**Hanoi University of Science and Technology**

**School of Information and Communications Technology**

**o0o**

# Blockchain And Applications

# Borrowing and Lending Decentralized Application

**Supervisor**:
Nguyen Binh Minh

**Group of students:**

| **Full Name** | **Student ID** |
|---|---|
| Nhu Duc Minh | 20214966 |
| Tran Minh Tuan | 20214978 |
| Dang Manh Cuong | 20214949 |

**June 2, 2024**

# Contents

## Abstract

Aave is a decentralized finance (DeFi) protocol built on the Ethereum blockchain, launched in 2020. Its market value, calculated by multiplying the number of tokens by their price, peaked at 7.58 billion in May 2021 and has since stabilized at around 1 billion as of May 2022.

LPs, or Lending Pools, are virtual spaces where users can deposit and borrow assets, sending specific transactions to a smart contract for processing. While deposits have no restrictions, borrowing requires users to provide collateral to cover their loans. Aave also offers additional features like repayment, redemption, and liquidation, as well as flash loans that don't require collateral since the borrowed amount must be repaid in the same transaction.

This project aims to provide a simplified implementation of Aave, focusing on its core functions: deposit, borrow, repay, and withdraw. It will highlight when these functions can be executed and how they impact the lending pool and user balances.

# 1 Introduction

## 1.1 Key Concept

**Decentralized Finance**: DeFI is a financial system based and living on blockchain technology. In DeFI, contrary to the classical finance, all actions are performed by users - that have direct and total control of their resources - without of need for any trusted central authority.

**Lending Pools and functionalities**: In DeFi, a LP can be considered a smart contract towards which users can send transactions in order to lend and borrow crypto-assets, trusting the contract without a central entity. In general, a LP contains different reserves represented by other smart contracts handling these assets. On the one hand, users that deposit their assets increase the liquidity of the reserve (and so of the LP), on the other hand, this liquidity can be borrowed by other users that must deposit – in the lending pool – collateral to cover their borrows.

**Collateralization**: A collateral is an amount of crypto-asset that a user must deposit in the Lending Pool as security for his loans. A borrower can open many loans, the only constraint is that he has enough collateral. For this purpose, the health factor is used in order to establish if he can borrow anymore or if his position can be liquidated. The health factor is a parameter that depends on the user's total borrows, the user's total collateral, and on token's price.

A borrow position of a user can be liquidated when the health factor is under a particular threshold (typically 1). In a liquidation scenario, a liquidator repays a part of a user's borrow, he receives a part of the collateral of the user under liquidation and a bonus (typically in percentage).

When a loan is completely repaid by the borrower (amount to borrow + interests), the Lending Pool returns the amount deposited as collateral.

**Interest**: Interests are an amount of crypto-asset that lenders receive for their deposits and borrowers must return for their borrows. In both cases, interests depend on the interest rate that is influenced by the utilization of the reserve as follows:

- When the reversed is underused (i.e. a lot of liquidity is available) interest rate for borrowers decreases because LP wants to incentivize users to borrow, and interests for lenders decrease to disincentivize depositing
- When a reserve is overused (i.e. liquidity is scarce) interest rate for borrowers increases, and lenders are incentivized to deposit assets by high-interest rate in order to provide more liquidity.

## 1.2 Defintion

**Loan to value**: The loan to value (LTV) is a parameter, typically expressed in percentage, indicating the maximum amount of crypto-asset that a user can borrow with a certain amount of collateral.

**Liquidation threshold**: The liquidation threshold (LT) is a parameter, typically expressed in percentage, under which a loan is considered undercollateralized and so eligible for liquidation.

**Optimal utilization rate**: The optimal utilization rate defines the optimal usage of a reserve, typically expressed in percentage. A reserve is considered overused when the its utilization rate is above this parameter, underused when under it.

**Health factor liquidation threshold**: This parameter indicates the health factor threshold under which a user is eligible for liquidation (i.e. his collateral does not cover properly his debt).

**Liquidation bonus**: It is the bonus, expressed in percentage, that belongs to the liquidator that repays a part of a user's debt.

**Origination fee**: It is a fixed amount, expressed in percentage, that is applied instantly to every loan

**Tokens**: Tokens play a central role in DeFi and in Lending Pools. A token is a crypto-asset having value and living in a blockchain. Tokens can be minted, transferred, and exchanged both in Decentralized Exchanges (DEX) and in other contracts handling them. LPs can handle tokens, they sometimes are minted in order to keep track of users' actions or to redeem them.

## 1.3 Problem Statement

### 1.3.1 Overview

This project aims to develop a lending and borrowing protocol on the blockchain inspired by Aave. The protocol allows users to lend cryptocurrency to earn interest and borrow against deposited collateral. It includes core features such as depositing, withdrawing, borrowing, and repaying, while managing collateralization and interest calculation.

