University of St. Gallen – School of
Management, Law, Social Sciences,
International Affairs, and
Computer Science (HSG)

MACoin Blockchain:

A Python-Based Cryptocurrency

Documentation

Authors:

Marko Cvijic
18-607-903

Robert Josua Michels
18-615-286

Anne Strunz
22-601-595

Julia Vogt
16-933-582

Skills: Programming with Advanced Computer Languages (7,789 | 8,789)

Mario Silic, PhD
Institute of Information Management
University of St. Gallen

Submitted on May 24, 2024

Table of Contents

1 Purpose

This MACoin group project aimed to develop a basic blockchain simulation to provide practical insights into how blockchain technology operates, particularly in the context of cryptocurrency transactions. The objective was to build a functional prototype that would allow for the creation and management of digital wallets, the execution of secure transactions, and the ability to track these transactions through a ledger system. Additionally, the prototype provides the basic infrastructure to implement smart contracts. This simulation was designed to demystify the technology behind cryptocurrencies and offer a clear, hands-on experience of its potential applications.

The team comprised students at various skill levels, from beginners to more experienced coders, which enriched the learning experience. Those with advanced skills shared their knowledge, helping to elevate the group's overall understanding and capabilities. This collaborative environment not only boosted the collective Python programming skills but also fostered a spirit of teamwork and peer-to-peer education.

The team’s motivation for choosing this project was driven by a collective curiosity about blockchain technology and a desire to strengthen our technical skills in a highly relevant area. Recognizing the growing importance of blockchain and cryptocurrencies in the digital economy, the decision was made to align the team’s skills with these trends. By building this blockchain simulation, a foundational understanding was sought that could support future projects and professional opportunities in technology and finance.

2 Setup

As a prerequisite for the proper functioning of the Python-based MACoin project, it is crucial to first install the subsequent Python libraries:

`hashlib`

The *hashlib* module is integral to the project, providing a variety of cryptographic hashing algorithms necessary for generating secure block hashes to maintain chain integrity.

`datetime`

The *datetime* library is utilized to handle date and time objects, crucial for accurately timestamping blocks and transactions within the blockchain.

secrets

Secrets is used to generate cryptographically strong random numbers, which are vital for creating secure wallet addresses and recovery phrases.

pandas

Pandas is employed for its powerful data manipulation and analysis capabilities, particularly useful in managing dataframes for transactions and wallet balances.

IPython: display, HTML

This library is leveraged to refine the user interface, facilitating testing and debugging through its sophisticated display functionalities for dataframes and HTML content.

requests

The *requests* library enables HTTP communication with external APIs, crucial for fetching real-time data such as currency exchange rates.

matplotlib.pyplot

Matplotlib is used for creating visual representations of the cryptocurrency's price development, enhancing the interpretability of financial data.

yfinance

This library is utilized to download historical market data from Yahoo Finance, providing essential data for analyzing currency price developments relative to the Swiss Franc to which the MACoin token price is pegged at a ratio of 1:1.

graphviz: digraph

Graphviz is implemented for its graph visualization capabilities, which are essential for depicting the blockchain's structure and understanding the flow and linkage between blocks.

3 Architecture

The MACoin system is composed of five distinct classes, each detailed alongside their respective functions in the subsequent sections. Furthermore, additional functions for managing financial data and their user-interactive visualization will also be examined hereinafter.

3.1 Class: Wallet

The *Wallet* class serves as a secure digital vessel for users' currency and enables the management and execution of transactions on the MACoin blockchain. It is initialized with the

attributes of address, password, recovery phrase, creation date and starting balance. This setup ensures that each wallet is uniquely identifiable for participating in transactions. In total, this class contains two main functions for managing the wallet balance:

`def add_amount(amount)`

This function increases the wallet balance. It is called when the wallet is the recipient of a transaction and adds the specified amount to the wallet's current balance, updating the “balance” attribute accordingly.

`def deduct_amount(amount)`

Conversely, the wallet balance is reduced if there is enough money in the wallet. Thus, this function checks whether the wallet balance is sufficient for the transaction before the specified amount is deducted. This is crucial to prevent overdrafts and ensure that the wallet balance never turns negative.

