Engineering Blockchain Applications Project Report

Ari Argoud
School of Computing and Augmented Intelligence
Ira A. Fulton Schools of Engineering
Arizon State University
Tempe, Arizona, United States
aargoud@asu.edu

***Abstract*—The following portfolio report details two projects completed for CSE598: Engineering Blockchain Applications. These projects encompass the development of decentralized applications and the exploration of the Ethereum blockchain and Hyperledger Fabric. Project 1 involved the creation of an Ethereum Request for Comments - 721-token-standard smart contract (ERC721). The project explored the minting and transfer of non-fungible tokens (NFTs), and the handling of token metadata. The smart contract was successfully deployed on the Polygon Mumbai Testnet. Project 2 focused on the development and deployment of chaincode for a product records management use-case using the permissioned blockchain system, Hyperledger Fabric. Successful implementation of the chaincode was confirmed through testing in Hyperledger Fabric's dev mode and demonstrated accurate data retrieval and efficient data management. The projects provide insight into the suitability of the discussed blockchain platforms for their implemented usage scenarios.**

I. Project 1: ERC-721-token-standard smart contract

A. Introduction

In the realm of blockchain technology, an ERC-721-token-standard smart contract refers to a self-executing digital agreement that adheres to the ERC-721 standard protocol. To comprehend the implications of ERC-721 NFTs, we must first delve into the structure and functionality of the Ethereum blockchain in contrast with its predecessor, Bitcoin.

Bitcoin, the pioneering force behind the inception of blockchain technology, primarily serves as a digital currency. Bitcoin is considered foundational for introducing the concept of a decentralized network with multi-node transaction validation, and thereby removes the need for a central currency authority. The Bitcoin blockchain is structured around blocks, each containing a header and list of transactions. These blocks are chronologically and cryptographically linked via their headers and collectively form an immutable chain. Bitcoin's scope is somewhat limited, and its main functionality is restricted to handling monetary transactions of its native cryptocurrency.

Ethereum, on the other hand, was designed to subvert the boundaries of simple value transfer protocols. It shares the core aspect of decentralization with Bitcoin, however, Ethereum's blockchain facilitates the execution of programmable scripts, or smart contracts, whose execution costs are paid in its native cryptocurrency, Ether. These smart contracts are self-executing, with the terms of their agreements written directly in code, which allows the platform to offer a wide range of potentially groundbreaking decentralized applications (DApps).

Ethereum is uniquely characterized by its Ethereum Virtual Machine (EVM), which operates as a smart contract runtime and execution environment. Any task that can be computed algorithmically can be run in the EVM, allowing developers to create myriad DApps, ranging from games to decentralized exchanges [1]. The ERC-721 standard enables the development and management of non-fungible tokens (NFTs). Unlike cryptocurrencies such as Bitcoin or Ether, which are fungible and identical to each other, NFTs have distinct attributes and hold different values. NFTs represent unique digital assets, such as digital art, collectibles, or in-game items, and provide verifiable proof of ownership or authenticity [2]. Smart contracts, being self-executing and tamper-proof, form the foundation for the creation, ownership, and transfer of NFTs.

In this project, the functionality of an ERC-721-token-standard smart contract is explored. This encompasses the minting of NFTs, management of token metadata, secure NFT transfers, and the utilization of the Interplanetary File System (IPFS) for decentralized file storage. The contract itself was implemented in the Solidity programming language via the Remix browser-based Integrated Development Environment (IDE) and deployed on the Polygon Mumbai Testnet.

B. Solution

Before beginning development of the contract, some set-up was required. First, the Metamask browser extension was downloaded. Of relevance, the extension includes an Ethereum wallet and injects the Ethereum web3 API into websites' JavaScript context, allowing for decentralized applications to be read from the blockchain. Next, the Metamask wallet was configured to interact with the Polygon Mumbai Testnet, a public, permissionless, and scalable test network for Ethereum-compatible, layer 2 sidechains. The testnet provides a secure and reliable smart contract testing environment without the need to use real assets. The on-chain currency for the selected testnet is MATIC, which was acquired through the relevant faucet [3]. The Metamask extension was then connected to the Remix IDE.

Development of the smart contract began with importing several dependencies from the OpenZepplin library, namely, the `ERC721` contract utility, which provides the basic implementation of the ERC-721 token standard, the `ERC721URIStorage` extension, which handles the storage of metadata for each token, and the `Counters` utility, which is used to manage token IDs. The contract class was defined

as inheriting from `ERC721` as well as `ERC721URIStorage`.

That done, method development could begin. The `tokenName()` and `tokenSymbol()` functions were implemented first. Respectively, these functions should return the given token name and token symbol when called. These implementations were accomplished in much the same way, by storing the token name and token symbol as state variables, then retrieving them using return statements. Both functions were constructed with the `public` qualifier, which allows the functions to be called from outside the contract, and the `view` qualifier, which indicates that the functions do not modify the state of the contract.