### 1.3.2 Key Features

Users can lend cryptocurrency to a liquidity pool, earning interest on deposits, and borrow cryptocurrency by providing collateral, paying interest on borrows.

### 1.3.3 Objective

The primary objective is to create a decentralized lending and borrowing platform using Solidity smart contracts on the Polygon blockchain (formerly known as Matic). The project aims to replicate the functionalities of Aave's protocol, providing users with opportunities to lend their digital assets and borrow against deposited collateral.

## 1.4 Stackholders

### 1.4.1 Lenders

: These are users who deposit digital assets into the lending pool with the intention of earning interest on the amount they lend. They contribute liquidity to the pool, facilitating borrowing activities.

### 1.4.2 Borrowers

: Users who require digital assets can borrow from the lending pool by providing collateral as security. They access liquidity provided by lenders, allowing them to fulfill their borrowing needs.

# 2 Main Feature of this work

The work proposed is called “ProtoAave”. It is a minimal prototype of Aave original implementation that proposes minimal actions about deposit, borrow, redeem, repay and liquidation, in order to understand how the state of the Lending Pool changes in response to specific transactions sent by users.

The following subchapters focus mainly on actors and their actions towards the Lending Pool contract of this work. All formulas that calculate interest rates, health factor, the amount of collateral needed to open a new borrow position, etc, have been taken from the original Aave’s implementation.

## 2.1 Actors

There are different actors involved in this work. All of them are represented by addresses and are:

- the “owner” of the Lending Pool: it is the address that deploys the contract. The owner can add a reserve (that is a contract that handles a particular ERC20 token) to the lending pool and initializes it, settings some parameters;
- the “price Oracle”, an address set by the “owner” that can modify ERC20 tokens’ price;
- users: they are addresses that mostly call the borrow and the deposit functions, manage their collateral, and query the Lending Pool in order to view its state.

## 2.2 Borrow Function

The borrow function is summarized by the pseudocode:

```solidity
pragma solidity 0.8.24;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract LendingPool{


    mapping(address => bool) public matic_isBorrower;
    mapping(address => bool) public ibt_isBorrower;
    uint256 public MTC_price;
    uint256 public matic_totalDeposits;
    uint256 public ibt_totalDeposits;
    mapping(address => uint256) public borrowable_amount;

    // ...
    ERC20 public mytoken_address;
    //...
    event Borrow(address indexed user, uint256 amount);

    function borrow_matic(uint256 _amount) external payable {
        require(!matic_isBorrower[msg.sender], "You have already borrowed funds! Clear Debt To borrow again!");
        require(matic_totalDeposits>_amount,"Not Enough Funds!");
        require(msg.value>_amount || (borrowable_amount[msg.sender]*(10**18)/MTC_price) >_amount,"Provide Collateral To continue the transaction");
        // ...
    }

    function borrow_ibt(uint256 _amount) external payable {
        ibt_totalDeposits = mytoken_address.balanceOf(address(this));
        require(!ibt_isBorrower[msg.sender], "You have already borrowed funds! Clear Debt To borrow again!");
        require(ibt_totalDeposits>_amount,"Not Enough Funds");
        require(msg.value>_amount || borrowable_amount[msg.sender]>_amount,"Provide Collateral To continue the transaction");
        // ...
    }

}
```

First, we check if the user has already borrow. If yes, alerting user for clearing debt for borrowing again. Otherwise, checking that the specified reserve has enough liquidity (int terms of the number of tokens). After that if the condition is satisifed, checking collateral available to continue the transaction.

After checking that user has enough collateral, the function updates the state of the reserve (interest rates and timestamps). Finally, it transfers the number of amountToBorrow tokens to the user, thanks to the Transfer method of the ERC20 contract.

