from state 30
depth.
    m0: A vector of double values representing m data from state 0 depth.
    m10: A vector of double values representing m data from state 10 depth.
    m20: A vector of double values representing m data from state 20 depth.
    m30: A vector of double values representing m data from state 30 depth.
    ec0: A vector of double values representing ec data from state 0 depth.
    ec10: A vector of double values representing ec data from state 10
depth.
    ec20: A vector of double values representing ec data from state 20
depth.
    ec30: A vector of double values representing ec data from state 30
depth.
    at50: A vector of double values representing atmosphere temperature data
from 50 height.
    at100: A vector of double values representing atmosphere temperature
data from 100 height.
    at150: A vector of double values representing atmosphere temperature
data from 150 height.
    h50: A vector of double values representing atmosphere humidity data
from 50 height.
    h100: A vector of double values representing atmosphere humidity data
from 100 height.
    h150: A vector of double values representing atmosphere humidity data
from 150
```

Multi-index table indices:

```
    primary key: The primary index of the table, based on the id field.
    iot key: An index based on the iot id field.
```

Each table record corresponds to an IoT device, with details about the sensor data at various states. The indices allow querying of the table based on various fields.

A *use case* could involve an IoT device deployed on a smart farm, which transmits soil moisture data to a blockchain. This data is then recorded and accessible for automated decision-making processes within the farm management system.

The following excerpts from the **'iotoracle'** smart contract exemplify common elements.

Multi-index table type definition for “m_reg”:

```
    ```cpp
       name("maritsareg"),
       m_reg,
       eosio::indexed_by<
           name("id"),
           eosio::const_mem_fun<
```
```

```
            m_reg,
            uint64_t,
            &m reg::primary key
        >
    >,
    ```
```

This defines “maritsareg”, a multi-index table for aggregated data with various indexing methods based on the “m_reg” structure.

### 4.4. Smart contract *“transferstat”*

This contract tracks and records token transfer activities within a blockchain network, commonly used for monitoring and auditing token transactions in financial systems, wallets, or applications requiring precise token movement tracking. Leveraging multi-index tables and native functionalities for tokens and timestamps, it defines a **“TABLE ledger”**` to store transfer details such as **“id”**, **“from”**, **“to”**, **“qty”** (quantity), **“memo”**, **“time”**, and **“token”**.

The contract features an **“[[eosio::on_notify(“eosio.token::transfer”)]]”** function to handle incoming token transfer notifications and an action **“clearallsts”** for clearing ledger table entries. It captures token transfer data upon each transfer, maintaining a comprehensive record of sender, receiver, amount, and timestamp. With every token transaction, the contract updates the blockchain state.

Additionally, it interacts with the **“eosio.token”** smart contract to receive token transfer notifications and offers the potential for extension to interact with external data sources.

**The “ledger” multi-index table data architecture:**

```
id: A unique identifier for each record in the table.
from: The account name of the user who sent the asset.
to: The account name of the user who received the asset.
qty: The quantity of the asset transferred.
memo: A memo associated with the transaction.
time: The timestamp when the transaction occurred.
token: The token symbol of the asset transferred.
```

The multi-index table indices:

```
primary_key: The primary index of the table, based on the id field.
time key: An index based on the time field.
from key: An index based on the from field.
to key: An index based on the to field.
token_key: An index based on the token field.
```

Each table record signifies a transaction, encompassing the sender, receiver, quantity, and the token of the transferred asset. The indices facilitate efficient queries of the table across diverse fields. The following excerpts from the **“transferstat”** smart contract illustrate common elements:

Multi-index table definition:

```
    ```cpp
    uint64_t id;
    name from;
    name to;
    eosio::asset qty;
    string memo;
    eosio::time_point_sec time;
    name token;
    uint64_t primary_key() const { return id; }
```

```
        uint64_t time_key() const { return time.sec_since_epoch(); }
        uint64_t from_key() const { return from.value; }
        uint64_t to key() const { return to.value; }
        uint64 t token key() const { return token.value; }
        ```
```

This code snippet defines a multi-index table called a "**ledger"** for storing token transfer data. It incorporates fields such as **"id"**, **"from"**, **"to"**, **"qty"**, **"memo"**, **"time"**, and **"token"**, along with methods for indexing the data.