Next, using the `Counters` utility, a private state variable, `counter token_Id`, is declared in the main body of the class. Utilizing this counter, a function was constructed which would mint a new NFT when passed a Uniform Resource Identifier (URI) and wallet address. The function increments the counter, mints a new token to the specified address, and sets the token URI for the newly minted token to the passed URI.

After testing the `mint()` function with an incomplete URI, a JSON file containing the metadata for the NFT was created [4]. The metadata consisted of a token name, description, and link to the appropriate token image. This JSON file was uploaded to the IPFS [5]. Following this, the `_burn()` and `tokenURI()` functions were implemented. Both functions were overridden to utilize the `ERC721URIStorage` implementations. The `burn()` function allows for the permanent destruction of a specified NFT, and was thereby constructed with the `internal` function qualifier, preventing external contracts from viewing or calling the function. The `tokenURI()` function returns the metadata of a specified NFT and was constructed with the `public` and `view` function qualifiers discussed earlier.

C. Results

To confirm the working state of the project, a new NFT was minted and transferred to a given wallet address using the `safeTransferFrom()` function provided through deployment of the smart contract in Remix. Additionally, 0.2 MATIC was transferred to the provided address for grading purposes. The smart contract was successful in meeting all the project criteria. The functions returned the expected values, the new tokens were successfully minted, the NFT metadata was correctly referenced, and the NFT was successfully transferred to the given wallet address.

D. Conclusion

This was an individual project, with myself as the sole contributor. Through the development of this project, I gained proficiency in the Solidity programming language, an understanding of smart contracts and their various use cases, and experience interacting with the Ethereum blockchain and using IPFS for decentralized file storage. These skills are likely to prove invaluable for future projects in blockchain development.

II. Project 2: Creating Chaincode for Permissioned Blockchain Systems with Hyperledger Fabric

A. Introduction

In the broader scope of blockchain technology, Hyperledger Fabric is a generalized implementation of the blockchain framework that is hosted by The Linux Foundation. The framework is designed to be a foundation for the development of applications or solutions with modular and configurable architecture [6]. Unlike traditional blockchains, Hyperledger Fabric adheres to a unique infrastructure that divides the transaction process into three discrete steps. The first of these steps is chaincode, which consists of distributed logic processing and agreement, the second is transaction ordering, and the third is transaction validation followed by commitment. This division yields various benefits, such as increased network scalability and performance, while simultaneously enhancing trust and confidentiality [7].

In Hyperledger Fabric, chaincode is written in standard programming languages such as Go, JavaScript, or Java, which can be installed, then instantiated via software development kits (SDKs) or command line interfaces (CLIs) Chaincode is typically hosted by peer nodes. These peers form the fundamental building blocks of the network's infrastructure. Peer nodes are further differentiated into two types; endorsing peers are responsible for simulating and endorsing transactions, while committing peers validate these transactions and write them to the ledger.

In addition to peers, the Fabric architecture includes orderer nodes that handle consensus and order transactions into blocks, as well as membership service providers (MSP) that manage identity and permissions in the network.

It is worth noting that Smart contracts, in the context of Hyperledger Fabric, refer to parts of the chaincode which constitute the specific business rules agreed upon by the network's involved parties. Smart contracts are invoked by applications to interact with the ledger, while chaincode represents the entirety of the code uploaded to the network.

This project involves setting up a private blockchain testing environment using Hyperledger Fabric, and from there, developing and deploying chaincode for a product records management use-case.

B. Solution

Before operating Hyperledger Fabric, certain prerequisites need to be installed on the intended platform. The Linux distribution used was Ubuntu 20.04. Docker, Docker Compose, Node.js, npm, Go Programming Language, and Python were installed.

Following the successful installation of these prerequisites, the subsequent step involved downloading and installing Hyperledger Fabric. A script is provided by the documentation to aid in this process. The script downloads and installs samples and binaries, and tags Docker images as latest in the local Docker registry.

To begin, a suitable location on the machine was selected for the `fabric-samples` repository. Curl and bash were used to download the script from a GitHub repository and

then execute that script in a bash shell. Upon executing the script within the directory, several operations were performed:

- A clone of the `hyperledger/fabric-samples` repository was created.
- The correct version tag was checked out.
- The platform-specific configuration files and binaries were installed into `fabric-samples/config` and `fabric-samples/bin`.
- The Docker images for Hyperledger Fabric were downloaded according to the specified version.

The `PATH` environment variable was then updated to include this directory [8].

Before beginning development, it is advantageous to set up Hyperledger Fabric's dev mode. In the context of Hyperledger Fabric, dev mode refers to a development mode where chaincodes are built and started by the user, rather than the peer, allowing for faster code/build/run/debug cycles. To set up dev-mode, navigate to the `fabric-samples/chaincode-docker-devmode` directory and start the network by running `docker-compose up` with the `SingleSampleMSPSolo` orderer profile. This will launch the peer in dev mode, as well as a chaincode environment container, and a CLI container. Access the chaincode docker container from another terminal, and once inside the chaincode's directory, compile and start the chaincode with node.js or go. Next the code must be installed and instantiated via the CLI docker. This done, the chaincode should now be interactable on the network. New chaincodes requiring testing should be added to the `chaincode` subdirectory and will necessitate that the network be relaunched [9].

