Swinburne University of Technology

Ho Chi Minh Campus

Decentralised Trading System

Project Design Document

COS30049 - Computing Technology Innovation Project

Project Group 3TL:

1. Nguyen Do Nhat Nam - 104061616
2. Nguyen Quoc Thang - 104193360
3. Tran Thanh Minh - 103809048

March 2024

Word Count: 2088 words

Table of Contents

Project Information

Motivation

There are several possible drawbacks to the conventional banking system that may cause discomfort for customers. They seize possession of your belongings, have the ability to restrict transactions, freeze accounts, and even influence the market. Furthermore, these systems have a low degree of transparency, which makes it challenging to understand market trends or the provenance of assets. In addition, a lot of individuals worldwide are unable to access traditional financial markets due to economic, regulatory, and geographic constraints. Lastly, one of the biggest obstacles is the exorbitant transaction expenses, which include spread, transaction, and transfer fees. (Guneet, 2023).

Advantages of the system

Decentralised exchange (DEX) systems address the shortcomings of conventional systems and provide a new lease on life for the financial industry. DEX gives you the ability to seize total control of your possessions. The private key belongs to you, therefore you may handle assets independently of third parties.

DEX guarantees high transparency as well. The blockchain permanently records every transaction, which makes it simple to trace and verify the provenance of assets. In addition, DEX eliminates regional barriers. Anyone may engage in the market with simply an internet connection, no matter where they are in the globe.

DEX furthermore considerably reduces transaction expenses. DEXs do away with the need for middlemen by automating the trading process with smart contracts, which drastically lowers expenses. Lastly, compared to traditional markets, DEX offers a wealth of fresh investing alternatives with larger return potential. (Wood, 2022).

Information

The Decentralised Trading System seeks to provide a peer-to-peer trading platform for a variety of assets that is safe, open, and effective. By giving consumers more control over their assets, it removes the need for middlemen, which might lower trading costs and increase user confidence in the trading system. The goal of the system is to completely transform the conventional trading environment by providing a more transparent, safe, and user-friendly substitute.

The following are the main features that the system provides:

- Order Matching: An automatic system that compares user-placed purchase and sell orders according to preset amounts and prices.
- Escrow Service: A safe system that retains assets until a deal is closed, guaranteeing the security of each party taking part in the deal.
- Dispute Resolution: A comprehensive mechanism that maintains fairness and confidence in the decentralised market by resolving disputes between traders.
- Secure Transactions: To ensure the security and immutability of transactions and thwart unwanted changes or manipulations, the system makes use of blockchain technology.

Team information

Our project implementation members include:

- *Nguyen Do Nhat Nam* : responsible for implementing and developing the project's user interface, ensuring that the created user interface is easy to use, eye-catching and suitable for users.
- *Nguyen Quoc Thang* : Assume a significant part in the project's cyber security and backend software development.
- *Tran Thanh Minh* : has in-depth knowledge about blockchain so he is in charge of the blockchain part. He contributes to the construction and development of smart contracts. His task is also to check and minimise vulnerabilities in the system

Version Information

- *Smart Contract Version* : 0.8.24 Solidity
- *Dependencies Version* :

- Python == 3.12.1
- Djangorestframework == 3.14.0
- djangorestframework-simplejwt == 5.3.1
- Web3 == 6.15.1
- ReactJS == 18.2.0
- axios == 1.6.5
- tailwindcss == 3.4.1

Executive Summary

The main specifications for the smart contracts in a decentralised trading system that is now being developed are described in this paper. The project employs a blend of technologies in order to accomplish its objective:

- The user interface that enables users to engage with the trading system is built using the ReactJS technology. Tailwind CSS offers ready-made utility classes for effective interface styling.
- Python's Django framework manages server-side functionality on the backend and uses specific Python modules to enable possible interaction with the selected blockchain network.

Given that the project makes use of smart contracts, it is reasonable to believe that transaction security and immutability are crucial. To facilitate safe peer-to-peer trade in the decentralised system, smart contracts will be created.

It looks that the project is either in the planning stage or early development stage based on the technologies used. In order to establish a trustless trading environment where users may conduct transactions directly without depending on a centralised authority, this brief highlights the significance of a decentralised method.

A number of crucial actions must be taken in order to proceed. First and foremost, it is essential to define precisely the features and functions that the smart contracts will manage. Second, choosing the right Python library is crucial for a smooth connection with the selected blockchain network. For a seamless user experience, it is crucial to ascertain how the ReactJS-built user interface will communicate with the backend system and the smart contracts.

