Sapienza Università di Roma

Department of Computer Science

FlashSwap Dapp

Blockchain and Distributed Technology

Professors:
Claudio Di Ciccio

Student:
Alessandro Di Patria

Academic Year 2022/2023

Contents

1 Introduction

The application presented in this paper, FlashLoan Swap proposes an innovative system for cryptocurrency arbitrage using a decentralized application called AAVE and a transaction type called FlashLoan. The application will allow the end user to borrow tokens from AAVE and employ them in arbitrage transactions between two Decentralized Exchanges to make a profit. All this is done in a single transaction making the operation truly secure. If something goes wrong the entire transaction is not written to the blockchain, as if it never happened.

The application presented was created purely for academic purposes. In fact, arbitrage takes place between two exchanges created by me and slightly simplified compared to traditional Decentralized exchanges. The end user will then be able to interact with them through the application.

2 Background

In This chapter will present all the various technologies used for this application.

2.1 Blockchain

Blockchain technology is a revolutionary way of storing and transferring data. It is a distributed ledger technology that allows for secure, transparent, and immutable transactions. Blockchain technology is based on a decentralized network of computers that are connected to each other and share information in real-time. This means that all transactions are recorded on a public ledger, which can be accessed by anyone with an internet connection. The most important feature of blockchain technology is its ability to provide trust and security in digital transactions. In fact the security of the blockchain is ensured through the use of public key cryptography to sign transactions and a decentralized consensus system to validate transactions and add new blocks to the chain. The consensus system used depends on the specific blockchain implementation, but the most common ones are Proof of Work (PoW) and Proof of Stake (PoS). The transparency provided by the blockchain allows the user to verify the validity of transactions occurring in the system .
The Blockchain offer a wide range of applications. The most frequent use cases are decentralized finance, voting systems authentication and certification of tangible objects.

2.2 Ethereum

Ethereum is a blockchain that offers proof-of-stake consensus technology. The token used by the platform is called ETH and is one of the most widely used in the blockchain world.
The ethereum platform allows pieces of code called Smart contracts to run directly on the blockchain.This allows developers to create decentralized applications running on top of it without any third party interference or downtime risk associated with traditional centralized systems.
Ethereum is the blockchain we will use for the development of our application.

2.3 Smart Contract

Smart contracts are self-executing contracts that are written in code and stored on a blockchain. They are designed to facilitate, verify, and enforce the performance of a contract between two or more parties without the need for a third-party intermediary. Smart contracts enable the exchange of money, property, shares, or anything of value in a transparent and conflict-free way while avoiding the services of a middleman.The main benefit of smart contracts is that they eliminate the need for trust between parties

involved in a transaction. Since all transactions are recorded on the blockchain, they cannot be altered or tampered with by any party involved in the transaction. This ensures that all parties involved in the transaction will receive what they agreed upon without any risk of fraud or manipulation.
Smart contracts also provide greater transparency than traditional contracts since all transactions are recorded on an immutable ledger. This makes it easier for parties to track their transactions and ensure that all terms of the agreement have been met.
The realized application uses smart contracts to handle various requests.

2.4 Token ERC-20

ERC-20 is a technical standard used for smart contracts on the Ethereum blockchain for implementing tokens . The ERC-20 standard defines a set of rules that all Ethereum tokens must follow in order to be accepted and used by the wider Ethereum community. These rules include how the tokens are transferred, how users can access data about a token and the total supply of tokens available. This helps ensure that all tokens are compatible with each other and can be exchanged easily on cryptocurrency exchanges.The ERC-20 standard has become so popular that it is now used as a base for most new tokens created on top of the Ethereum blockchain. This has helped create an ecosystem where developers can easily create new projects and users can easily exchange different types of tokens with each other without having to worry about compatibility issues. ERC-20 tokens will play a central role in the development of our application. In fact, the exchange of tokens that take place in the platform are precisely ERC20 tokens

2.5 Decentralized Exchange

A decentralized exchange (DEX) is a type of tokens exchange that operates without a central authority. This means that users can trade tokens directly with each other, without the need for an intermediary such as a bank or broker.
The logic of these Decentralized exchanges is governed entirely by smart contracts. Decentralized exchanges are different from traditional centralized exchanges in several ways. Firstly, they do not require users to provide personal information or go through a lengthy verification process in order to start trading. Secondly, they are not subject to the same regulations and restrictions as centralized exchanges, which makes them more attractive to traders who want more freedom and flexibility when trading cryptocurrencies. Finally, decentralized exchanges are generally more secure than centralized ones, as there is no single point of failure that could be targeted by hackers.

When a user wants to buy or sell a tokens , they place an order on the exchange's platform. This order is then broadcasted to all other users on the network who can choose to accept it if they wish. Once an order has been accepted by another user, the

two parties can then complete the transaction directly between themselves without any involvement from the exchange itself. This means that all transactions occur directly between users and no third party is involved in the process.
Exchanges can be implemented in two ways:

- **Order book model**: Buyers and sellers file orders. And the centralized system matches the buy orders to the sell orders. This is how the traditional stock exchange works.
- **Automated market makers (AMM)** : There is no centralized matchmaker. There are people who provide both tokens (Dogecoin and Shiba). They are called liquidity providers. These liquidity providers create a pool of Dogecoin and Shiba tokens. Now traders can come and deposit Dogecoin and get Shiba in return. This is done automatically, without a centralized entity. Traders pay a small percentage fee for the trade which goes to liquidity providers for their services

.

In addition to this basic functionality, some decentralized exchanges also offer additional features such as pooling which allow users to combine their funds together in order to increase their buying power and gain access to larger trades or better prices on certain assets. Pooling allows multiple users to pool their funds together in order to purchase larger amounts of cryptocurrency at once while staking allows users to lock up their funds for a set period of time in return for rewards such as interest payments or discounts on trading fees.
Uniswap is one of the most famous Decentralized exchanges on Ethereum Network but there are several and they operate on different blockchains.

2.6 Uniswap

Uniswap is a decentralized exchange protocol built on the Ethereum blockchain. It enables users to swap tokens without the need for a centralized exchange or counterparty. Uniswap works by allowing users to deposit tokens into a liquidity pool, which is then used to facilitate trades between buyers and sellers. The protocol uses an automated market maker (AMM) algorithm to determine the prices of tokens based on their supply and demand.

When a user deposits tokens into the liquidity pool, they receive liquidity provider (LP) tokens in return. These LP tokens represent their share of the pool and can be used to earn fees from trades that occur in the pool. The fees are split between the buyer and seller, with Uniswap taking a small portion as well.

2.6.1 The AMM algorithm

The AMM algorithm used by Uniswap is designed to ensure that prices remain stable over time, even when there are large fluctuations in supply and demand. This helps protect users from sudden price swings that can occur on other exchanges due to market manipulation or other factors. Prices are set according to a mathematical formula and not by intermediaries. In the case of Uniswap, it relies on the XYK model, which uses the following equation:

$$x * y = k$$

Where x is the amount of one token (let's call it Token X), y is the amount of the other token (let's called it Token Y), and k is the product or constant.

Let's go through a practical example. A pool was created with 1000 units of Token X and 1000 units of Token Y. That means that, for this example, k equals 1,000,000, and the initial price ratio between tokens is 1:1. Remember that k will remain constant. Consequently, if a user wants to deposit 10 Token X, the new amount of Token Y the pool needs to have can be calculated as:

$$new_y = k/new_x => new_y = 1000000/1010 = 990.1$$

This means that there is a surplus of 9.9 Token Y to keep the pool balanced (k constant). This is sent to the user who deposited the 10 Token X, which got a price of 0.99 Token Y per Token X

2.6.2 Uniswap Smart Contracts

The Uniswap Factory contract is arguably the core contract of the protocol. Its primary purpose is to create pools (called pairs in the contract). To do so, it uses the create2 opcode, which is capable of deploying contracts with deterministic addresses. This is useful to calculate the address of the pool of a pair of tokens (even off-chain)

The router contract is the smart contract used to interact with a pool. Routers are stateless, meaning they don't hold token balances. Therefore, they can be replaced safely and in a trustless way for a more efficient router in the future. The most important functions of this contract are :

- **AddLiquidity** : functions that handle adding liquidity to an already existing pool. As input, you provide the address of both tokens, the amount you want to provide
- **RemoveLiquidity** : similar to the previous function, but handles the extraction of liquidity from an already existing pool. As inputs, you provide the address of both pool tokens, the number of LP tokens you want to burn, the minimum amount of tokens you wish to receive,
- **SwapExactTokensForTokens** : this function calculates and withdraws a certain number of tokens for a given input amount. For example, let's say a pool holds tokensA and tokensB, such as tokensA * tokensB = k. The amount of tokensB a user would withdraw from the pool (tokensBout) for a given input of tokensA (tokensAin) is given by the following equation:

$$tokenBout = tokensB - (k/tokensAin)$$

Uniswap Pair Contract a smart contract that implements the functionality for swapping, minting, burning of tokens. This contract is created for every exchange pair like Dogecoin-Shiba, DAI-USDC, ecc... **Scrivi le funzioni dello smart contract**

2.6.3 Web Interface

Uniswap also provides an easy-to-use interface for users to interact with the protocol. Users can easily view token prices, deposit funds into pools, and initiate trades without needing any technical knowledge or experience with cryptocurrency trading. This makes it an ideal platform for both experienced traders and those new to cryptocurrency trading alike.