3.2 Class: Block

The *Block* class represents an individual record on the MACoin blockchain, containing a single transaction and the associated data, which is vital to maintaining the integrity and continuity of the blockchain. As such, it is initialized with a timestamp, data, the hash of the previous block, and an empty list for transactions, ensuring that each block is securely linked to its predecessor to create a tamper-proof chain. The class implements the following method:

`def calculate_hash()`

This function generates a unique hash for the block. This involves combining the block’s timestamp, data, previous hash and transactions into a string and then applying a SHA256 hash to create a unique identifier. This hash serves as a cryptographic seal that protects the block from tampering.

3.3 Class: Smart_Contract

The *Smart_Contract* class defines the attributes and behaviors of individual smart contracts, focusing on their structure and conditions. This class provides the foundation for how contracts are created, detailed, and executed, while the Blockchain class will later outline how these contracts are managed and integrated within the broader blockchain ecosystem.

`def show_SC_terms()`

The **`show_SC_terms`** function retrieves and prints the standard terms and conditions of a smart contract from the *standard_terms* list. This ensures that users are informed about the predefined contractual obligations and conditions.

`def explain_conditions(self):`

This function provides an overview of the structure and requirements for various smart contract types. It explains how the blockchain automatically processes “funding” and “transaction” contracts based on specified conditions, while “other” contracts necessitate manual checking and execution.

3.4 Class: Blockchain

The *Blockchain* class acts as the backbone of the MACoin project, orchestrating the overall management of the blockchain. Upon instantiation, the class initializes the first block on the chain - the so-called genesis block - and sets up a registry for wallets and transactions. This ensures that the blockchain is operational and has mechanisms for proper transaction processing. In addition, a comprehensive set of 27 functions is implemented for managing blocks, wallets, transactions and smart contracts while also allowing for visual display functionalities.

a) Block Management

`def create_genesis_block()`

Starting the blockchain, the function `create_genesis_block` establishes the first block, known as the genesis block, setting a foundation with a timestamp, labeled “Genesis Block” and a hash of zero. This block serves as the immutable anchor for all subsequent blocks added to the chain, ensuring the integrity of the blockchain from its inception.

`def add_block(transaction, data)`

This function appends a new block to the blockchain by capturing the specified transaction details and linking it to the preceding block through a hash reference. This ensures chronological integrity and security of the blockchain, making each addition a tamper-proof continuation of the ledger.

b) Wallet Management

`def create_wallet(testing=False)`

The `create_wallet` function prompts the user to enter a password and uses a combination of randomly selected words to generate a unique recovery phrase. Thereby, a new wallet is set up with a unique address, a starting balance and a recorded creation date, ensuring that each wallet is uniquely identifiable and secure for transaction management. If the testing parameter is set to *True*, the function will skip the password entry step in order to streamline the blockchain testing process.

`def authenticate_user(address=None, password=None)`

The **`authenticate_user`** function verifies the identity of a user by checking their wallet address and password. If either the address or password is not provided, the function prompts the user to input the missing information and validates it against the stored wallet details, raising an error if the authentication fails.

`def change_password(address)`

This function allows users to update the password of a specific wallet to enhance security. It uses the **`authenticate_user`** function to verify the existing password before permitting the user to set a new one, ensuring that only an authorized person can change the wallet credentials.

`def recover_wallet(address)`

The function enables users to regain access to their wallet using a pre-established recovery phrase. After verifying the phrase, it prompts the user to set a new password, effectively restoring secure access to the wallet on the blockchain.

`def delete_wallet(address)`

With this function, users can permanently remove their wallet from the MACoin blockchain, ensuring that the user data is securely deleted. It uses the **`authenticate_user`** function to validate the user's credentials before deletion, preventing unauthorized access and confirming the owner's intention.

`def get_wallet_balance(address)`

This **`get`** function queries the blockchain's wallet registry using a specific address to retrieve and return the current balance of the wallet. It ensures that users have accurate and up-to-date financial information required to manage transactions and monitor asset status.