**Notes**: IBT which we are minting to users accounts and is used for transactions as a second cryptocurrency (testing)

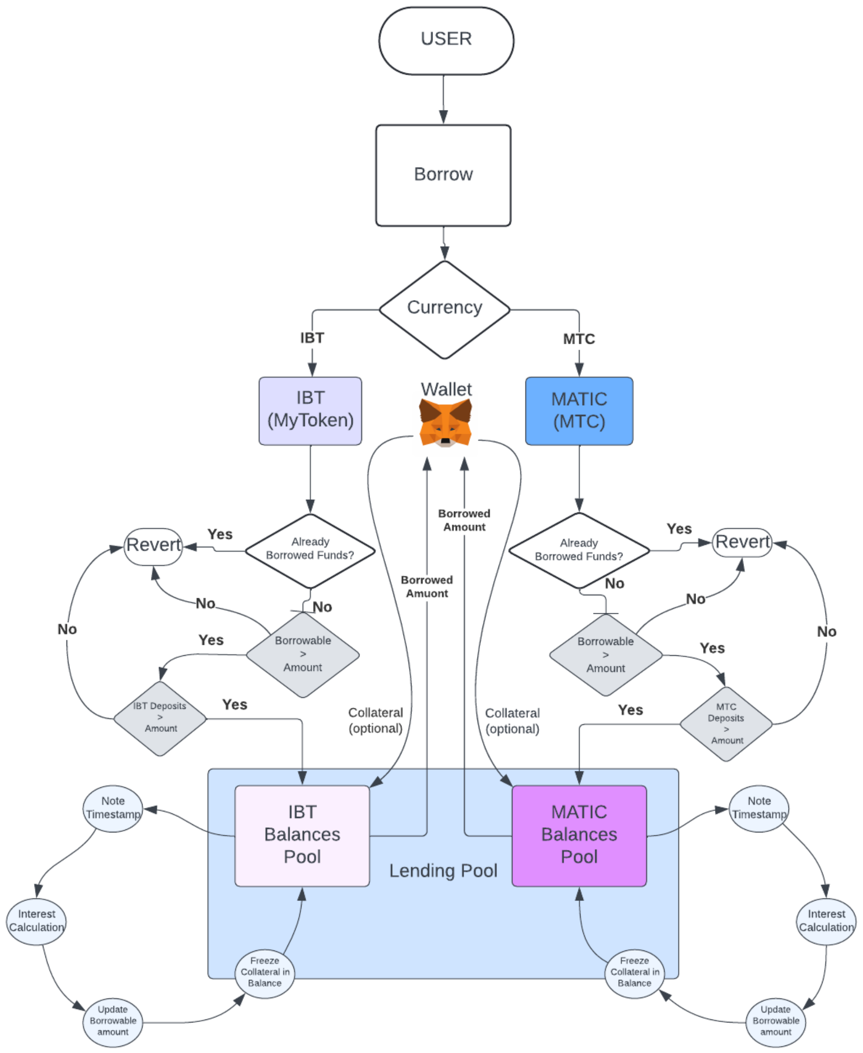

Figure 1: Borrow flowchart

## 2.3 Deposit Function

The deposit function is summarized by the pseudocode:

```

pragma solidity 0.8.24;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "./Erc20.sol";

contract MyLendingPool{

    uint256 public MTC_price;
    bool public ibt_isFirstWithdraw;
    bool public matic_isFirstWithdraw;
    event Deposit(address indexed user, uint256 amount, uint256
        tokensMinted);
    event BalanceAfterDeposit(address account, uint256 balance)
        ;
    event BalanceAfterWithdrawal(address account, uint256
        balance);
    uint256 public matic_totalDeposits;
    uint256 public ibt_totalDeposits;

    mapping(address => uint256) public matic_balances;
    mapping(address => uint256) public ibt_balances;

    mapping(address => uint256) public borrowable_amount;

    mapping(address => uint256) public matic_interest_balances;
    mapping(address => uint256) public matic_deposit_timestamp;
    mapping(address => uint256) public matic_accruedInterest;

    mapping(address => uint256) public ibt_interest_balances;
    mapping(address => uint256) public ibt_deposit_timestamp;
    mapping(address => uint256) public ibt_accruedInterest;

    AToken public token; // Token contract address
    ERC20 public mytoken_address;

    //...
    // Deposit funds into the lending pool and mint tokens
    function depositMATIC() external payable {
        uint256 _maticAmount = msg.value;

        MTC_price = 1.72 * (10 ** 18);
        require(_maticAmount > 0, "Amount must be greater than
            0");

        token.mintTokenswithMTC(msg.sender, _maticAmount); //
            Mint tokens directly to the user
        matic_balances[msg.sender] += _maticAmount;


        matic_totalDeposits += _maticAmount;
        if (matic_balances[msg.sender] == _maticAmount) {
            matic_isFirstWithdraw = true;
            borrowable_amount[msg.sender] = 0;
            matic_deposit_timestamp[msg.sender] = block.
                timestamp;
            matic_accruedInterest[msg.sender] = 0;
            matic_interest_balances[msg.sender] = _maticAmount;
        } else {

            matic_accruedInterest[msg.sender] += ((block.timestamp - matic_deposit_timestamp[msg.sender]) * matic_balances[msg.sender]) / 1000000;
            matic_deposit_timestamp[msg.sender] = block.timestamp;
            matic_interest_balances[msg.sender] += _maticAmount;
        }
        borrowable_amount[msg.sender] += (_maticAmount * (7) * (MTC_price)) /10**19;

        emit Deposit(msg.sender, _maticAmount, _maticAmount);
        emit BalanceAfterDeposit(msg.sender, matic_balances[msg.sender]);
    }

    function depositIBT(uint256 _tokenAmount) external {
        uint256 _ibtAmount = _tokenAmount;
        require(_ibtAmount > 0, "Amount must be greater than 0");
        require(
            mytoken_address.transferFrom(msg.sender, address(this), _ibtAmount),
            "Transfer failed"
        );

        token.mintTokensWithUSD(msg.sender, _ibtAmount); // Mint tokens directly to the user
        ibt_balances[msg.sender] += _ibtAmount;
        ibt_totalDeposits = mytoken_address.balanceOf(address(this));
        if (ibt_balances[msg.sender] == _ibtAmount) {
            ibt_isFirstWithdraw = true;
            ibt_deposit_timestamp[msg.sender] = block.timestamp;
            ibt_accruedInterest[msg.sender] = 0;
            ibt_interest_balances[msg.sender] = _ibtAmount;
        } else {
            ibt_accruedInterest[msg.sender] +=
                ((block.timestamp - ibt_deposit_timestamp[msg.sender]) *
                    ibt_balances[msg.sender]) /
                1000000;
            ibt_deposit_timestamp[msg.sender] = block.timestamp;
            ibt_interest_balances[msg.sender] += _ibtAmount;
        }

        emit Deposit(msg.sender, _ibtAmount, _ibtAmount);
        emit BalanceAfterDeposit(msg.sender, ibt_balances[msg.
            sender]);
    }

}
```