2.7 Arbitrage

An arbitrage operation in DeFi is a trading strategy that takes advantage of price discrepancies between different decentralized exchanges (DEXs) and other DeFi protocols. By buying an asset on one platform and selling it on another, traders can take advantage of the price difference to make a profit. This type of trading is often done with automated bots that monitor the markets for potential opportunities. This type of operation is the basis of the architecture of our operation. In fact, a contract has been made that will exploit the difference in prices of two different exchanges to make a profit

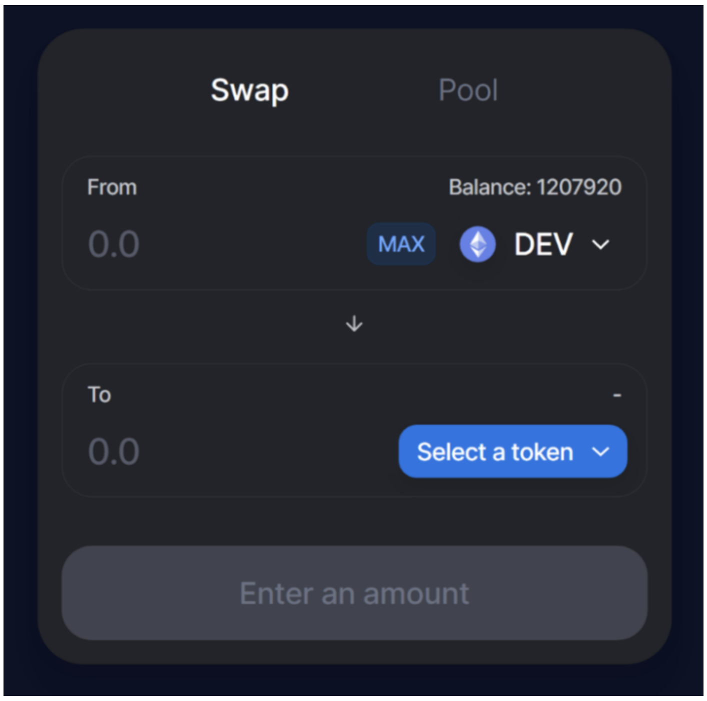


Figure 1: Uniswap web Interface

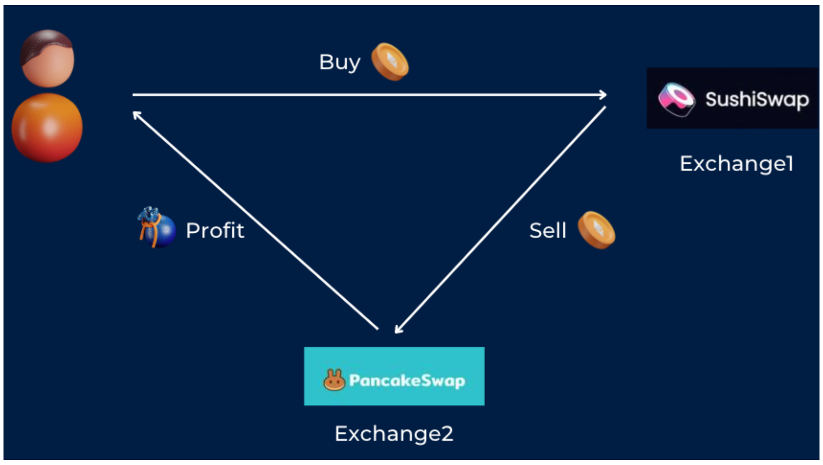


Figure 2: The user Buy a Token in a first exchange in wich the price is low and sell in another Dex in which the price is highter

2.8 Lending and Borrowing

Traditionally, to get a loan, you'd need to go to a bank or other financial institution with lots of liquid cash. The bank will ask for collateral in the case of a car loan, that would be the car title itself in exchange for the loan. You then pay the principal to the bank every month, plus interest. Lending and borrowing in DeFi refers to the ability to lend and borrow digital assets without the need for a centralized intermediary. This is made possible through the use of smart contracts, which are self-executing agreements that are stored on the blockchain. With DeFi, users can lend and borrow digital assets such as cryptocurrencies, stablecoins, and other tokens with no middleman involved. This allows for faster transactions, lower fees, and more secure transactions than traditional lending and borrowing methods.
In decentralized finance, there are different types of platforms that allow people to lend and receive tokens. One of them is AAVE.

2.9 AAVE Protocol

Aave is a decentralized finance (DeFi) protocol that lets people lend and borrow tokens and real-world assets (RWAs) without having to go through a centralized intermediary. When they lend, they earn interest; when they borrow, they pay interest. Aave was originally built atop the Ethereum network, with all the tokens on the network also using the Ethereum blockchain to process transactions; they are known as ERC20 tokens. The protocol itself uses a decentralized autonomous organization, or DAO. That means it's operated and governed by the people who hold and vote with AAVE tokens. Aave currently has pools for 30 Ethereum-based assets, including the stablecoins Tether, DAI, USD Coin.

Use cases

Although it often makes more sense to buy or sell cryptocurrency, borrowing it can be practical in some circumstances. One of the most obvious is for arbitrage. If you see a token trading at different rates on different exchanges, you can make money by buying it at one place and selling it at another. However, since differences tend to be minor after taking into account transaction fees and spreads, you'd have to have a lot of the cryptocurrency to turn a decent profit. Aave pioneered the use of flash loans, in which people borrow cryptocurrency without collateral, use it to buy an asset, sell that asset, and then return the original amount in the same transaction while pocketing their profit.

2.10 FlashLoan

A flash loan is a type of loan that is executed and settled on the blockchain in a single transaction. It is an innovative financial instrument that allows users to borrow funds without collateral or credit checks. We can borrow potentially millions of tokens and use in a secure way to obtain a kind of advantage. Usually flash loans are quite safe infact is something is going wrong in some operations executed the entire transaction is reverted.

Use cases
They are useful for arbitrage trading, liquidity provision and hedging purposes which makes them attractive for traders looking for quick access to capital without having to worry about the risks associated with traditional lending platforms.

2.11 AAVE FlashLoan

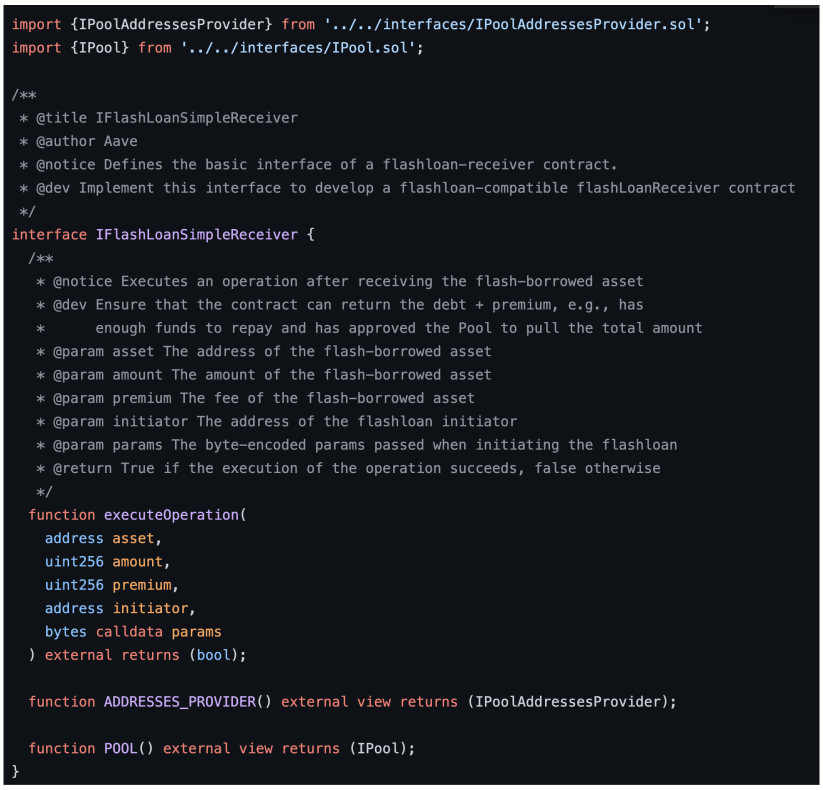
Aave V3 offers two options for flash loans executed by smart contracts:

- **FlashLoan**: Allows borrower to access liquidity of multiple reserves in single flashLoan transaction. The borrower also has an option to open stable or variabled rate debt position backed by supplied collateral or credit delegation in this case.
- **FlashLoan Simple**: Allows borrower to access liquidity of single reserve for the transaction. In this case flash loan fee is not waived nor can borrower open any debt position at the end of the transaction. This method is gas efficient for those trying take advantage of simple flash loan with single reserve asset.

