Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Informatik - Technische Systeme

Bachelor-Arbeit

zur Erlangung des akademischen Grades

Bachelor of Science - B.Sc.

Study and Implementation of a Decentralized Application That Can Provide Permissionless Financial Services Using an EVM Based Blockchain

Vorgelegt von	Mario Alberto Maita Orozco
am	30.06.2022
Referent	Prof. Dr. Eva-Maria Iwer
Korreferent	Prof. Dr. Marc-Alexander Zschiegner

Erklärung gem. ABPO, Ziff. 4.1.5.4

Ich versichere, dass ich die Bachelor-Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 30.06.2022

Mario Alberto Maita Orozco

Erklärung zur Verwendung der Bachelorthesis

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Arbeit:

Verbreitungsform	**Ja**	**Nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger		×
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	×	
Veröffentlichung des Titels der Arbeit im Internet	×	
Veröffentlichung der Arbeit im Internet	×	

Wiesbaden, 30.06.2022

Mario Alberto Maita Orozco

Contents

1 Introduction **4**

1.1 Motivation 4

1.2 Goals 4

1.3 Thesis Structure 5

2 Fundamentals **6**

2.1 Blockchain 6

2.1.1 Cryptographic hash functions 7

2.1.2 Merkle tree 8

2.1.3 Consensus Protocol 9

2.1.4 Public-Key Cryptography 9

2.2 Ethereum 10

2.2.1 Accounts 10

2.2.2 Ethereum Virtual Machine (EVM) 11

2.2.3 Smart Contracts 13

2.2.4 Tokens 14

2.2.5 Decentralized Applications (Dapps) 14

3 Decentralized Finance (DeFi) **17**

3.1 Maker Protocol 18

3.2 Aave Protocol 18

3.2.1 Aave Rates and Risk Parameters 19

3.2.2 Tokenization: aTokens 21

3.2.3 Lending Pool 21
3.2.4 Aave Governance 22

4 Application Requirements 23
4.1 Introduction 23
4.1.1 Purpose 23
4.1.2 Scope 23
4.1.3 Product Perspective 24
4.2 Requirements 24
4.2.1 Functional 24
4.2.2 Nonfunctional 25
4.2.3 Use Cases 26

5 Implementation 28
5.1 Interaction with the Smart Contracts 28
5.1.1 The Pool Contract 29
5.1.2 The Address Provider Contract 30
5.1.3 The IERC20 Contract 31
5.1.4 The IWETH Contract 31
5.1.5 Node Provider 31
5.1.6 Web3.py and Brownie 31
5.2 Interaction with the users 32
5.2.1 Wallet Provider: Metamask 33
5.2.2 Web3-react 34
5.2.3 Web3 Library: ethers 34

6 Conclusion 38
6.0.1 Achieved Goals 38
6.0.2 Future work 39

Chapter 1

Introduction

1.1 Motivation

In the last years, blockchain technology has gained a lot of traction and many developers are willing to build applications on top of this technology. The Ethereum network with the Ethereum Virtual Machine, or EVM (whose state is validated and copied by every node of the network) offers a new way of application development. Meaning that developers can now implement some functionality and deploy it to a public EVM blockchain, making the code or functionality tamper-resistant; in addition, the implemented functionality (or smart contract) can be called by any participant or other contracts within the network.

This composability allows anyone to interact with already deployed contracts and build other applications on top of them. These kinds of applications are called *Decentralized Applications* or Dapps because they are built on top of a decentralized network such as Ethereum. A set of Dapps composes DeFi or *Decentralized Finance*, which replicates and goes beyond traditional financial infrastructure without any centralized institution, enabling for the first time in history access to financial services for anyone with an internet connection.

1.2 Goals

The goals of this thesis are to study and provide a better understanding of:

- The way that dapps are built.
- The used technology.

- Developing a Dapp that uses well-known DeFi protocols 3, to let users access permissionless financial services such as lending and borrowing of crypto assets.

1.3 Thesis Structure

The Thesis is divided into the following chapters:

- **Chapter 2: Fundamentals** gives a technical overview of blockchain technology, the Ethereum network, and decentralized applications.
- **Chapter 3: Decentralized Finance** introduces the concept of decentralized finance or DeFi, and outlines the protocols used in this thesis.
- **Chapter 4: Application Requirements** describes the functional and non-functional requirements that the proposed decentralized application for this thesis shall have.
- **Chapter 5: Implementation** outlines the used technology and the important implementation details of the thesis dapp.
- **Chapter 6: Conclusion** closes with a description of the goals achieved, the challenges faced during dapp development, and possible future work for the dapp.