After checking that the amount to deposit is greater than zero, the function checks if the user allowed the Lending Pool to withdraw the amount, thanks to the "allowance" function of ERC20 contract.

Now, the Lending Pool calls the "TransferFrom" method of ERC20 contract, that transfers the amount from the msg.sender to the lending pool. An amount of "amountToDeposit" of aTokens are minted for the user. These aTokens provide the user can redeem them in the future.

Finally, the function updates interest rates and timestamps for the reserve and keeps track in a data structure if the user wants to use the reserve as collateral.

**Notes**: IBT which we are minting to users accounts and is used for transactions as a second cryptocurrency (testing)

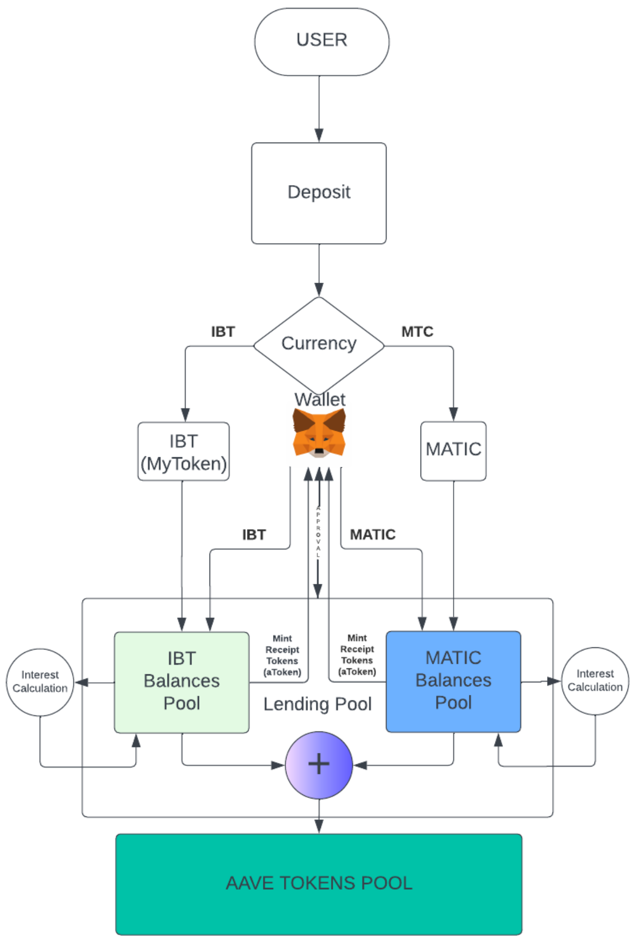

Figure 2: Deposit flowchart

## 2.4 Repay Function

The repay function is summarized by the pseudocode:

```
pragma solidity 0.8.24;


import "@openzeppelin/contracts/token/ERC20/ERC20.sol";


contract LendingPool{
    mapping(address => bool) public matic_isBorrower;
    mapping(address => bool) public ibt_isBorrower;

    mapping(address => uint256) public matic_borrowedAmounts;
    mapping(address => uint256) public ibt_borrowedAmounts;

    mapping(address => bool) public matic_isFirstRepay;
    mapping(address => bool) public ibt_isFirstRepay;

    mapping(address => uint256) public matic_timestamp_borrow;
    mapping(address => uint256) public ibt_timestamp_borrow;

    uint256 public MTC_price;

    uint256 public matic_totalDeposits;
    uint256 public ibt_totalDeposits;

    mapping(address => uint256) public matic_repayable_interest
       ;
    mapping(address => uint256) public ibt_repayable_interest;

    mapping(address => uint256) public total_borrowed;

    mapping(address => uint256) public borrowable_amount;

    ERC20 public mytoken_address;

    event Repay(address indexed user, uint256 amount);

    // ...
    function matic_repay() external payable{
        require(matic_isBorrower[msg.sender],"You have not
           borrowed any funds!");
        uint256 time_now;
        time_now = block.timestamp;

        if(matic_isFirstRepay[msg.sender]){
            matic_repayable_interest[msg.sender] = 0;
            matic_isFirstRepay[msg.sender] = false;
        }

        matic_repayable_interest[msg.sender] +=(time_now -
           matic_timestamp_borrow[msg.sender])*
           matic_borrowedAmounts[msg.sender]/100000;

        uint256 total_repayable;
        total_repayable = matic_repayable_interest[msg.sender]
           + matic_borrowedAmounts[msg.sender];

        matic_timestamp_borrow[msg.sender] = time_now;

        if (msg.value >= total_repayable) {


            matic_isBorrower[msg.sender] = false;
            if (msg.value > total_repayable) {

                // Return Extra funds and collateral
                payable(msg.sender).transfer(msg.value - total_repayable);




            }

            matic_totalDeposits += total_repayable;
            borrowable_amount[msg.sender] += matic_borrowedAmounts[msg.sender]*(MTC_price)/10**18;
            total_borrowed[msg.sender] -= matic_borrowedAmounts[msg.sender]*(MTC_price)/10**18;
            matic_borrowedAmounts[msg.sender] = 0;
            matic_repayable_interest[msg.sender] = 0;