Execution of a FlashLoan

In order to execute a AAVE FlashLoan with your smart contract code we have to follow these rules.

- Your flashloan contract calls the Pool contract, requesting a flashloan of a certain amount(s) of reserve(s) calling *flashLoanSimple() or flashLoan()* functions from contract pool.
- After some sanity checks, the Pool transfers the requested amounts of the reserves to your contract, then calls *executeOperation()* on your smart contract. (Your contract that receives the flash loaned amounts must conform to the **IFlashLoanSimpleReceive** or **IFlashLoanReceiver.sol** interface by implementing the relevant executeOperation() function.)



```
import {IPoolAddressesProvider} from '../../interfaces/IPoolAddressesProvider.sol';
import {IPool} from '../../interfaces/IPool.sol';

/**
 * @title IFlashLoanSimpleReceiver
 * @author Aave
 * @notice Defines the basic interface of a flashloan-receiver contract.
 * @dev Implement this interface to develop a flashloan-compatible flashLoanReceiver contract
 */
interface IFlashLoanSimpleReceiver {
  /**
   * @notice Executes an operation after receiving the flash-borrowed asset
   * @dev Ensure that the contract can return the debt + premium, e.g., has
   *      enough funds to repay and has approved the Pool to pull the total amount
   * @param asset The address of the flash-borrowed asset
   * @param amount The amount of the flash-borrowed asset
   * @param premium The fee of the flash-borrowed asset
   * @param initiator The address of the flashloan initiator
   * @param params The byte-encoded params passed when initiating the flashloan
   * @return True if the execution of the operation succeeds, false otherwise
   */
  function executeOperation(
    address asset,
    uint256 amount,
    uint256 premium,
    address initiator,
    bytes calldata params
  ) external returns (bool);

  function ADDRESSES_PROVIDER() external view returns (IPoolAddressesProvider);

  function POOL() external view returns (IPool);
}
```

Figure 3: The figure above illustrates all interfaces, addresses and functions that we must use to execute a AAVE flashloan

Once you have performed your logic with the flash loaned assets (in your *executeOperation()* function), you will need to pay back the flash loaned amounts if you used *flashLoanSimple()* or interestRateMode=0 in flashLoan()for any of the assets in modes parameter.

- **Paying back a flash loaned asset**: Ensure your contract has the relevant amount + premium to payback the borrowed asset. You can calculate this by taking the sum of the relevant entry in the amounts and premiums array passed into the *executeOperation()* function. You do not need to transfer the owed amount back to the Pool. The funds will be automatically pulled at the conclusion of your operation.

- **Incurring a debt** : If you initially used a mode=1 or mode=2 for any of the assets in the modes parameter, then the address passed in for *onBehalfOf* will incur the debt if the onBehalfOf address has previously approved the msg.sender to incur debts on their behalf. This means that you can have some assets that are paid back immediately, while other assets incur a debt.

3 Context

3.1 Application's logic

FlashSwap Dapp is a protocol that allows the end user to borrow tokens from the AAVE pool for use in arbitrage operations between two Decentralized Exchanges called Sushiswap and Uniswap. All this is done through a single transaction atomically. If something goes wrong the entire transaction is aborted as if it never happened.
Our Dapp allows the user to choose different arbitrage opportunities between various pairs of tokens. The system reads the prices of the same pairs from the two exchanges and tells the user which arbitrage opportunity might be more profitable by calculating the price differences. At that point the user can enter the amount he or she wishes to borrow and execute the transaction (a flash loan). Automatically our application returns the borrowed tokens to the Aave pool and they are now withdrawable by the user.
Our application being a demo, also allows the user to interact with the Decentralized Exchanges with which they are connected. In fact, the user can interact with them by creating new pools, exchanging tokens and adding liquidity to token pairs.

3.2 Use Cases

3.2.1 FlashLoan Logic

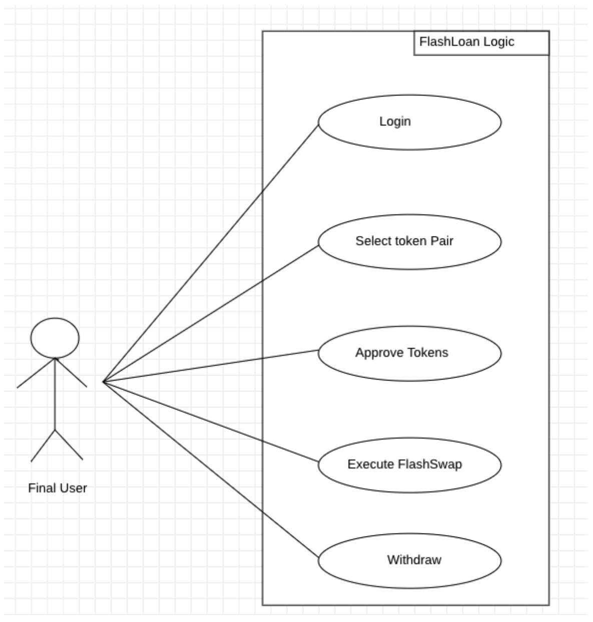


Figure 4: The figure above illustrates all interfaces, addresses and functions that we must use to execute a AAVE flashloan

- **Login**: The system must allow the user to login into the system with a web3 provider.

- **Select Token Pair**: The system must allow the user to select the tokens pairs that want to trade.
- **Approve tokens**: The system must allow the user to approve the amount of token that the smart contract have to use for the user.
- **Execute FlashLoan**: The system must execute for the user a flashloan with the value and token pairs selected.
- **Withdraw**: The system must allow the user to withdraw his funds after the operations executed.

3.2.2 Decentralized Exchange Logic

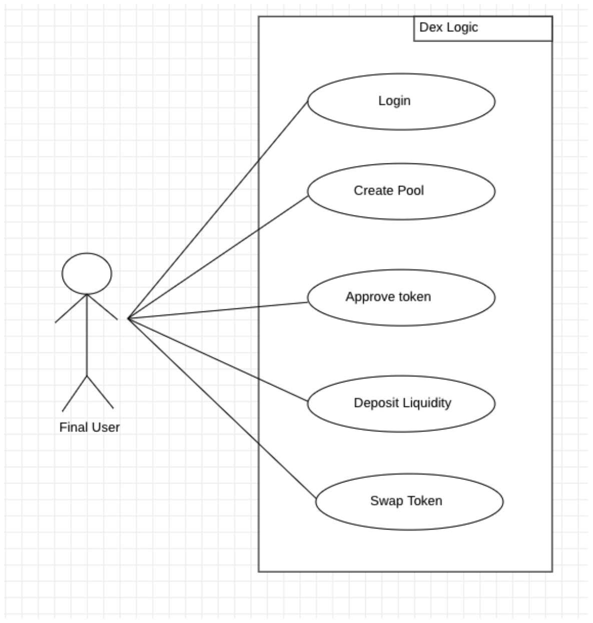


Figure 5: The figure above illustrates all interfaces, addresses and functions that we must use to execute a AAVE flashloan

- **Login**: The system must allow the user to login into the system.
- **Create Token Pair**: The system must allow the user to create a token pair pool.
- **Approve tokens**: The system must allow the user to approve the amount of token that the smart contract have to use for the user.
- **Deposit Liquidity**: The system must allow the user to Deposit liquidity into a pool .
- **Swap tokens** : The system allow the user to swap a tokanA from a tokenB taking tokens from the corresponding pool.

3.2.3 Scenario1 : Execute a flashLoan

The following actions are useful to understand how correctly use the system.

- Firstly the user login to the system.
- After that the user can choose a profitable pair of tokens.
- Now the user approve the amount of token that he want use for the Loan.
- Finally the user execute the FlashLoan and withdraw fund earned with this operation.

3.2.4 Scenario2 : Decentralized exchange

The following actions are useful to understand how correctly use the system.

- Firstly the user login to the system.
- After that the user can create a new pool pairs of tokens
- Now the user can deposit liquidity in the pool created.
- The user can now swap the first token for the second token in the pool. If a pool already exist the user can immediately swap a token for another without the other steps.