`def get_wallets_overview()`

The **`get_wallet_overview`** function complies a comprehensive list of all wallets registered on the MACoin blockchain, including each wallet’s address, balance, and number of transactions. It provides a detailed overview that helps to assess the overall exposure and distribution of assets in the network.

c) Transaction Management

`def transfer_funds(sender, receiver, amount, data, testing=False)`

This function validates the sender's credentials using the **`authenticate_user`** function unless it is in *testing* mode. It checks the sender's balance, deducts the specified amount, credits it to the receiver's wallet, and creates a transaction record with the relevant details, which is

then appended to a new block on the blockchain to ensure immutability and traceability of the transaction.

`def get_wallet_transactions(wallet_address)`

The **`get_wallet_transactions`** function iterates through the blockchain's ledger, examining each block to identify transactions that involve the specified wallet address, either as sender or receiver. For each relevant transaction found, it aggregates the details, including the transaction amount, timestamp, and counterparties, into a list providing a historical account of the wallet's activity.

`def get_all_transactions()`

This **`get`** function iterates through all blocks in the blockchain, extracting and compiling details of each transaction into a structured list. It ensures a complete and accessible record of all transactions, facilitating audit and oversight activities across the network.

d) Smart Contract Management

`def create_SC(address, conditions=None, contract_type=None, testing=False)`

Initiating with a user authentication check using the **`authenticate_user`** function, unless it is in *testing* mode, the **`create_SC`** method generates a unique contract name. It then utilizes the **`create_contract`** function to set up the contract with the designated conditions and type, subsequently registering it in the blockchain's smart contract registry.

`def create_contract(address, contract_name, conditions, contract_type)`

This function assesses the provided conditions and contract type, ensuring they align with predefined allowable types such as “funding” or “transaction”. If the conditions are valid, it initializes a new smart contract instance, which includes setting the contract name and conditions, then returns this instance for further processing or acceptance.

`def conditions()`

The **`conditions`** function guides the user through an interactive process to define the type and specific conditions of a smart contract by presenting options such as funding or transaction-related conditions. Hence, it is called when the user aims to create a smart contract yet did not provide any conditions. Depending on user input, the function then compiles and returns these conditions along with the chosen contract type, ensuring the smart contract's parameters are tailored to the user's requirements.

`def accept_contract(address, contract_name, conditions=None, contract_type=None, testing=False)`

This function handles both the signing of a contract by the initial creator as well as by further members who wish to be added as users. Upon user confirmation to proceed, it verifies the existence of the specified smart contract or initializes a new one if it does not exist. It then adds the user's address to the contract's party list, records the acceptance in the blockchain, and updates the contract's details accordingly. If in *testing* mode, it automatically confirms acceptance without user input.

`def show_contracts(contract_name=None)`

When provided with a specific contract name, this function retrieves and displays detailed information about that smart contract, including its type, conditions, and participating parties. If no contract name is specified, the function lists details for all contracts stored within the blockchain, enhancing transparency and accessibility.

`def add_party_SC(address, contract_name, testing=False)`

After validating the user with the **`authenticate_user`** function, the function checks if the specified smart contract exists within the system using the **`contract_name`**. If the contract is found, it uses the **`accept_contract`** function to add the user's address as a new party to the contract, thereby expanding the contractual agreement to include additional stakeholders. If in *testing* mode, it skips user input and automatically proceeds with the addition.

`def show_parties_SC(contract_name)`

This function retrieves the list of parties involved in a specific smart contract identified by **`contract_name`**. It returns detailed information about each party's involvement, enhancing understanding of who is bound by the contract's terms.

`def execute_contract()`

The function iterates through all registered smart contracts, checking if the conditions for each are met based on the latest blockchain data. If the conditions are satisfied, it executes the contract's stipulated actions, such as transferring funds or updating records, thereby enforcing the agreed terms automatically.

`def delete_contract(contract_name, address)`

The function first uses the **`authenticate_user`** function to verify that the user associated with the provided address has the necessary permissions and confirms their intent. If conditions allow, such as consensus among all parties or administrative rights, it removes the specified smart contract from the blockchain, effectively ending its terms and associated transactions.