It's critical to keep in mind that the information supplied on the technologies utilised forms the basis of this overview. These criteria heavily rely on the supposition that smart contracts be utilised for decentralised trading.

ABI Illustration

For Decentralised Trading Systems, the Application Binary Interface, or ABI, establishes guidelines and conventions for inter-software communication. It guarantees that parts and operations may communicate and receive data with accuracy, completeness, and thoroughness. The ABI's code is created with readability, maintainability, and a clear structure in mind, especially true of the manner in which the ABI mandates compatibility and interoperability with other components.

Application Binary Interface (ABI) Properties

Application Binary Interface, or ABI for short, describes how software communicates with the system and with each other. The following are ABI's primary attributes (Chaiken, 2022):

1. Data Types: The ABI describes the representation of fundamental data types, including characters, real numbers, integers, and pointers. This guarantees correct data interchange between programs.

2. Function Calling: The ABI specifies how calls to functions are made between programs. This covers return values, data types for arguments, and function naming standards.

3. Linking: An executable program's linkage to libraries and other code objects is specified by the ABI. File formats, symbols, and version details are all included in this.

4. Memory use: ABI specifies how memory is allocated and released, as well as how applications may use it.

5. Exception Handling: ABI specifies how errors like page faults and memory access violations are handled.

6. Interrupts and Signals: ABI specifies how operating system interrupts and signals are handled by applications.

7. Multithreading: ABI specifies how multithreading is implemented in applications, including the creation and management of threads.

8. Support for Operating Systems: Every operating system has a unique ABI. Programs designed for one operating system might not necessarily execute on others because various operating systems have distinct ABIs.

9. Compatibility: Applications built for an earlier ABI version should be able to operate on computers running a more recent version of ABI if ABI is backward compatible.

10. Performance: ABI is made to maximise speed and efficiency so that applications may operate rapidly.

Furthermore, ABI may have other attributes like:

- Support for sophisticated data structures and data types.
- Support for particular APIs and libraries.
- Support with security features.

ABI is an essential component of the system that guarantees correct and effective communication between programs and the system.

Our projects include ABI: ABI for Transaction and ABI for Transaction Factory.

ABI for Transaction :

This ABI includes:

- Constructor: this function is deployed when the contract is deployed and it is used to call the initial transaction information of the contract. The constructor has parameters “_sender” and “_receiver” representing the recipient and sender in the transaction.
- Next the contract defines an event called “TransactionCompleted” which will be triggered when the transaction is completed and notifies all parties that the transaction was successful.
- In the Transaction ABI there are also a number of functions used to define methods that users or other contracts can call or use to interact with the smart contract. Some functions you can see in the image below are:
  + “*amount*” : used to return the current amount of money in the wallet
  + “*complete*” : returns the completion status of a transaction, used to check whether the transaction process is completed or not
  + *"manager"* : used to return the address of the transaction manager
  + *"receiver"* : used to return the recipient's address

+ "*returnInformation*": used to return transaction information along with some information such as: amount, address, ...
+ "*send*" is used to send money from the intermediary to the contract
+ "*withdraw*" is used to refund money that has been sent from an intermediary to the sender

Below is a photo of part of our Transaction ABI :


```
{
  "abi": [
    {
      "inputs": [
        { "internalType": "address", "name": "_sender", "type": "address" },
        { "internalType": "address", "name": "_receiver", "type": "address" }
      ],
      "stateMutability": "payable",
      "type": "constructor"
    },
    {
      "anonymous": false,
      "inputs": [
        {
          "indexed": false,
          "internalType": "address",
          "name": "sender",
          "type": "address"
        },
        {
          "indexed": false,
          "internalType": "address",
          "name": "receiver",
          "type": "address"
        },
        {
          "indexed": false,
          "internalType": "uint256",
          "name": "amount",
          "type": "uint256"
        },
        {
          "indexed": false,
          "internalType": "uint256",
          "name": "timestamp",
          "type": "uint256"
        }
      ],
      "name": "TransactionCompleted",
      "type": "event"
    },
    {
      "inputs": [],
```


Image 1 : API of Transaction

ABI for TransactionFactory:

Below are a few functions in Transaction Factory :

- “*createTransaction” :* to create a transaction with a set amount and send it to the specified address.
- "*deployTransaction*": used to retrieve the transaction address based on the transaction's index, to be able to track the transaction status and check whether the transaction is completed or not
- "*getDeployedTransactions*" : is used to get and show all the lists that have been transacted on the system and then it deploys to the user interface for the user. It is also used to manage and track the status of transaction lists