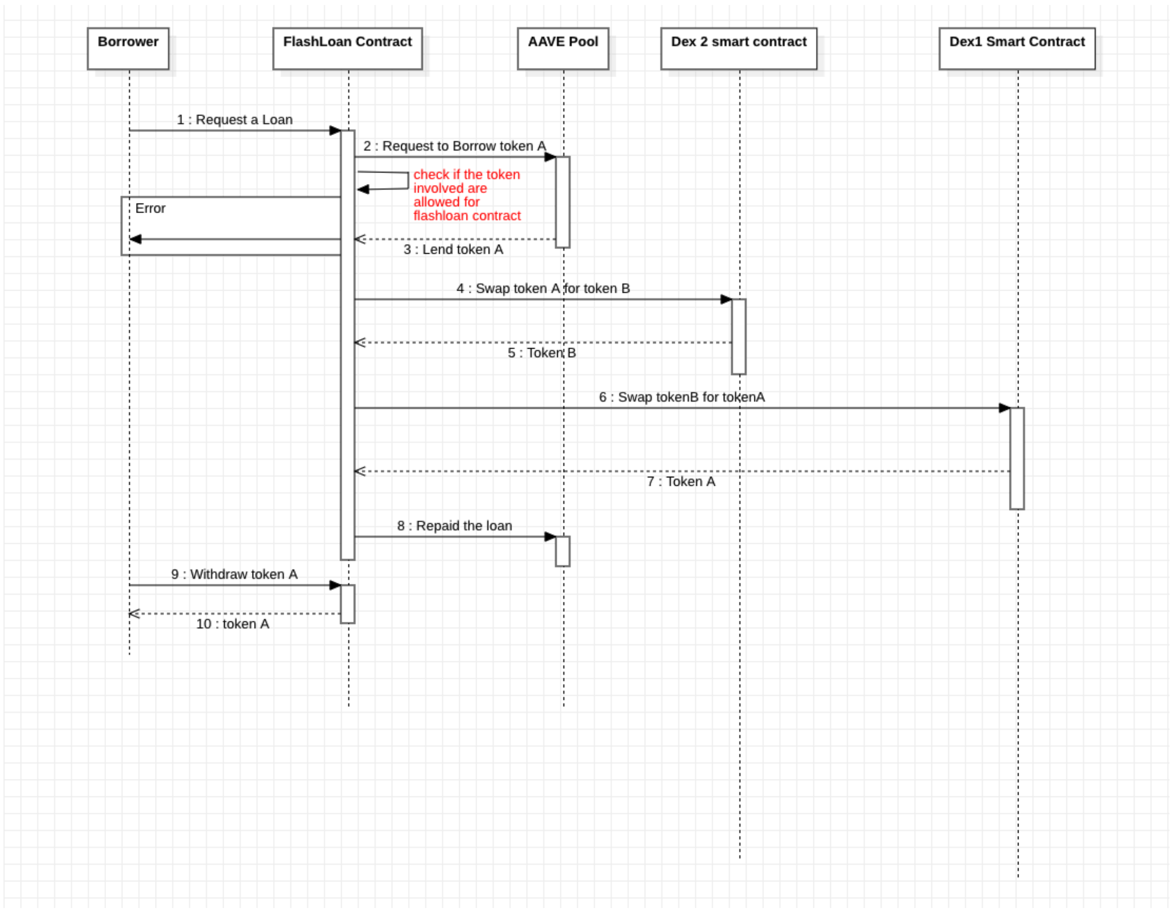


Figure 6: The sequence diagram above show how a flahloan transaction work and who are the actors involved the amount of token involved in the transactions (tokenA and token b must be allowed by the ERC-20 token A and token B smart contract

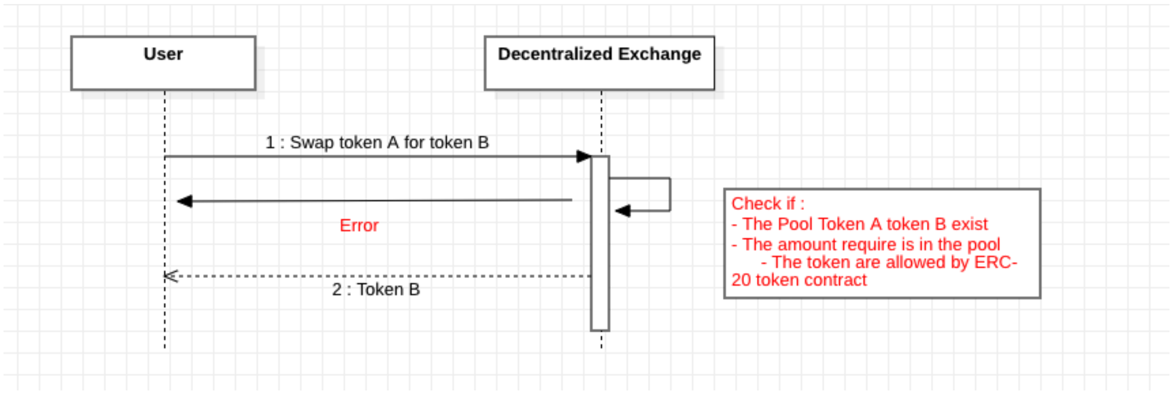

User
Decentralized Exchange
1 : Swap token A for token B
Error
2 : Token B
Check if :
- The Pool Token A token B exist
- The amount require is in the pool
- The token are allowed by ERC-20 token contract

Figure 7: The sequence diagram above show the swap function provided by Dex contract.This function allow the user to swap a token for another if the pool for this two token already exist

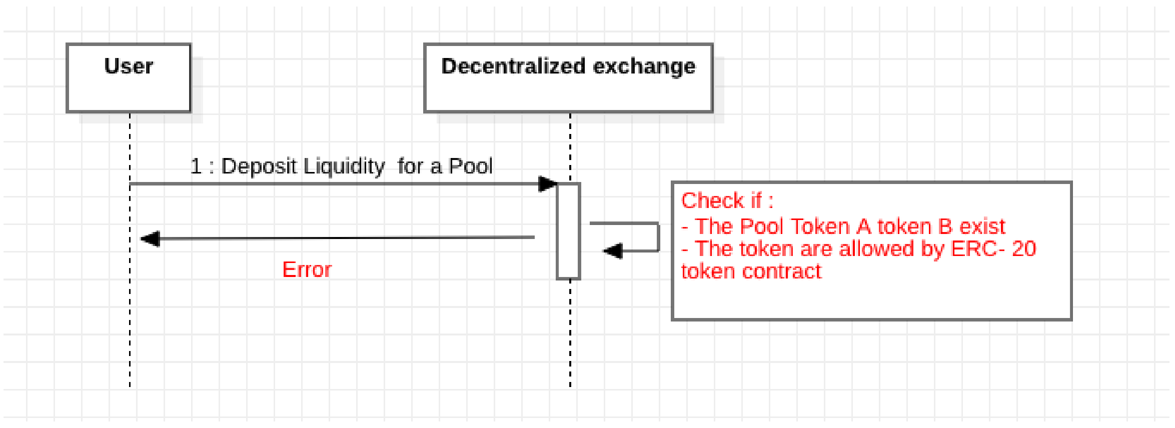

User
Decentralized exchange
1 : Deposit Liquidity for a Pool
Error
Check if :
- The Pool Token A token B exist
- The token are allowed by ERC- 20 token contract

Figure 8: The sequence diagram above show a Deposit transaction in which the user deposit two tokens in a Pool

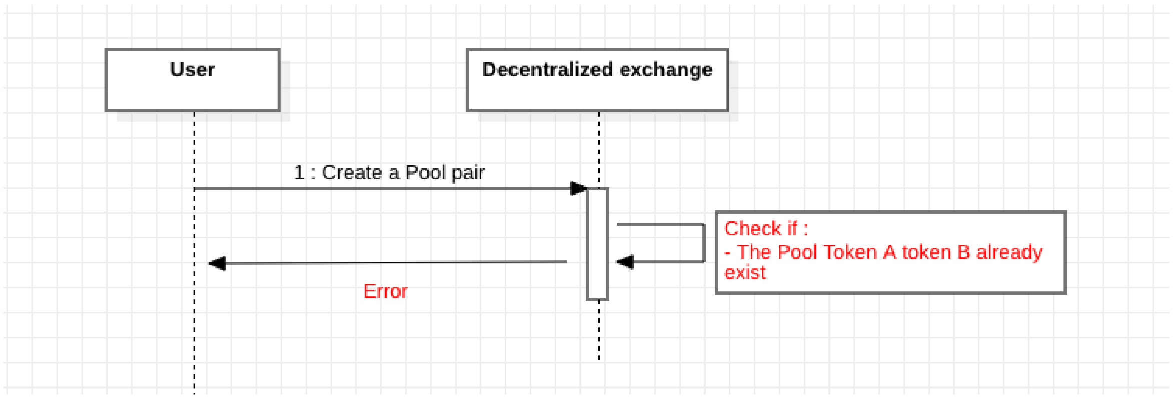


Figure 9: The sequence diagram above show a Create Pool transaction in which we create a pool deciding it's price

4 SoftwareArchitecture

In this chapter we are going to analyze all the tools that were used to realize the logic and operation of our application. In particular, we made use of :

- **AAVE protocols** to execute FlashLoan and borrow the tokens.
- **ERC-20 standard smart contract** for tokens managed by Decentralized exchanges and to approve transactions.
- **Hardat** for development
- **Infura and alchemy Providers** : allow us to quickly connect to the blockchain network .
- **Remix idle** : Smart contract writing software used to write contracts .
- **Etherscan** : Software that allows us to read every transaction in the blockchain. It was used to approve contracts by making them readable on the platform and to verify that the transactions are working properly.
- **Goerli Eth** : Ethereum test net on which the entire FlashSwap protocol is based.