`def check_conditions(contract_name)`

The `check_conditions` function retrieves and displays the specific conditions under which a named smart contract operates. By providing insights into the contractual terms and types, it aids stakeholders in understanding and verifying the obligations and criteria set within the contract.

e) Blockchain Display Operations

`def print_wallets_overview()`

By iterating through registered wallets on the blockchain, the `print_wallets_overview` function collects and formats key details such as wallet address, balance, number of transactions, and creation date for each wallet. It then formats these details into a structured HTML table, which is displayed to provide a clear and comprehensive overview of all wallets within the system.

`def print_wallet_transactions(wallet_address)`

Initiating with a call to the `get_wallet_transactions` function, this function retrieves a detail a list of all transactions associated with the given wallet address, detailing sender, receiver, amount, and timestamp for each transaction. These details are then also structured in an HTML table.

`def print_all_transactions()`

This function leverages the `get_all_transactions` method to gather all transaction records from the blockchain before formatting the retrieved information into an HTML table.

`def print_chain()`

The `print_chain` function iterates through each block in the blockchain's chain sequence, extracting and printing essential details such as the block's timestamp, transaction data, previous hash, and current hash. By systematically displaying these attributes for each block, it thus provides a sequential view of the entire blockchain. A such, it serves a simplified version of the `display_chain_diagram` function.

`def display_chain_diagram()`

The function `display_chain_diagram` visually represents a blockchain as a directed graph. It uses the graphviz library to create nodes for each block in the chain, displaying key details such as the block's previous hash, data, and transactions. Blocks are connected by edges that illustrate their sequential relationship, each pointing back to the previous hash that they are based on. This graph provides a clear visualization of the blockchain's structure, enhancing transparency and auditability.

3.5 Class: UserInterface

The *UserInterface* class serves as the main interaction layer for users, allowing them to engage with the blockchain system and execute various functions related to wallet management, transactions, and smart contracts. As such, it contains nine functions in total, categorized as main user interfaces (UI) for the blockchain and smart contracts, readability functions called within the previous function category, and functions representing the UI testing mode.

a) Main UI Functions

`def menu()`

The **`menu`** function serves as the primary interaction point between the user and the blockchain system, displaying a list of available actions within the *UserInterface* class. Upon user selection, it invokes the corresponding function from the *Blockchain* class based on the input, ensuring that each choice triggers the appropriate method for wallet management, transactions, or smart contract interactions.

`def smart_contract_menu(address=None)`

This function acts as the gateway for users to engage with smart contracts, presenting a menu that offers options such as creating, signing, viewing, and managing smart contracts. Based on the user's choice, it coordinates the execution of specific smart contract-related functions within the Blockchain class, effectively facilitating various contract operations directly from the user interface.

b) UI Readability Functions

`def get_contract_name()`

The **`get_contract_name`** function, designed for readability purposes within the U*serInterface* class, prompts the user to input the name of a smart contract, verifying its existence within the blockchain's stored contracts. If the contract is not found, it allows up to three attempts before raising an error.

`def get_wallet_address(transfer=None)`

This function, which is also intended for readability purposes, prompts the user to enter a wallet address, verifying its existence within the blockchain's stored wallets. If the *transfer* parameter is *True*, it specifically asks for the receiver's address in a transfer, otherwise, it assumes the user's own wallet address, allowing up to three attempts before raising an error.

c) UI Testing Functions

`def initiate_testing()`

This function is automatically invoked when the *UserInterface* class is instantiated. It prompts the user with an option to go directly to the testing menu, enabling a streamlined setup for testing purposes and initiating the blockchain in testing mode.

`def testing_UI()`

The `testing_UI` function serves as the testing menu interface, providing a limited set of choices for creating wallets, smart contracts, or transactions. It ensures that the testing environment mimics the normal operations while bypassing certain user prompts for efficiency, operating in testing mode.

`def create_test_wallets(number=10)`

This function creates a specified number of test wallets (default is 10) by repeatedly calling the `create_wallet` function with the *testing=True* parameter. This bypasses user authentication and the need to create a password, facilitating automated wallet creation for testing purposes in testing mode.

`def def create_test_transactions(number=10)`

The `create_test_transactions` function generates up to ten transactions between random wallets. It ensures no wallet is selected more than once per transaction pair and bypasses authentication by setting the *testing=True* parameter.