        }
        else{
            matic_isBorrower[msg.sender] = true;

            if(matic_repayable_interest[msg.sender]<= msg.value){
                borrowable_amount[msg.sender] += (msg.value-matic_repayable_interest[msg.sender])*(MTC_price)/10**18;
                total_borrowed[msg.sender] -= (msg.value-matic_repayable_interest[msg.sender])*(MTC_price)/10**18;
                matic_borrowedAmounts[msg.sender] -= msg.value-matic_repayable_interest[msg.sender];
                matic_repayable_interest[msg.sender] = 0;

            }
            else{
                matic_repayable_interest[msg.sender] -= msg.value;
            }
            matic_totalDeposits += msg.value;


        }


    emit Repay(msg.sender, msg.value);



    }

    function ibt_repay(uint256 _amount) external{
        uint256 repay_amnt = _amount;

        require(ibt_isBorrower[msg.sender],"You have not
           borrowed any funds!");
        require(mytoken_address.transferFrom(msg.sender,
           address(this), repay_amnt));
        uint256 time_now;
        time_now = block.timestamp;

        if(ibt_isFirstRepay[msg.sender]){
            ibt_repayable_interest[msg.sender] = 0;
            ibt_isFirstRepay[msg.sender] = false;
        }

        ibt_repayable_interest[msg.sender] +=(time_now -
           ibt_timestamp_borrow[msg.sender])*
           ibt_borrowedAmounts[msg.sender]/100000;

        uint256 total_repayable;
        total_repayable = ibt_repayable_interest[msg.sender] +
           ibt_borrowedAmounts[msg.sender];

        ibt_timestamp_borrow[msg.sender] = time_now;

        if (repay_amnt >= total_repayable) {


            ibt_isBorrower[msg.sender] = false;
            if (repay_amnt > total_repayable) {

                // Return Extra funds and collateral
                mytoken_address.transfer(msg.sender, repay_amnt
                    - total_repayable);



            }

            ibt_totalDeposits = mytoken_address.balanceOf(
               address(this));
            borrowable_amount[msg.sender] +=
               ibt_borrowedAmounts[msg.sender];
            total_borrowed[msg.sender] -= ibt_borrowedAmounts[msg.sender];
            ibt_borrowedAmounts[msg.sender] = 0;
            ibt_repayable_interest[msg.sender] = 0;

        }
        else{
            ibt_isBorrower[msg.sender] = true;

            if(ibt_repayable_interest[msg.sender]<= repay_amnt){
                borrowable_amount[msg.sender] += repay_amnt-ibt_repayable_interest[msg.sender];
                total_borrowed[msg.sender] -= repay_amnt-ibt_repayable_interest[msg.sender];
                ibt_borrowedAmounts[msg.sender] -= repay_amnt-ibt_repayable_interest[msg.sender];
                ibt_repayable_interest[msg.sender] = 0;

            }
            else{
                ibt_repayable_interest[msg.sender] -= repay_amnt;
            }
            ibt_totalDeposits = mytoken_address.balanceOf(address(this));


        }


    emit Repay(msg.sender, repay_amnt);



    }
}
```

First, the function checks the user has an active borrow in the reserve. If the checks are satisfied, the function updates the state of the lending pool and the user, and finally transfers the amount to repay from msg.sender to the reserve of the LP.