4.1 FlashLoan Contract

The **FlashLoan.sol** smart contract contains all the functionality to manage a Flashloan through AAVE.
The smart contract must conform to the two interfaces **IFlashLoanSimpleReciever.sol** and **IFlashLoanReciever.sol** by implementing the function *executeOperation()*. The interface of our decentralized exchange smart contract is imported into the smart contract. In this way knowing the addresses of the two exchange contracts we can call their functions directly. We also import the **ERC-20 interface** to access tokens related functions such as *balance() approve() and allowance()*. Finally we add the **IPoolAddressProvider** interface to access the AAVE contract pool functions.
There are two main functions : the first is **executeOperation()**in which we are going to enter the operations we want to do with the tokens borrowed from the AAVE Pool. In our case we do an arbitrage operation between the two Dex contracts. *RequestFlashLoan()* which is used to get us in touch with the interface **IPoolAddressProvider** to request the loan. This function internally calls the *executeOperation()* function of our contract to execute our logic.

4.1.1 Limitations

We can only borrow from the pool the USDC token so we could do arbitrage only with pools that year as USDC participation token.

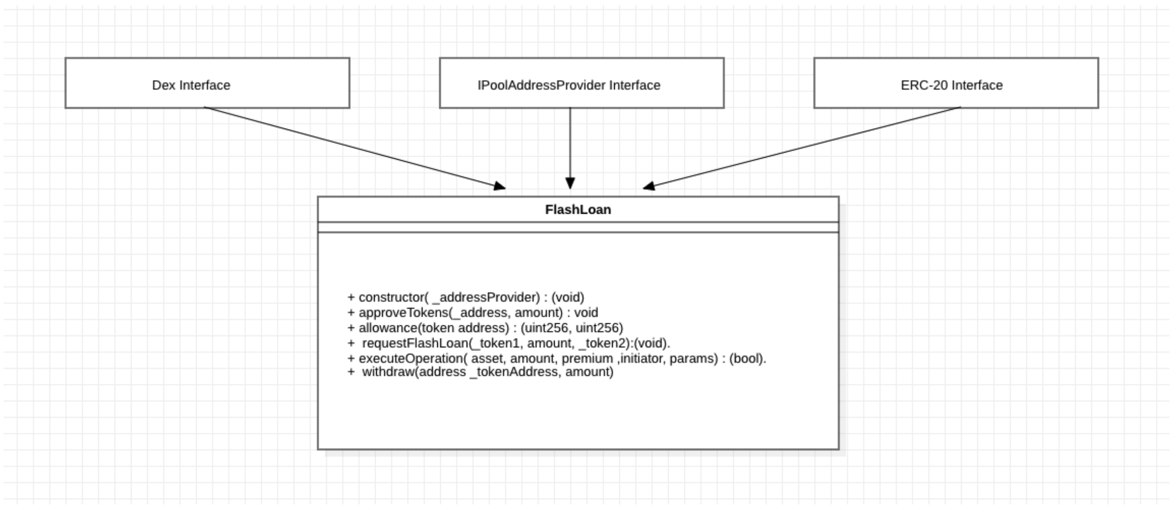

Dex Interface
IPoolAddressProvider Interface
ERC-20 Interface
FlashLoan
+ constructor(_addressProvider) : (void)
+ approveTokens(_address, amount) : void
+ allowance(token address) : (uint256, uint256)
+ requestFlashLoan(_token1, amount, _token2):(void).
+ executeOperation(asset, amount, premium ,initiator, params) : (bool).
+ withdraw(address _tokenAddress, amount)

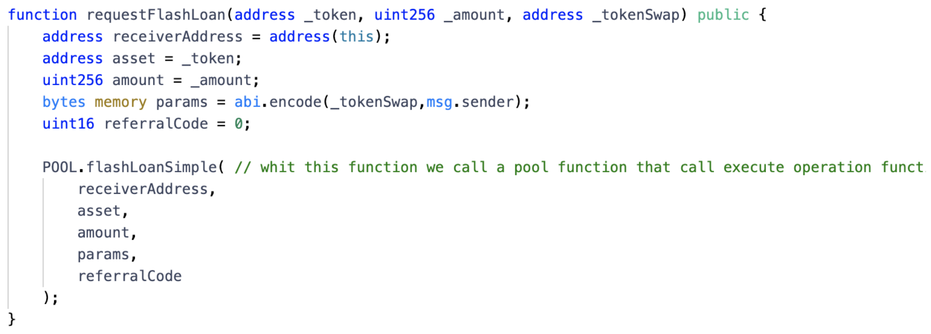
Figure 10: FlashLoan class diagram

4.1.2 Contract Functions Event

In this section we will look at all the functions used to realize the smart contract

requestFlashLoan()

The function allows you to call the AAVE Pool's *flashLoanSimple()* function to request the flashloan. This function receives as input the two tokens for which arbitration is desired. In *flashLoanSimple()* are given as input the address of the two tokens one in direct mode the other encoded in ABI, the amount of tokens to be borrowed and the referral code which is an internal parameter and indicates the type of flashloan to be performed.



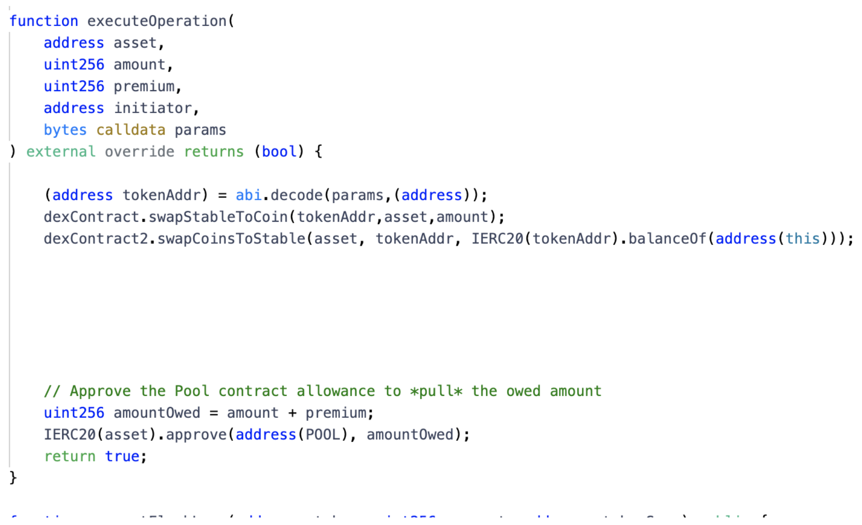
```
function requestFlashLoan(address _token, uint256 _amount, address _tokenSwap) public {
    address receiverAddress = address(this);
    address asset = _token;
    uint256 amount = _amount;
    bytes memory params = abi.encode(_tokenSwap,msg.sender);
    uint16 referralCode = 0;

    POOL.flashLoanSimple( // whit this function we call a pool function that call execute operation funct
        receiverAddress,
        asset,
        amount,
        params,
        referralCode
    );
}
```

executeOperation()

This function will be called from the IPoolAddressProvider interface when we call the *requestFlashLoan()* function. The function is necessary for the flashloan to be performed with AAVE. It receives as input some parameters such as :

- *asset* that corresponds to the address of the token we want to borrow
- *value* of the token
- *params* : special parameter in which we can enter any value needed for our operations. It is necessary that the values entered are encoded in Abi code. In our case we are going to enter the address of the second token to perform the arbitrage operations.