`def create_test_contracts(number=5)`

This function creates ten smart contracts, split evenly between “funding” and “transaction" types. By using *testing=True*, it bypasses authentication and user input prompts, setting hard-coded values for simplicity and adding random addresses to existing smart contracts.

3.6 Other Functions for Financial Visualization

This category of functions provides an interactive way for users to visualize how different currencies, especially cryptocurrencies, compare to the Swiss Franc, against which the established cryptocurrency MACoin is pegged with a 1:1 ratio, over a specified timeframe.

`def fetch_currency_data(currency, is_crypto=False)`

This function retrieves the market data for a specified currency, accommodating both cryptocurrencies and traditional currencies linked against the Swiss Franc (CHF). For cryptocurrencies, it constructs a URL to interact with the CoinGecko API, setting parameters to fetch market chart data for the past year with CHF as the base. Conversely, for traditional currencies, it

utilizes Yahoo Finance to download historical data of the currency pair (e.g. USDCHF) covering the same timeframe. This dual approach ensures precise and tailored data retrieval suitable for various financial analysis needs.

`def parse_data(data, is_crypto=False)`

The **`parse_data`** function processes the raw data obtained from the **`fetch_currency_data`** function into a structured and usable format. For cryptocurrencies, it extracts prices and timestamps from the data, converts these timestamps into date objects, and calculates the inverse of the prices to express the token's price in Swiss Francs (CHF). In contrast, for traditional currencies, it directly accesses the indexed “Close” prices from the data, transforms these into date objects, and performs similar inversions to represent the currency's value in CHF. This standardized format supports further financial analysis and visualization across different currency types.

`def plot_data(dates, values, currency)`

This function plots the price data that has been formatted by **`parse_data.`** It creates a line graph showing how the price of the currency has changed over time relative to CHF. The graph includes labels for dates, currency value, and a title that reflects the specific currency being analyzed.

`def display_crypto_price()`

This is the main function that orchestrates the user interaction and the flow of data fetching, parsing, and plotting: Initially, it queries the user to choose between cryptocurrency and traditional currency prices. Based on the user's response, it then prompts for the specific currency to be analyzed. It calls **`fetch_currency_data`** to retrieve the data, employs the **`parse_data`** function to format this data, and concludes with the **`plot_data`** function to visually display the currency price trends over time, enabling a comprehensive view of currency performance relative to the Swiss Franc (CHF).