With the setup completed, development began in earnest. The primary code base for the project was written in JavaScript and tested using node.js. The chaincode was comprised of several key components: a `DeviceRecord` class that represents the device, a `DeviceRecordList` class for managing the state of the device records on the ledger, a `DeviceRecordContract` class, which defines the transactions that interact with the ledger, and a JSON file, `companyIndex`.

First, the `getDeviceByKey()` function from the `DeviceRecordContract` class was modified to read a record by its key. This was achieved by utilizing the `getDRecord()` function from the `DeviceRecordList class`, and allowed for the retrieval of a specific device record from the ledger using only its unique key. This asynchronous function modification utilized the `await` operator, which is used throughout the code base to prevent race conditions when interacting with Hyperledger contexts.

Next the `DeviceRecord` class was expanded to include a `last_update` field with correspondent getter and setter methods. This served to track the last update timestamp of a device record, and more generally, would provide the basis for monitoring the changes made to each device record. This was complemented by the implementation of the `updateLastUpdate()` function in the `DeviceRecordContract` class, which was designed to update the `last_update` field of a device record whenever a modification was made.

The `queryByCompany()` function was added to the `DeviceRecordContract` class in order to enable the retrieval of all device records associated with a specific company. The function begins by constructing a JSON CouchDB selector query string [10]. The selector field of the query is designed to match the company field of the device records with the company parameter supplied to the function. The `use_index` field in the query string is set to `companyIndex`, directing CouchDB to utilize this specific index for the query execution. Once the query string is assembled, it is passed to the `queryWithQueryString()` method. This method executes the query against the ledger and retrieves the device records that belong to the specified company.

A JSON index, `deviceTypeIndex`, was created for the `device_type` field. This was followed by the addition of the `queryByDevice_Type_Dual()` function to the `DeviceRecordContract` class. This function was intended to enhance the ledger query functionality by allowing users to fetch device records filtered by two different device types. The function leverages the CouchDB selector query feature in combination with `deviceTypeIndex` and is differentiable from the `queryByCompany()` function in that it uses the `$in` operator within the `device_type` field selector, which allows it to match the `device_type` field of the device records with either of the two device types provided as parameters to the function. The `use_index` field in the query string is set to `deviceTypeIndex`, which instructs CouchDB to use this specific index when performing the query.

Finally, error handling for unknown transactions in the `DeviceRecordContract` class was implemented. The `unknownTransaction()` method was designed to throw an error stating "Function name missing" in response to invocations of transactions that do not exist in the contract.

C. Results

The chaincode was tested using the Hyperledger Fabric's dev mode, as well as the auto-grader. This entailed deploying the chaincode to the local network and invoking the transactions from the CLI. The testing involved creating new device records, retrieving them by key, updating their attributes, and querying them by company and device type. All transactions returned the expected results, confirming the successful implementation of the chaincode.

D. Conclusion

This was an individual project. A code base was provided through the course, but I am the sole contributor with respect to the modifications and implementations discussed in this report. This project provided hands-on experience in setting up a private blockchain network with Hyperledger Fabric and in developing and deploying chaincode for a product record management use case. The tasks completed in this project

emphasized the importance of data integrity, precise data retrieval, and efficient data management in blockchain applications. The experience gained in this project is integral for developing more complex blockchain solutions.

REFERENCES

[1] [1] V. Buterin, "Ethereum whitepaper," ethereum.org, https://ethereum.org/en/whitepaper/ (accessed May 10, 2023).

[2] D. S. William Entriken (@fulldecent), "ERC-721: Non-Fungible token standard," Ethereum Improvement Proposals, https://eips.ethereum.org/EIPS/eip-721 (accessed May 10, 2023).

[3] "Mumbai faucet," Mumbai Faucet, https://mumbaifaucet.com/ (accessed May 10, 2023).

[4] "Introducing json," JSON, https://www.json.org/json-en.html (accessed May 10, 2023).

[5] "Free decentralized storage and bandwidth for nfts on ipfs & filecoin.," NFT.Storage - Free decentralized storage and bandwidth for NFTs on IPFS & Filecoin., https://nft.storage/ (accessed May 10, 2023).

[6] An overview of Hyperledger Foundation, https://www.hyperledger.org/wp-content/uploads/2021/11/HL_Paper_HyperledgerOverview_102721.pdf (accessed May 11, 2023).

[7] Hyperledger architecture, volume 1, https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf?ref=grid-dynamics-blog (accessed May 11, 2023).

[8] "Getting started¶," hyperledger, https://hyperledger-fabric.readthedocs.io/en/release-1.4/getting_started.html (accessed May 11, 2023).

[9] "Chaincode for developers¶," hyperledger, https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html (accessed May 11, 2023).

[10] "Using couchdb¶," hyperledger, https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_tutorial.html (accessed May 11, 2023).