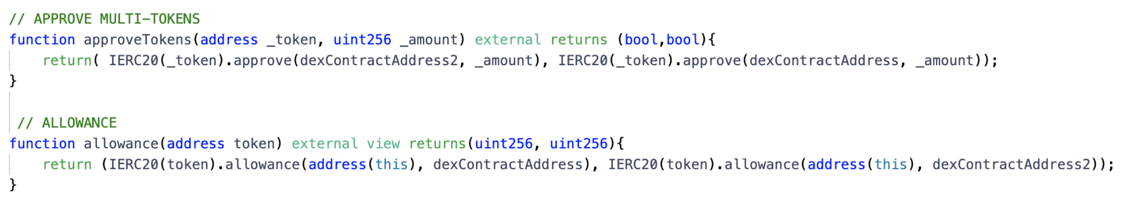
At this point we make use of the Dex interface by calling the functions *swapStableToCoin()* on the first dex and *swapCoinToStable()* on the second. Finally we return the borrowed value to the pool.



```
function executeOperation(
    address asset,
    uint256 amount,
    uint256 premium,
    address initiator,
    bytes calldata params
) external override returns (bool) {

    (address tokenAddr) = abi.decode(params,(address));
    dexContract.swapStableToCoin(tokenAddr,asset,amount);
    dexContract2.swapCoinsToStable(asset, tokenAddr, IERC20(tokenAddr).balanceOf(address(this)));






    // Approve the Pool contract allowance to *pull* the owed amount
    uint256 amountOwed = amount + premium;
    IERC20(asset).approve(address(POOL), amountOwed);
    return true;
}
```

approve() and allowance ()

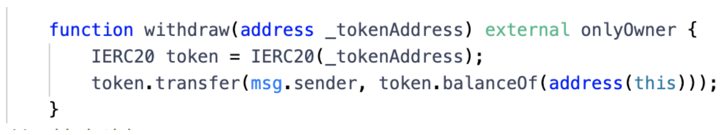
It is necessary for the smart contract to be authorized to manage a certain type of ERC-20 token against a third party and thus to perform arbitrage. Therefore through the ERC-20 interface we need to call the *approve()* function which will allow the user to manage funds of which he is not the owner. Allowance, on the other hand, allows for verification of the amount that the smart contract is authorized to spend.



```
// APPROVE MULTI-TOKENS
function approveTokens(address _token, uint256 _amount) external returns (bool,bool){
    return( IERC20(_token).approve(dexContractAddress2, _amount), IERC20(_token).approve(dexContractAddress, _amount));
}

// ALLOWANCE
function allowance(address token) external view returns(uint256, uint256){
    return (IERC20(token).allowance(address(this), dexContractAddress), IERC20(token).allowance(address(this), dexContractAddress2));
}
```

withdraw()

The function allows the user to take ERC-20 tokens generated by arbitrage operations.



```
function withdraw(address _tokenAddress) external onlyOwner {
    IERC20 token = IERC20(_tokenAddress);
    token.transfer(msg.sender, token.balanceOf(address(this)));
}
```

4.2 Dex Contract

The two equal smart contracts to realize decentralized exchanges contain several functions of the smart contracts that make up the UniswapV3.sol exchange.

These functions have been revised and simplified a great deal as it was intended to give this contract a slightly more marginal role than the flashloan contract. This smart contract contains all the functions to create and manage a pool just like Sushi swap. It also allows you to swap tokens from the same pool.
To the contract we connect the ERC-20 interface critical to the operation of a decentralized exchange.

4.2.1 Limitations

Being a simplified reinterpretation of SushiSwap, we decided not to include the AMM algorithm within the smart contract and not to use oracles for pricing. Simply the user who creates the pool sets the price. That corresponds to the price of the first on the second coin. For example if we create a pool with USDC and AAVE and assign 130 [1] to it as the value it means that 1 USDC is worth 1.3 AAVE. This price will remain unchanged until the user destroys the pool. In addition, the pools created must necessarily contain a stable coin of your choice between USDT and USDC this is because the flashLoan can only borrow USDC from the pool.

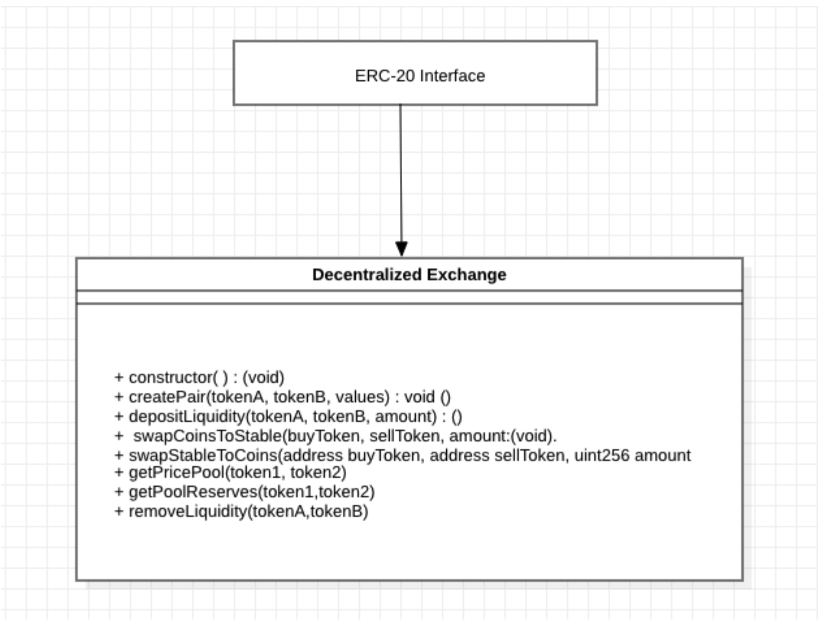

ERC-20 Interface
Decentralized Exchange
+ constructor( ) : (void)
+ createPair(tokenA, tokenB, values) : void ()
+ depositLiquidity(tokenA, tokenB, amount) : ()
+ swapCoinsToStable(buyToken, sellToken, amount:(void).
+ swapStableToCoins(address buyToken, address sellToken, uint256 amount
+ getPricePool(token1, token2)
+ getPoolReserves(token1,token2)
+ removeLiquidity(tokenA,tokenB)

Figure 11: Decentralized exchange class diagram

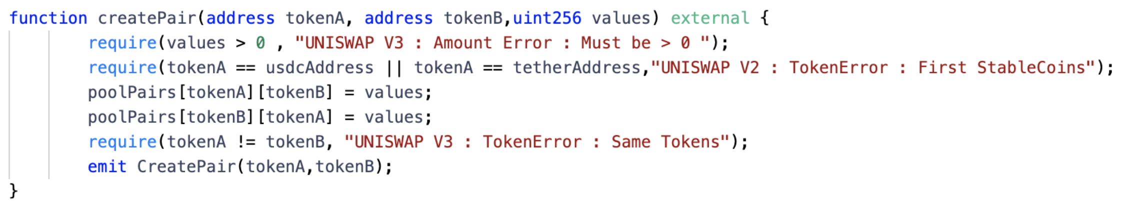
[1]we use 130 because in solidity you cannot represent decimals

4.2.2 Contract Functions

In this section we will look at all the functions used to realize the smart contract

createPair()

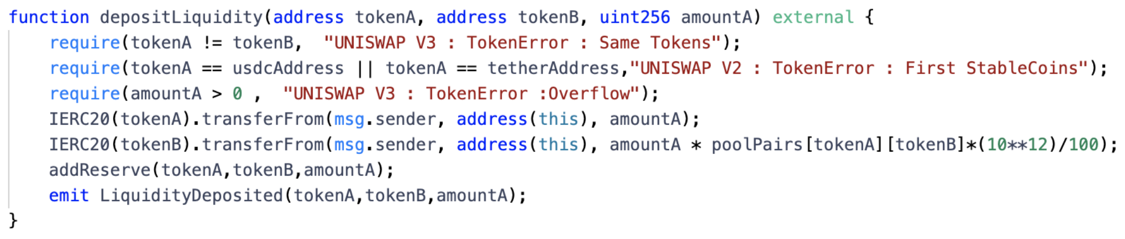
Allows the user to create a pool. The user gives as input the addresses of the two ERC-20 tokens and how much one is worth relative to the other We check that the pool does not already exist and that the value is greater than zero. Then we call add the token pair and its value in both directions to the poolPairs memory and issue the event,



```
function createPair(address tokenA, address tokenB,uint256 values) external {
        require(values > 0 , "UNISWAP V3 : Amount Error : Must be > 0 ");
        require(tokenA == usdcAddress || tokenA == tetherAddress,"UNISWAP V2 : TokenError : First StableCoins");
        poolPairs[tokenA][tokenB] = values;
        poolPairs[tokenB][tokenA] = values;
        require(tokenA != tokenB, "UNISWAP V3 : TokenError : Same Tokens");
        emit CreatePair(tokenA,tokenB);
}
```

depositLiquidity()