Below is an image snippet of part of the Transaction Factory ABI :


```
{
  "abi": [
    {
      "inputs": [
        { "internalType": "uint256", "name": "_amount", "type": "uint256" },
        { "internalType": "address", "name": "_receiver", "type": "address" }
      ],
      "name": "createTransaction",
      "outputs": [],
      "stateMutability": "payable",
      "type": "function"
    },
    {
      "inputs": [{ "internalType": "uint256", "name": "", "type": "uint256" }],
      "name": "deployedTransactions",
      "outputs": [{ "internalType": "address", "name": "", "type": "address" }],
      "stateMutability": "view",
      "type": "function"
    },
    {
      "inputs": [],
      "name": "getDeployedTransactions",
      "outputs": [
        { "internalType": "address[]", "name": "", "type": "address[]" }
      ],
      "stateMutability": "view",
      "type": "function"
    }
  ],
```


Image 2 : ABI for Transaction Factory

Call Graph Discussion

We use extension vscode solidity auditor for generating out the call graph of our contract (Consensys, 2023)

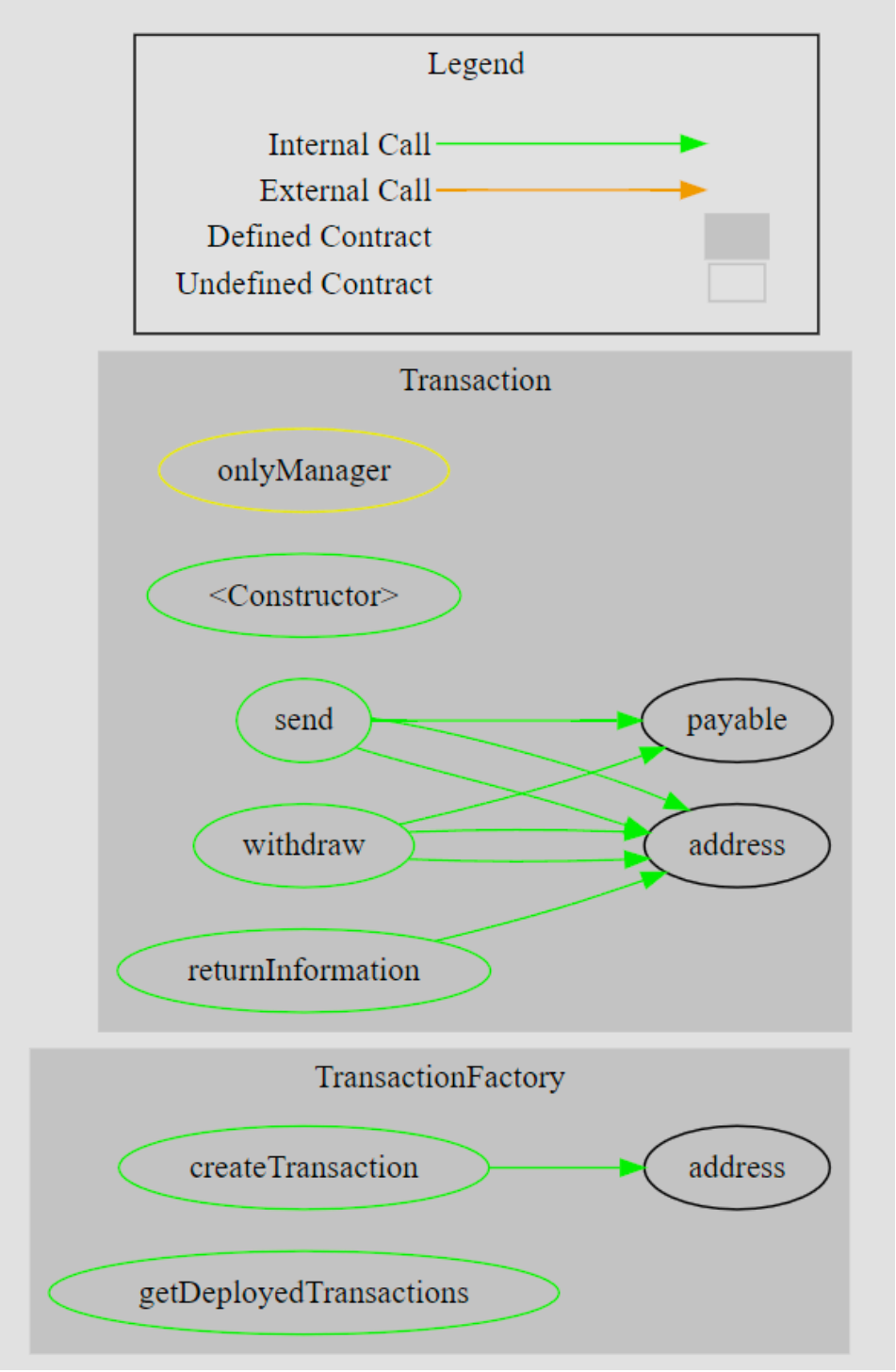

Image 3: Call graph

Calling logic

TransactionFactory contract:

- *createTransaction* is the entry point, initiating new Transaction contracts

It processes in order at first, it validates the transaction amount, then it ensures that sufficient Ether is sent. After That, it deployed a new Transaction contract by passing the sender and receiver addresses along with the transaction amount.

- *getDeployedTransactions* provides a read-only list of deployed transactions.

Transaction contract:

1. *onlyManager* modifier restricts access to *send*, *withdraw*.

It only allows the one who makes the transaction to execute the function with that contract.

2. *constructor* sets up initial transaction details.

It sets up details like sender, receiver, amount, completion status and timestamp.

3. *send* executes the transaction, transferring funds to the receiver

This function allows the transaction manager to execute the transaction by transferring funds to the receiver. It enforces checks to ensure the transaction isn't already completed and that there are sufficient funds within the contract.

4. *withdraw* allows the manager to reclaim funds if the transaction is not completed

This function empowers the manager to reclaim any remaining funds in the contract if the transaction hasn't been completed. Similar checks for completion and fund availability are implemented here.

5. *returnInformation* offers read-only access to transaction details.

This read-only function provides transparency by allowing anyone to view transaction details like sender, receiver, amount, completion status, timestamp, and the current contract balance.

Interrelationships and Core Features

Transaction Creation and Management:

1. TransactionFactory.*createTransaction* and Transaction constructor are central for establishing new transactions.
2. *onlyManager* modifier safeguards key actions within Transaction.
3. The interplay between TransactionFactory and Transaction is essential for managing transactions.

Fund Transfer and Withdrawal:

- Transaction.send and Transaction.withdraw facilitate fund movement and recovery.

This design promotes a clear separation of concerns between transaction creation and individual transaction management. By examining the call graph (once provided), we can delve deeper into potential external interactions, function dependencies, and how these relationships influence the overall security, efficiency, and reusability of the contract.

Inheritance Graph Discussion

Image 4: Inheritance Graph

Inheritance Relationships

The inheritance diagram doesn't exhibit traditional inheritance relationships between the two contracts and there is no indication of one contract directly inheriting from another. The TransactionFactory contract serves as the base contract, even though there's no explicit inheritance declaration in the Transaction contract. In this project there is no inheritance contract because we only use the TransactionFactory to create Transaction instances.

Code Reusability

Although there is no inheritance, code reusability can still be achieved through contract instantiation. The TransactionFactory contract allows for the creation of multiple instances of the Transaction contract. This promotes code reuse by enabling the deployment of multiple transactions with the same logic.

Function Visibility and Access Control

Both contracts use public visibility modifiers for most functions and state variables. This means anyone can interact with them unless restricted by modifiers.
The Transaction contract employs the *onlyManager* modifier to restrict access to certain functions. While not directly related to inheritance, the visibility and access control of functions are crucial aspects. Developers and auditors can examine the functions affected by this modifier and understand their access permissions within the derived contracts.

Modifier Application

The *onlyManager* modifier is applied to *send* and *withdraw* functions within the Transaction contract. This modifier restricts these functions to be called only by the address stored in the *manager* state variable.

Reference

Consensys (2023). *GitHub - Consensys/vscode-solidity-auditor: Solidity language support and visual security auditor for Visual Studio Code*. [online] GitHub. Available at: https://github.com/Consensys/vscode-solidity-auditor [Accessed 26 Mar. 2024].

Chaiken, A. (2022). *A 10-minute guide to the Linux ABI*. [online] Opensource.com. Available at: https://opensource.com/article/22/12/linux-abi [Accessed 1 Apr. 2024].

Wood, J. (2022). *Why Decentralized Exchanges Are Important in the Crypto Economy*. [online] Coindesk.com. Available at: https://www.coindesk.com/tech/2022/03/10/why-decentralized-exchanges-are-important-in-the-crypto-economy/ [Accessed 2 Apr. 2024].

Kaur, G. (2023). *What are decentralized exchanges, and how do DEXs work?* [online] Available at: https://cointelegraph.com/learn/what-are-decentralized-exchanges-and-how-do-dexs-work [Accessed 2 Apr. 2024].