When the repay function is successfully executed, the userToRepay's health factor increases allowing him to redeem the value of the collateral used in the loan just repaid.

**Notes**: IBT which we are minting to users accounts and is used for transactions as a second cryptocurrency (testing)

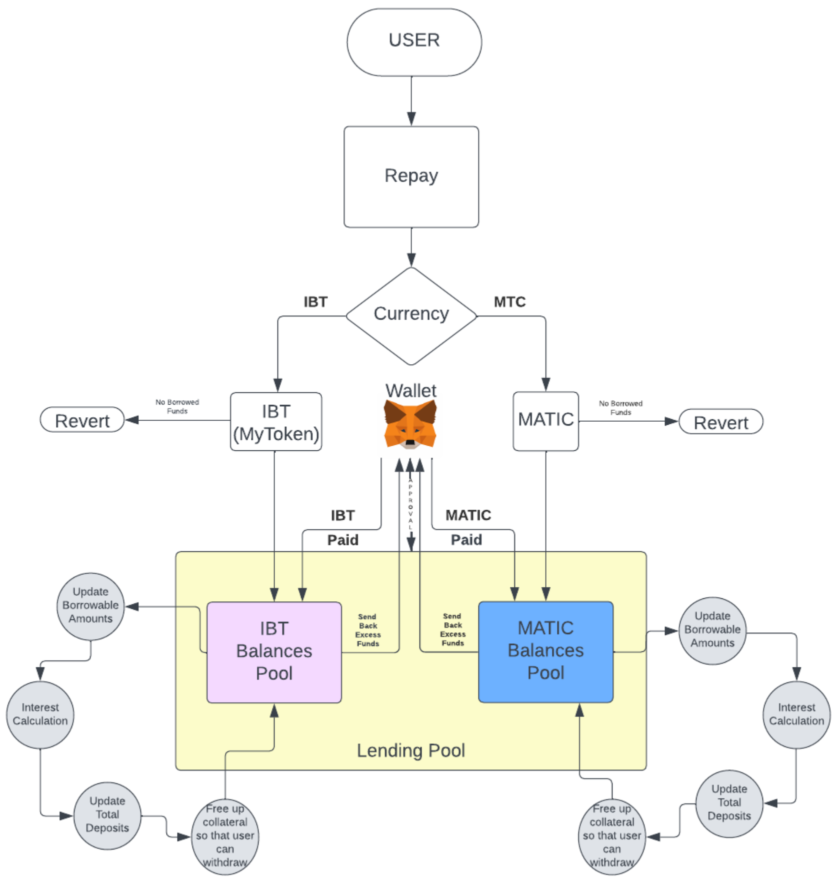

Figure 3: Repay flowchart

## 2.5 Withdraw Function

The withdraw function is summarized by the pseudocode:

```
pragma solidity 0.8.24;


import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "./Erc20.sol";


contract LendingPool{
    mapping(address => uint256) public matic_balances;
    mapping(address => uint256) public ibt_balances;

    mapping(address => uint256) public matic_interest_balances;
    mapping(address => uint256) public matic_deposit_timestamp;
    mapping(address => uint256) public matic_accruedInterest;
    mapping(address => uint256) public ibt_interest_balances;
    mapping(address => uint256) public ibt_deposit_timestamp;
    mapping(address => uint256) public ibt_accruedInterest;

    mapping(address => uint256) public ibt_withdrawInterest;
    mapping(address => uint256) public matic_withdrawInterest;

    bool public ibt_isFirstWithdraw;
    bool public matic_isFirstWithdraw;
    uint256 public ibt_totalDeposits;
    mapping(address => uint256) public total_borrowed;

    uint256 public MTC_price;
    uint256 public matic_totalDeposits;

    AToken public token; // Token contract address
    ERC20 public mytoken_address;
    mapping(address => uint256) public borrowable_amount;

    event Withdraw(address indexed user, uint256 amount);
    //...
    function withdrawIBT(uint256 _amount) external payable{
        uint256 time_now;
        time_now = block.timestamp;
        require(ibt_balances[msg.sender] >= _amount, "Insufficient balance");

        //require(mytoken_address.transfer(msg.sender, _amount), "Transfer failed");
        if (ibt_isFirstWithdraw) {
            ibt_withdrawInterest[msg.sender] = ((
                ibt_accruedInterest[msg.sender]
            ) +
                ((time_now - ibt_deposit_timestamp[msg.sender]) *
                    ibt_balances[msg.sender]) /
                1000000);
            mytoken_address.transfer(msg.sender,
                _amount + ibt_withdrawInterest[msg.sender]
            );
            ibt_withdrawInterest[msg.sender] = 0;

            ibt_interest_balances[msg.sender] -= _amount;
            ibt_isFirstWithdraw = false;
        } else if (!ibt_isFirstWithdraw) {
            ibt_withdrawInterest[msg.sender] = (((time_now -
                ibt_deposit_timestamp[msg.sender]) * ibt_balances[msg.sender]) /