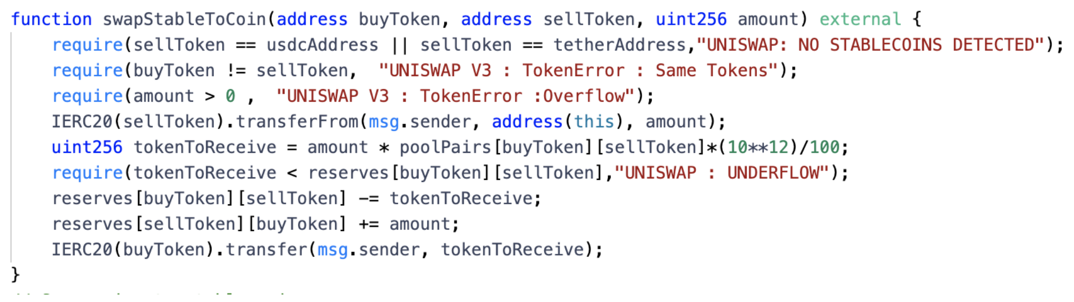
The function allows the user to deposit liquidity for the pool corresponding to the input tokens. The function checks that the tokens given as input are not equal and that the deposit value is greater than zero. It uses the *transferFrom* function of the ERC-20 interface to transfer funds from the user to the smart contract. The function calls the internal *addReserve()* function to add the amount of reserve added to the memory. Finally, it issues the *LiquidityDeposited()* event.



```
function depositLiquidity(address tokenA, address tokenB, uint256 amountA) external {
    require(tokenA != tokenB,  "UNISWAP V3 : TokenError : Same Tokens");
    require(tokenA == usdcAddress || tokenA == tetherAddress,"UNISWAP V2 : TokenError : First StableCoins");
    require(amountA > 0 ,  "UNISWAP V3 : TokenError :Overflow");
    IERC20(tokenA).transferFrom(msg.sender, address(this), amountA);
    IERC20(tokenB).transferFrom(msg.sender, address(this), amountA * poolPairs[tokenA][tokenB]*(10**12)/100);
    addReserve(tokenA,tokenB,amountA);
    emit LiquidityDeposited(tokenA,tokenB,amountA);
}
```

swapStableToCoin()

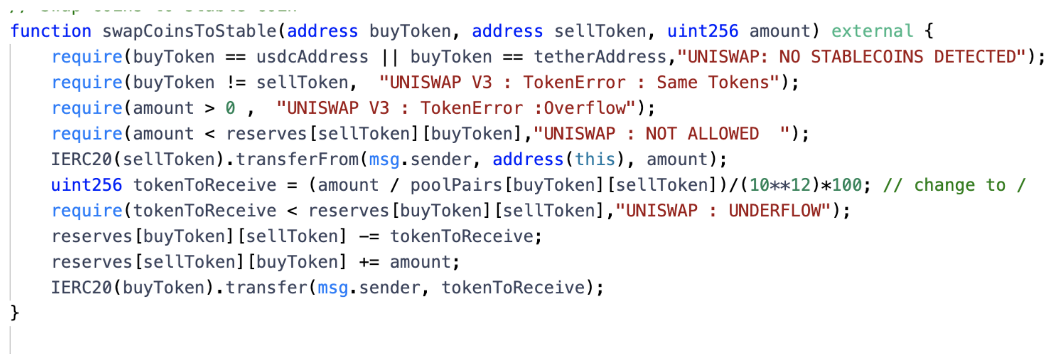
The function allows you to exchange USDC or USDT for another type of token. It checks that the necessary funds are present in the smart contract and that the exchange value is greater than zero. It then uses the ERC-20 interface function *transfer()* to transfer the requested token to the user and the function *transferFrom()* to transfer the tokens from the requestor to the pool.



```
function swapStableToCoin(address buyToken, address sellToken, uint256 amount) external {
    require(sellToken == usdcAddress || sellToken == tetherAddress,"UNISWAP: NO STABLECOINS DETECTED");
    require(buyToken != sellToken,  "UNISWAP V3 : TokenError : Same Tokens");
    require(amount > 0 ,  "UNISWAP V3 : TokenError :Overflow");
    IERC20(sellToken).transferFrom(msg.sender, address(this), amount);
    uint256 tokenToReceive = amount * poolPairs[buyToken][sellToken]*(10**12)/100;
    require(tokenToReceive < reserves[buyToken][sellToken],"UNISWAP : UNDERFLOW");
    reserves[buyToken][sellToken] -= tokenToReceive;
    reserves[sellToken][buyToken] += amount;
    IERC20(buyToken).transfer(msg.sender, tokenToReceive);
}
```

swapCoinsToStable()

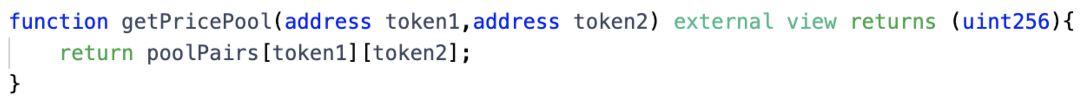
The function allows you to exchange a token for USDC or USDT. It checks that the necessary funds are present in the smart contract and that the exchange value is greater than zero. It then uses the ERC-20 interface function *transfer()* to transfer the requested token to the user and the function *transferFrom()* to transfer the tokens from the requestor to the pool.



```
function swapCoinsToStable(address buyToken, address sellToken, uint256 amount) external {
    require(buyToken == usdcAddress || buyToken == tetherAddress,"UNISWAP: NO STABLECOINS DETECTED");
    require(buyToken != sellToken,  "UNISWAP V3 : TokenError : Same Tokens");
    require(amount > 0 ,  "UNISWAP V3 : TokenError :Overflow");
    require(amount < reserves[sellToken][buyToken],"UNISWAP : NOT ALLOWED  ");
    IERC20(sellToken).transferFrom(msg.sender, address(this), amount);
    uint256 tokenToReceive = (amount / poolPairs[buyToken][sellToken])/(10**12)*100; // change to /
    require(tokenToReceive < reserves[buyToken][sellToken],"UNISWAP : UNDERFLOW");
    reserves[buyToken][sellToken] -= tokenToReceive;
    reserves[sellToken][buyToken] += amount;
    IERC20(buyToken).transfer(msg.sender, tokenToReceive);
}
```

getPricePool()

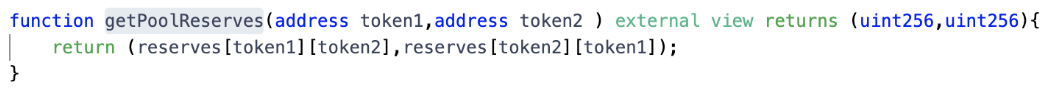
Given two tokens as input, the function returns the price of the pool.



```
function getPricePool(address token1,address token2) external view returns (uint256){
    return poolPairs[token1][token2];
}
```

getPoolReserves()

Given two tokens as input, the function returns the reserve of the pool.



```
function getPoolReserves(address token1,address token2 ) external view returns (uint256,uint256){
    return (reserves[token1][token2],reserves[token2][token1]);
}
```

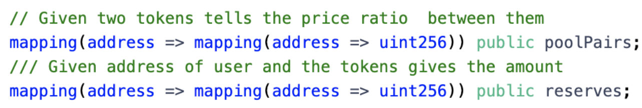
4.2.3 Contract memory

poolPairs

It is a double mapping that allows the token pair and its price to be stored.

reserves

It is a double mapping which allows the token pair and its reserve to be stored.



```
// Given two tokens tells the price ratio  between them
mapping(address => mapping(address => uint256)) public poolPairs;
/// Given address of user and the tokens gives the amount
mapping(address => mapping(address => uint256)) public reserves;
```

5 Implementation

5.1 Tools

5.1.1 Metamusk

Metamask.io is a browser extension that allows users to securely store, manage, and access Ethereum-based tokens and decentralized applications (Dapps). It is a bridge between the Ethereum blockchain and the user's browser, allowing users to interact with dApps without having to run a full Ethereum node. It will be used in the FlashLoan Dapp platform to carry out transactions.

5.1.2 React

React is a JavaScript library for building user interfaces. It is powerful because it allows developers to create complex user interfaces with minimal code. React also makes it easy to create reusable components, which can be used across multiple projects. Additionally, React has a large community of developers who are constantly creating new tools and libraries to make development easier and faster.

5.1.3 CSS-HTML-Javascript paradigm

The CSS-HTML-Javascript paradigm is a web development approach that combines the three core technologies of the web: HTML, CSS, and Javascript. HTML is used to structure content on a web page, CSS is used to style the content, and Javascript is used to add interactivity and dynamic behavior. This approach allows developers to create powerful and engaging websites that are accessible across all devices.