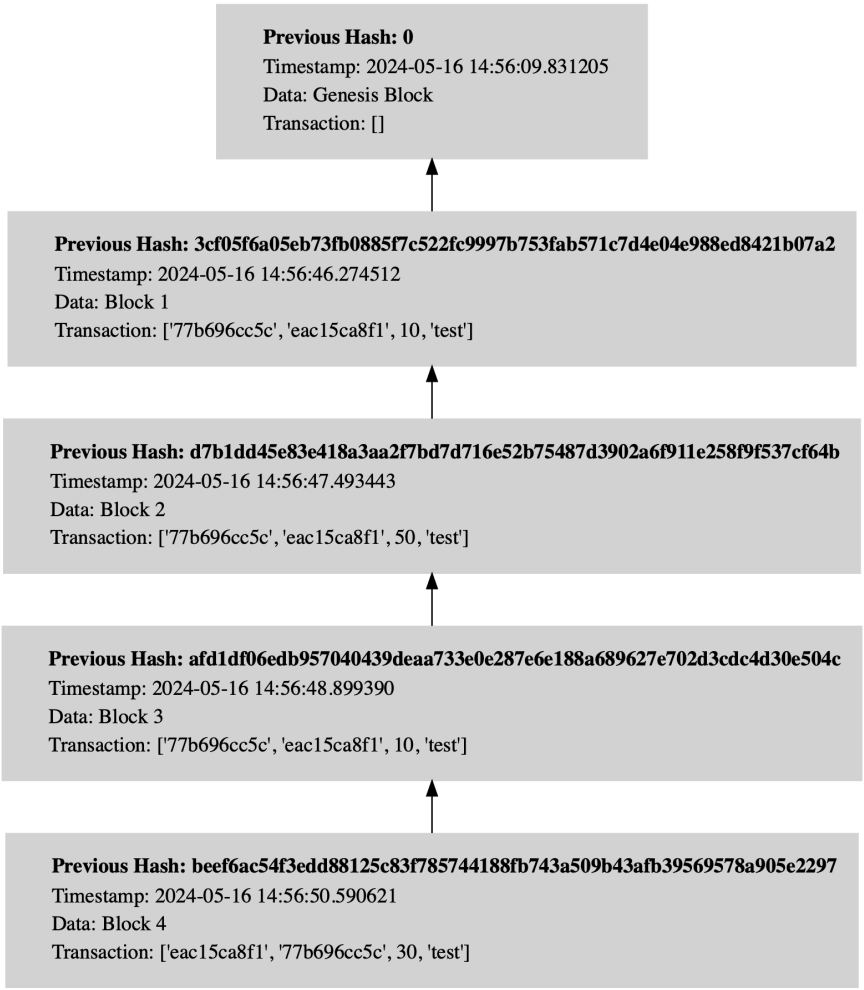
4 Limitations

Despite the broad range of features of the MACoin blockchain, it faces several limitations in terms of its execution and the given scope of the project.

Firstly, the execution of the function **`def display_chain_diagram()`** was successfully carried out in Google Colab, which offers a controlled environment pre-configured with numerous libraries and dependencies. However, its execution in other environments might not be straightforward as it requires specific plugins and libraries that may not be present or are differently

configured outside of Colab. To ensure compatibility and functionality across different platforms, it is crucial to verify the presence and correct configuration of all required dependencies in the target environment. This might involve installing additional software packages or adjusting system configurations to match the requirements established in the Google Colab environment.

Here is an example output:



Furthermore, constraints exist which result from the limited time and resources available for the given project. It is essential that the following aspects are addressed and implemented if this project was to be continued and scaled up.

For one, the current implementation runs on a single system and is therefore not truly decentralized. Nevertheless, it still provides the basic infrastructure to implement smart contracts, setting the foundation for future decentralization. Yet this very aspect, in addition to the fact that MACoin runs on Python with all “public” variables, limits the security of the blockchain. The lack of a consensus mechanism to check the validity of mined blocks further adds to the security concerns. In addition, this blockchain operates with unsecured passwords since they

are saved as the exact string provided instead of being hashed prior to storage. The ease of deleting the entire blockchain by simply reloading the function, resulting in the loss of all previous transactions, further compromises security. Furthermore, the algorithm for generating wallet addresses currently severely limits the number of users. To facilitate scalability, the length of the addresses could be increased. However, it should be highlighted that the MACoin project only represents a conceptual implementation. Thus, subsequent steps should include the implementation of robust security and growth features.

In regard to the smart contracts on the blockchain, several significant limitations are evident. Outstanding obligations are not verified to ensure that transactions can proceed if the wallet lacks sufficient resources to fulfill them, which could result in breaches of contract. Hence, in the further development of the project, the introduction of margin accounts could serve as a potential solution to this issue. The current implementation also restricts the ability to remove parties from smart contracts, necessitating either administrative rights or unanimous consent from all parties involved, with contract deletion only feasible as a sole member. Additionally, the smart contract conditions offer limited flexibility, with contracts categorized as “Other” not being automatically verified by the blockchain. These constraints underscore the necessity for more advanced mechanisms in future iterations to effectively manage outstanding obligations, party modifications, and condition verification.

Moreover, incorporating a more sophisticated graphical user interface (GUI) would significantly enhance user interaction and overall experience. A GUI could provide intuitive visual elements and controls, making the system more accessible and user-friendly. Additionally, the testing mode, while useful for demonstration and debugging, allows for the creation of wallets that function like normal ones, which would not exist in a real-world application. In this context, it therefore serves as a means to display features of the blockchain for testing purposes only.

In conclusion, while the MACoin project demonstrates a foundational prototype for blockchain-based digital wallets and smart contracts, several critical areas require enhancement for practical deployment. Future developments should focus on improving security, decentralization, and scalability, ensuring robust performance and reliability across diverse environments.