                1000000);
            mytoken_address.transfer(msg.sender,
                _amount + ibt_withdrawInterest[msg.sender]
            );
            ibt_interest_balances[msg.sender] -= _amount;
            ibt_withdrawInterest[msg.sender] = 0;
        }

        ibt_deposit_timestamp[msg.sender] = time_now;
        token.burnTokensWithUSD(msg.sender, _amount);
        ibt_balances[msg.sender] -= _amount;
        ibt_totalDeposits = mytoken_address.balanceOf(address(this));

        emit Withdraw(msg.sender, _amount);
    }


    function withdrawMATIC(uint256 _amount) public {
        uint256 time_now;
        time_now = block.timestamp;
        require(matic_balances[msg.sender] >= _amount, "Insufficient balance");
        require((matic_balances[msg.sender]-_amount)*(7)*(MTC_price)/10**19>= total_borrowed[msg.sender], "Sori! Collateral depreciated :( ");
        //require(total_borrowed[msg.sender] == 0, "Sorry can’t withdraw until the debt is cleared.");
        if (matic_isFirstWithdraw) {
            matic_withdrawInterest[msg.sender] = ((
                matic_accruedInterest[msg.sender]
            ) +
                ((time_now - matic_deposit_timestamp[msg.sender]) *
                    matic_balances[msg.sender]) /
                1000000);
            payable(msg.sender).transfer(
                _amount + matic_withdrawInterest[msg.sender]
            );
            matic_withdrawInterest[msg.sender] = 0;

            matic_interest_balances[msg.sender] -= _amount;
            matic_isFirstWithdraw = false;

        } else if (!matic_isFirstWithdraw) {
            matic_withdrawInterest[msg.sender] = (((time_now -
                matic_deposit_timestamp[msg.sender]) *
                matic_balances[msg.sender]) / 1000000);
            payable(msg.sender).transfer(
                _amount + matic_withdrawInterest[msg.sender]
            );
            matic_interest_balances[msg.sender] -= _amount;

        }

        matic_deposit_timestamp[msg.sender] = time_now;
        token.burnTokenswithMTC(msg.sender, _amount);
        matic_balances[msg.sender] -= _amount;
        borrowable_amount[msg.sender] =
            (matic_balances[msg.sender] * (MTC_price) * 7) /
            10**19;
        matic_totalDeposits -= _amount+ matic_withdrawInterest[
           msg.sender];
        matic_withdrawInterest[msg.sender] = 0;
        emit Withdraw(msg.sender, _amount);
    }
}
```

About IBT, we just if the balance greater than withdraw amount. Then, it will put to the IBT Balances Pool and update the state of reserve (interest calculation) else revert the withdraw