5.1.4 Web3.js

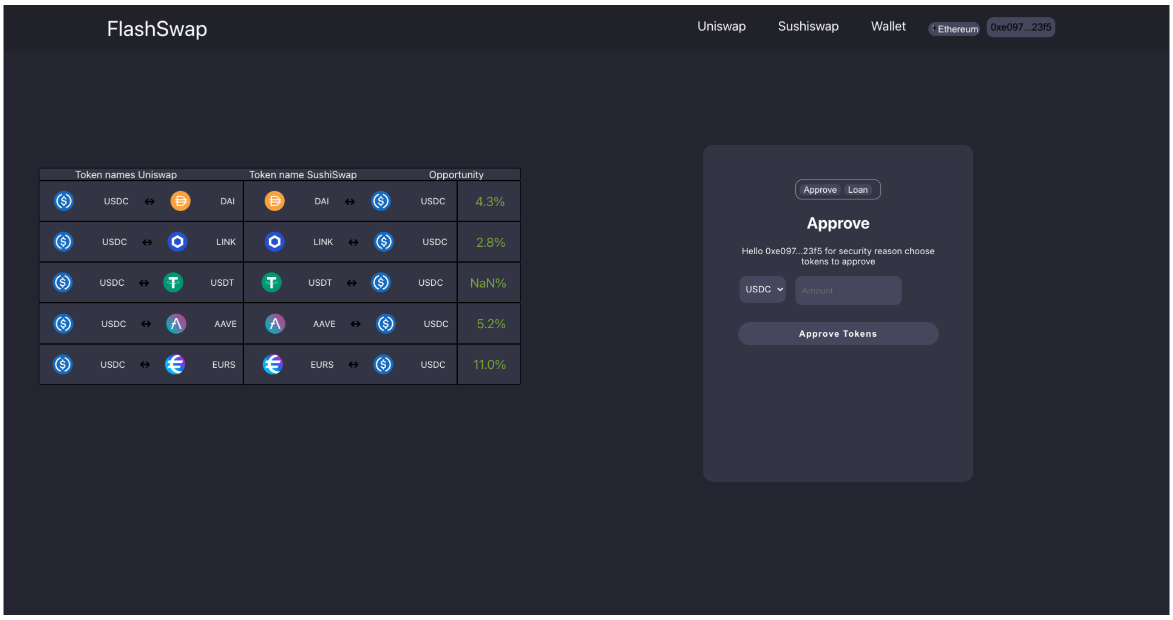
Web3.js is a collection of libraries which allow you to interact with a local or remote Ethereum node using an HTTP, IPC or WebSocket connection. It provides an API for interacting with smart contracts and the blockchain more generally. It is useful because it allows developers to easily interact with the Ethereum blockchain and build decentralized applications (dApps).

5.2 FlashSwap Dapp

The FlashSwap page consists of the main page in which the user can perform all operations. It consists initially of a header in which the user can log in via metamusk and consult the other pages that make up the application. In the left section the user can consult all available arbitrage options that are calculated by the system. The application calls the two decentralized exchange smart contracts, find pools price and calculate the most profitable arbitrage operation. The user can choose on the most profitable option and at that point the system saves those settings to make arbitrage trades. In the right section we have react components (cards) set up to call the flashLoan.sol In the first card we have the Approve function in which the user can delegate the smart contract to spend the tokens. The second card is intended to make flash loans. Finally there is a wallet section where the user can see his balance.

5.2.1 Workflow

- The user chooses in the left section the most profitable trade.
- He then uses the right section to approve the amount of tokens to be delegated to the contract.
- Finally he selects the value to be borrowed and calls and executes the trade.



Token names Uniswap	Token name SushiSwap	Opportunity
USDC ↔ DAI	DAI ↔ USDC	4.3%
USDC ↔ LINK	LINK ↔ USDC	2.8%
USDC ↔ USDT	USDT ↔ USDC	NaN%
USDC ↔ AAVE	AAVE ↔ USDC	5.2%
USDC ↔ EURS	EURS ↔ USDC	11.0%

Hello 0xe097...23f5 for security reason choose tokens to approve

Figure 12: The figure above show the flashSwap front-end application

5.3 Uniswap Sushiswap Dapp

These two sections of the application provide access to the decentralized exchanges Uniswap and SushiSwap. It initially consists of a header where the user can login via metamusk and browse the other pages that make up the application. We created a React component for each functionality of the decentralized exchanges. One to approve tokens, one to swap ,one to deposit funds and one to create the pool.

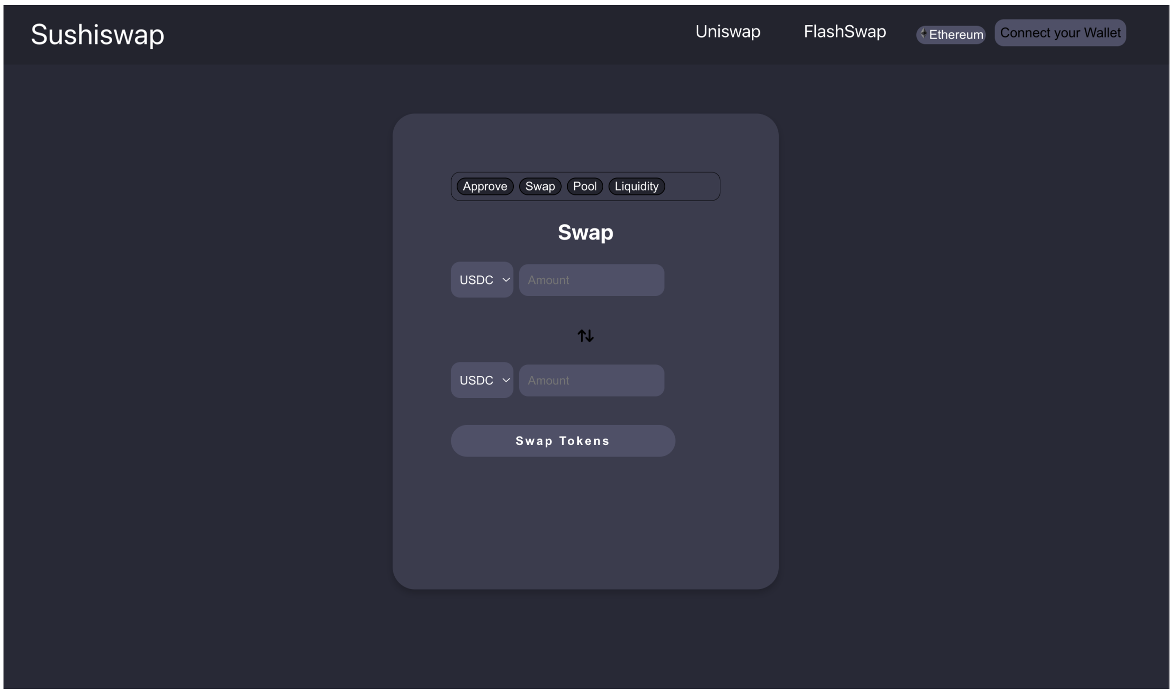


Figure 13: The figure above show the flashSwap front-end application

6 Conclusions

The application that was presented in this paper , FlashLoan Swap, proposes an innovative system for ERC-721 token arbitrage using a decentralized application called AAVE and a transaction type called FlashLoan. The application, as we have reiterated within this paper allows the end user to borrow tokens from AAVE and use them for arbitrage transactions between the two decentralized exchanges to make a profit.

6.1 Future Deployment

For future development, the two decentralized exchanges could be modified by adding oracles on pool prices and implementing the AMM used by Uniswap. Or connect the application to the main Ethereum network and use the official exchanges to carry out arbitrage transactions.

References

[1] IBM. Blockchain overview. https://www.ibm.com/topics/what-is-blockchain, 2022.

[2] Minimalsm. Intro into ethereum. https://ethereum.org/en/developers/docs/intro-to-ethereum/, 2019.

[3] Wackerow. Smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/, 2022.

[4] AAVE Foundation. Aave flashloan. https://docs.aave.com/developers/guides/flash-loans, 2022.

[5] AAVE Foundation. Aave v3 overview. https://docs.aave.com/developers/getting-started/readme, 2022.

[6] Uniswap. Uniswap smart contract overview. https://docs.uniswap.org/contracts/v3/overview, 2022.

[7] Jacky Yap. What is an automated market maker (amm) like uniswap? https://chaindebrief.com/what-is-an-automated-market-maker-amm-like-uniswap/, 2022.

[8] Word Coin. All you need to know about crypto arbitrage trading. https://worldcoin.org/articles/crypto-arbitrage, 2023.

[9] DHRUV PARMAR. Lending and borrowing in cryptocurrency (crypto loans) explained. https://geekflare.com/crypto-lending-and-borrowing, 2022.

[10] Coinbase.com. What is a dex? https://www.coinbase.com/it/learn/crypto-basics/what-is-a-dex, 2022.

[11] Metamask. Introduction. https://docs.metamask.io/guide/#why-metamask, 2022.

[12] Web3. web3.js - ethereum javascript api. https://web3js.readthedocs.io/en/v1.8.2/, 2022.

[13] Ingo Lütkebohle. BWorld Robot Control Software. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2008.