The multi-index table type definition specifies indexed keys (**"id"**, **"timestamp"**, **"from"**, **"to"**, **"token"**) for the table, facilitating querying and data manipulation.

**Notification handler function:**

```
        ```cpp
        ("eosio.token::transfer")]]  void  depo  (name  from,  name  to,
eosio::asset quantity, string memo) {
            // print("in notify");
            std::string tkn = quantity.symbol.code().to string();
            std::transform(tkn.begin(),  tkn.end(),  tkn.begin(),[](unsigned
char c){ return std::tolower(c);}
        ```
```

The **"depo"** function serves as a notification handler triggered by a **"transfer"** action from **"eosio.token"**. It processes incoming token transfers, converts the token symbol to a lowercase string, and then stores the transfer details in the **"ledger"** table.

A *use case* involves using a blockchain wallet to maintain a ledger of all incoming and outgoing token transactions for user accounts.

The source codes of the SCs are accessible in a GitHub repository [29].

## 5. Discussion and conclusion

The diversity of dApps is characterized by a variety of architectural patterns, each with unique characteristics and mechanisms of operation. The architectural models offer different degrees of decentralization and efficiency. For example, fully decentralized architectures maximize decentralization but may have scalability and efficiency issues. Conversely, hybrid models balance the benefits of decentralization with the efficiencies of centralized systems.

SCs are a critical component in many dApp architectures. They facilitate the automation of business logic, transaction rules, and consensus mechanisms, thereby improving the functionality and security of dApps.

The presented SCPDx dApp is designed with a three-layer architecture (Presentation Layer, Business Logic Layer, and Data Access Layer) that provides storage and exchange of data and information. This structure facilitates user interaction, and data processing, and provides easy data access. Deploying a dApp on a private infrastructure increases security, efficiency, and control. Private blockchains, unlike public ones, can provide faster transactions, higher throughput, and more regulated access rights, which are relevant for applications requiring strict data access and privacy.