**Notes**: IBT which we are minting to users accounts and is used for transactions as a second cryptocurrency (testing)

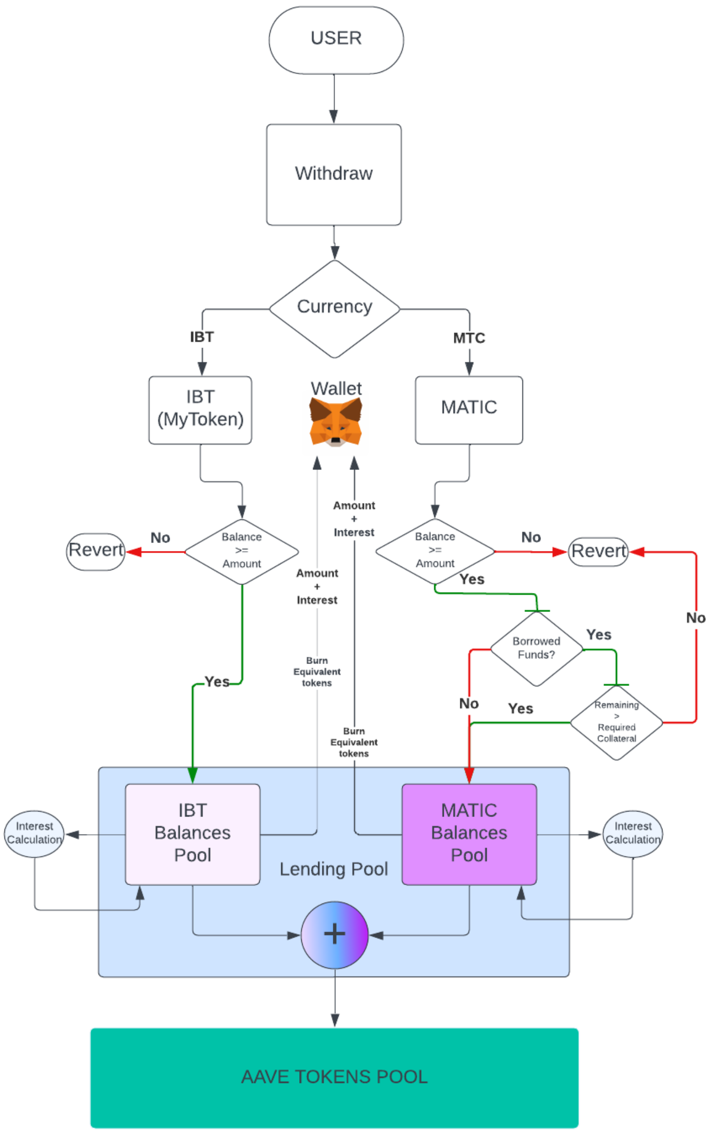

Figure 4: Withdraw flowchart

# 3 Detail Differences

## 3.1 Differences between borrow functions

- In Aave, both users' data and the collateral needed to cover a borrow position are computed by the "Data Provider" contract, while the "Fee Provider" contract provides fee calculus. In this work, these computations are made by the "Lending Pool" contract, using the same formulas proposed in Aave but with different data structures.

- The last difference regards an additional portion of code, in Aave, executed when the msg.sender wants to borrow at a stable rate. This part of code checks if the reserve is enabled for stable borrows, if msg.sender is allowed to borrow a stable rate, and the maximum amount msg.sender can borrow at a stable rate, that is 25% of the reserve's available liquidity.

## 3.2 Differences between deposit functions

In both implementations, after depositing, an number of aTokens are minted for the msg.sender. These tokens keep track of how much a certain user has deposited in the reserve. In Aave, aTokens are managed by a smart contract called "AToken", while in ProtoAave these tokens are stored in a double mapping called "aTokens" having type (address=>address=>uint256).

- In the Aave, when the msg.sender deposits in a reserve for the first time, the Lending Pool considers this deposit as collateral. On the contrary, in ProtoAave the msg.sender must indicate if wants to use the reserve as collateral or not, storing that in a particular data structure.

- In ProtoAave, before the Lending Pool transfers the amount of tokens from the msg.sender to its address, it checks (thanks to "allowance" method of ERC20) if msg.sender has allowed this transfer. In Aave this check is not executed.

## 3.3 Differences between repay function

The main difference between the repay function proposed in this work and the original implementation in Aave, is that in the proposed implementation it is possible only repay completely the debt, while in Aave the msg.sender can repay also only a part of it. In both implementations, the caller (msg.sender) can repay the debt of another user specifying the address, or the own debt specifying his address.

## 4 Conclusion

In this work, we have seen an overview of Lending Pools, how users can use them in order to handle their crypto-assets (ERC20 tokens in this case), and how Lending Pools help users in specific actions (deposit, borrow, repay, withdraw) without explicitly trusting each other, but trusting the smart contract that handles the LP. We have also seen the definition and a minimal implementation of flash loans that allow users to borrow crypto-assets without deposit collateral.

ProtoAave has been developed and has been proposed as a minimal implementation of Aave Protocol, it has highlighted when the actions can be executed, how users can manage their collaterals according to their health factors, and how interests for borrowers and lenders depend on time and on the utilization rate of a reserve.

Although this work differs from the original implementation of Aave, and differences have been shown, it provides a simpler way to understand the role of Lending Pools in Decentralized Finance. In the evaluation phase, some experiments have been shown using different libraries and including different cases in order to demonstrate the correctness of transactions.

## 5 References

[1] Github, *"Aave Protocol Version 1.0 - Decentralized Lending Pools"*. Available: `https://github.com/aave/aave-protocol`

[2] *Aave Protocol Whitepaper* Documentation. Available: `https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf.`.

[3] *Getting started with Hardhat*. Available: `https://hardhat.org/hardhat-runner/docs/getting-started#quick-start`.

[4] *Introducing the Amoy Testnet for Polygon PoS*. Available: `https://polygon.technology/blog/introducing-the-amoy-testnet-for-polygon-pos`.

[5] *Polygon Pos Chain Amoy Blockchain Explorer*. Available: `https://amoy.polygonscan.com/`.

[6], *AAVE tutorial (how to Lend & Borrow Crypto on AAVE)*. Available: `https://www.youtube.com/watch?v=2DfZXOijqcw`