Chapter 2

Fundamentals

2.1 Blockchain

A blockchain[14][20] is a type of database (append-only data structure), that stores data in blocks. These blocks are chronologically ordered by discrete timestamps and linked to each other using cryptographic hash functions. Commonly, a blockchain is used as a public distributed ledger of transaction records, shared and synchronized by a peer-to-peer network 2.1, where every party can participate in the validation of new blocks based on a consensus protocol, see 2.1.3.The idea of a cryptographically secured chain of blocks was originally presented by Haber and Stornetta[11] in 1991. However, the implementation and adoption of this technology started with the conception of the bitcoin cryptocurrency whitepaper[93] in 2008.

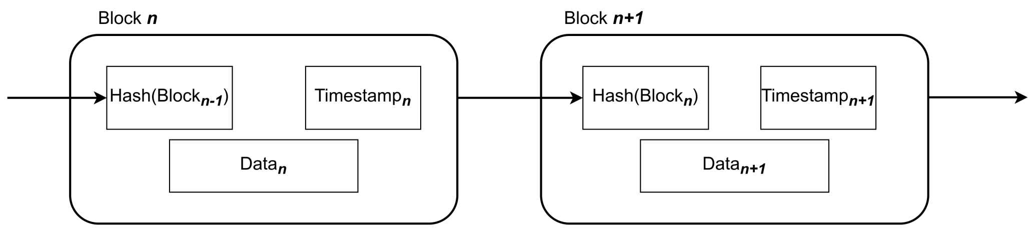


Figure 2.1: View of two blocks in a blockchain

Determined by the blockchain implementation, block contents can be different. A block usually has a timestamp, the data or transaction records, and the hash value of the previous block in the chain. See figure 2.1. The way that a blockchain guarantees the security and keeps the integrity of the data is through cryptographic hash functions 2.1.1, and Merkle trees 2.1.2

By adding the cryptographic hash value of the previous block (Block n to Block

$n + 1$), blockchains ensure the integrity of previous blocks. Because of the second preimage resistance of cryptographic hash functions 2.1.1, it is computationally infeasible to find an input other than $Block\ n$ in order to generate an output = $Hash(Block\ n)$. Due to that, the further the blockchain grows beyond block n, the more secure is the integrity of block n and previous blocks.

Depending on who can participate in the consensus protocol, blockchain networks can be divided into two groups: permissionless and permissioned blockchains. This thesis will be focused on permissionless blockchains that use an EVM 2.2.2, such as the Ethereum network 2.2.

Peer-to-peer (P2P) network

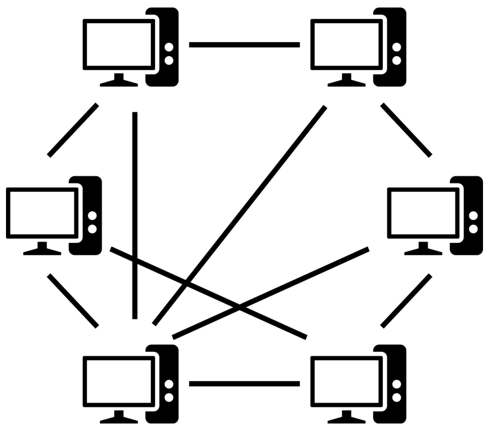


Figure 2.2: Peer-to-peer network[101]

A peer-to-peer network[2][101] is a decentralized communication model where each participant (node) of the network stores and shares data with other nodes without any central party. Every peer is equal and there is no special node. Due to the decentralized design, P2P networks are resilient and robust compared to typical client-server architectures. The distributed ledger or blockchain is managed by the peer-to-peer network based on a set of rules or consensus mechanisms 2.1.3. In fact, the term *blockchain network* refers to a set of nodes, of a peer-to-peer network, running the consensus protocol of the specific blockchain.

2.1.1 Cryptographic hash functions

A cryptographic hash function[17][12] maps a given data of variable size to a fixed length n-bit string called *hash value*. $H : \{0,1\}^* \rightarrow \{0,1\}^n$. Such hash functions have the following properties:

Collision resistance
: For two inputs (x_1, x_2), it should require $O(2^{\frac{n}{2}})$ computation power, such that $H(x_1) = H(x_2)$. This means, that it will be computationally infeasible that different inputs, i.e., blocks of data, will generate the same hash value.