The dApp uses IPFS for hash-based hosting and addressing combined with blockchain technology. This approach eliminates the need for a backend server, thus
```

reducing points of failure, but also leverages the IPFS capabilities and event logging feature of the blockchain to optimize data management and availability.

The SCs ("ipfsstore", "gorgo", "iotoracle", "transferstat") are designed to perform functions in the dApp ecosystem. This modular approach increases the efficiency and security of the system, as each contract is focused on a specific set of tasks and works independently. The flexible concept of SCs implies that if new functionalities are needed, actions can be added to them while fully preserving the existing business logic.

To ensure security and efficiency in user authorization and execution of blockchain transactions, a standard Anchor wallet is used to provide a better protection for users' private keys. It is used to replace the custom wallet developed earlier for this purpose [24]. Combined with the WharfKit library, the Anchor wallet enabled faster and easier integration without compromising access security.

To enhance performance in private blockchain and IPFS networks in terms of transaction and operation execution speed [19], the use of a backend server is eliminated. All blockchain and IPFS related operations were implemented in the frontend of the dApp. When working with blockchain, WharfKit is again employed.

To use the IPFS network capability to host the dApp and to modify the source code, it is necessary to consider how the individual components of the application are addressed. This is necessary because in IPFS, each file is characterized by its hash, not by a path as if the application is hosted on a web server. The approach used is described in Section 4.2. dApp deployment.

Not to be ignored is the matter of how to manage and establish permissions for users of SCs and actions within SCs by individual blockchain accounts. The Antelope blockchain allows permissions to be created and used for a given smart contract or detached actions from it. This is achieved through a combination of authorizations (createauth, linkauth) and user permissions (which can be defined at the creation of the blockchain accounts or subsequently) as the user can access all or part of the smart contract resources (all actions). Since users are individually registered by the administrator (i.e., the dApp is not public at this stage), they have full rights to use the SCs. This aspect should be approached carefully when the dApp is made publicly available.

The future development of dApps is expanding the processing of different data types and dividing users into groups with access to target resources (files and datasets). Another direction is the development of an interface for using open-source LLM (Large Language Models) on specific user datasets, testing and evaluating the performance of the various models [30, 31]. This implies the use of SCs to document the process to achieve transparency and traceability of progress and results.

Acknowledgments: This work was supported by the Bulgarian Ministry of Education and Science under the National Research Program "Intelligent Animal Husbandry", Grant agreement No D01-62/18.03.2021 approved by Decision of the Ministry Council No 866/26.11.2020.

References

1. Bogdanov, D. What Are dApps: A 2021 Guide to Decentralized Applications – LimeChain (Accessed 9 January 2024).
   https://limechain.tech/blog/what-are-dapps/
2. Wan, S., H. Lin, W. Gan, J. Chen, P. S. Yu. Web3: The Next Internet Revolution (Accessed 9 November 2023).
   http://arxiv.org/abs/2304.06111
3. Huang, R., J. Chen, Y. Wang, T. Bi, L. Nie, Z. Zheng. An Overview of Web3 Technology: Infrastructure, Applications, and Popularity. – Blockchain: Research and Applications, 2023, 100173.
   https://doi.org/10.1016/j.bcra.2023.100173
4. Wang, Q., R. Li, Q. Wang, S. Chen, M. Ryan, T. Hardjono. Exploring Web3 from the View of Blockchain (Accessed 9 November 2023).
   http://arxiv.org/abs/2206.08821
5. Yehia Ibrahim Alzoubi, Ali Aljaafreh. Blockchain-Fog Computing Integration Applications: A Systematic Review. – Cybernetics and Information Technologies, Vol. **23**, 2023, No 1, pp. 3-37.
6. Yaga, D., P. Mell, N. Roby, K. Scarfone. Blockchain Technology Overview. – Publication NIST IR 8202. National Institute of Standards and Technology, Gaithersburg, MD, 2018, NIST IR 8202.
7. Sumathi, M., S. P. Raja, N. Vijayaraj, M. Rajkamal. A Decentralized Medical Network for Maintaining Patient Records Using Blockchain Technology. – Cybernetics and Information Technologies, Vol. **22**, 2022, No 4, pp. 129-141.
8. Penelova, M. Access Control Models. – Cybernetics and Information Technologies, Vol. **21**, 2021, No 4, pp. 77-104.
9. Wu, B., B. Wu. Smart Contracts and Dapps: From Theory to Practice. – In: Blockchain for Teens: With Case Studies and Examples of Blockchain Across Various Industries. B. Wu, B. Wu, Eds. Apress, Berkeley, CA, pp. 183-227.
10. Anthal, J., S. Choudhary, R. Shettiyar. Decentralizing File Sharing: The Potential of Blockchain and IPFS. – In: Proc. of International Conference on Advancement in Computation & Computer Technologies (InCACCT'23), 2023.
    https://doi.org/10.1109/InCACCT57535.2023.10141817
11. Cai, W., Z. Wang, J. B. Ernst, Z. Hong, C. Feng, V. C. M. Leung. Decentralized Applications: The Blockchain-Empowered Software System. – IEEE Access, Vol. **6**, 2018, pp. 53019-53033.
    https://doi.org/10.1109/ACCESS.2018.2870644
12. Zheng, Z., J. Su, J. Chen, D. Lo, Z. Zhong, M. Ye. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. 2023.
    https://doi.org/10.48550/arXiv.2305.08456
13. Popchev, I., I. Radeva. Decision Making Model for Disruptive Technologies in Agriculture. 2020.
    https://doi.org/10.1109/IS48319.2020.9199962
14. Radeva, I. Blockchains: Practical Approaches. – Engineering Sciences, Vol. **LIX**, 2022, No 1, pp. 3-23.
15. Ilieva, G., T. Yankova, I. Radeva, I. Popchev. Blockchain Software Selection as a Fuzzy Multi-Criteria Problem. – Computers, Vol. **10**, 2021, No 10.
    https://doi.org/10.3390/computers10100120
16. Radeva, I., I. Popchev. Blockchain-Enabled Supply-Chain in Crop Production Framework. – Cybernetics and Information Technologies, Vol. **22**, 2022, No 1, pp. 151-170.
17. Popchev, I., I. Radeva, V. Velichkova. Auditing Blockchain Smart Contracts. – In: Proc. of International Conference Automatics and Informatics'2022, Varna, Bulgaria, 2022.
    https://doi.org/10.1109/ICAI55857.2022.9960058

18. P o p c h e v, I., L. D o u k o v s k a, I. R a d e v a. A Framework of Blockchain/IPFS-Based Platform for Smart Crop Production. – In: Proc. of ICAI'22, Varna, Bulgaria, 2022.
https://doi.org/10.1109/ICAI55857.2022.9960070
19. P o p c h e v, I., L. D o u k o v s k a, I. R a d e v a. A Prototype of Blockchain/Distributed File System Platform. – In: Proc. of IEEE International Conference on Intelligent Systems (IS'22), Warsaw, Poland, 2022.
https://doi.org/10.1109/IS57118.2022.10019715
20. P o p c h e v, I., I. R a d e v a, L. D o u k o v s k a. Oracles Integration in Blockchain-Based Platform for Smart Crop Production Data Exchange. – Electronics, Vol. **12**, 2023, No 10, p. 2244.
https://doi.org/10.3390/electronics12102244
21. Getting Started Guide | EOSIO Developer Docs (Accessed 15 March 2023).
https://developers.eos.io/welcome/latest/getting-started-guide/index
22. IPFS Documentation | IPFS Docs (Accessed 3 April 2023).
https://docs.ipfs.tech/
23. Antelope. GitHub (Accessed 11 January, 2024).
https://github.com/AntelopeIO
24. P o p c h e v, I., I. R a d e v a, M. D i m i t r o v a. Towards Blockchain Wallets Classification and Implementation. – In: Proc. of International Conference Automatics and Informatics (ICAI'23), 2023.
https://doi.org/10.1109/ICAI58806.2023.10339101
25. P o p c h e v, I., I. R a d e v a, L. D o u k o v s k a, M. D i m i t r o v a. A Web Application for Data Exchange Blockchain Platform. – In: Proc. of International Conference on Big Data, Knowledge, and Control Systems Engineering (BdKCSE'23), 2023.
https://doi.org/10.1109/BdKCSE59280.2023.10339770
26. Z h e n g, P., Z. J i a n g, J. W u, Z. Z h e n g. Blockchain-Based Decentralized Application: A Survey. – IEEE Open Journal of the Computer Society, Vol. **4**, 2023, pp. 121-133.
https://doi.org/10.1109/OJCS.2023.3251854
27. Anchor Wallet for Desktop and Mobile | Greymass (Accessed 2 October 2023).
https://www.greymass.com/anchor
28. C h r i s t o p h M i c h e l. Learn EOS Development. – Learn EOS Development (Accessed 11 January 2024).
https://learneos.dev
29. GitHub – Scpdxtest/SCPDx: Blockchain/IPFS – Based Platform for Smart Crop Production Scientific Data Exchange. – GitHub (Accessed 20 June 2023).
https://github.com/scpdxtest/SCPDx
30. P o p c h e v, I., D. O r o z o v a. Towards a Multistep Method for Assessment in e-Learning for Emerging Technologies. – Cybernetics and Information Technologies, Vol. **20**, 2020, No 3, pp. 116-129.
31. P o p c h e v, I., D. O r o z o v a. Towards Big Data Analytics in the e-Learning Space. – Cybernetics and Information Technologies, Vol. **19**, 2019, No 3, pp. 16-24.

Received: 12.01.2024; Second Version: 21.01.2024; Accepted: 07.02.2024