Preimage resistance
: For a hash value h, it should require $O(2^n)$ computation power in order to obtain an x such that $H(x) = h$. In other words, it should be a one-way function. This means that it is computationally infeasible to retrieve the input data from its hash value. For instance, it is nearly impossible to get a private key from its public key.

Second preimage resistance
: For an input x_1 and its hash value h_1, it should require $O(2^n)$ computation power in order to obtain a x_2 such that $H(x_2) = h_1$. This means that for a given hash value, it is computationally infeasible to find another input, i.e., a block data input, that generates the same hash value.

Such properties prevent attackers from modifying existing blocks keeping the blockchain intact. For example, the bitcoin network uses the SHA-256[73] hashing algorithm to validate a block, and append it to the chain. This means, that once a consensus among all the participants is done, and a new block is added, $O(2^{256})$ computation power is required to tamper with the blockchain. In other words, it is nearly impossible to manipulate an added block as a result of the tremendous computer power that would be needed.

2.1.2 Merkle tree

A Merkle tree[13][9] or hash tree is a binary tree where every leaf node (nodes at the bottom of the tree) is the cryptographic hash value of some data or transaction. A merkle tree is constructed bottom-up; the upward nodes are the cryptographic hash values of its two child nodes. Therefore, the root node will contain a complete fingerprint of the entire set of transactions.

Merkle trees allow secure and efficient data verification using cryptographic hash functions. For example; if a participant in the network needs to verify the validity of these four transactions, see figure 2.3, only a check of the root hash value is necessary. Due to the Merkle tree structure, if any of the data blocks is modified, so will be the root hash value. Furthermore, Merkle tree is a very efficient data structure; for instance, checking that a given transaction is included in the tree will take a maximum of $2log(N)$ operations.

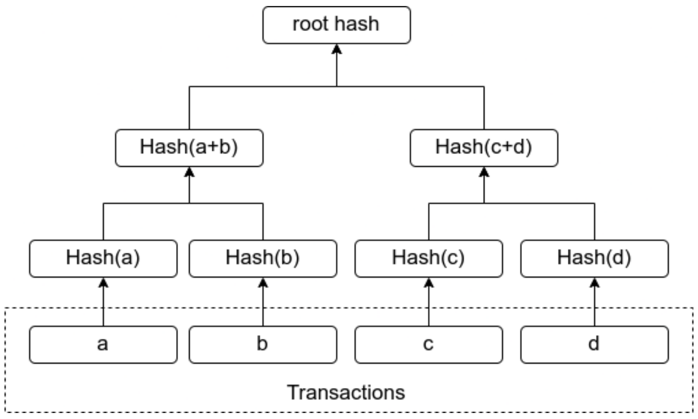


Figure 2.3: Merkle tree of four transactions

2.1.3 Consensus Protocol

The consensus protocol[20] of a blockchain network gives a specific rule for verifying whether a transaction is valid or not. As mentioned in 2.1, any participant or node of the network, depending on the blockchain type, can append a new block. For the reason that blockchains typically do not have a centralized authority validating transactions, participants on the blockchain must verify any transaction according to the set of rules or consensus protocol of the blockchain. The most common consensus mechanisms nowadays are:

Proof of work (PoW)
In a PoW blockchain, nodes have to solve a cryptographic task in order to validate a block; the first node to find a solution can submit the transaction. Most of the first blockchains in the blockchain space use proof-of-work-based protocols; for example, Bitcoin uses PoW based on the cryptographic hash function SHA-256.

Proof of state (PoS)
In a PoS blockchain, the entity that can validate a transaction is randomly selected, depending on the "stake" that a node has on the network. Since nodes have a large stake in the blockchain, they will pursue the integrity of the network.

2.1.4 Public-Key Cryptography

In order to interact with a blockchain, participants use public key cryptography or asymmetric key cryptography[7]. The public key acts as the ID of the sender or receiver of a given message; and it can be shared with others without any security downfall. On the other hand, the private key should be kept secure and private by the message sender; it is used to sign (digital signature) the messages or transactions, i.e., giving permission to change the state of the user records in the blockchain.

> **Digital signature:** The message or transaction can only be signed with a private key, and anyone knowing the public key of the signed message can verify its authenticity. Meaning that any party of the blockchain can participate in the verification of the transactions or messages.

The private key is a random number and the public key is generated from it using one-way cryptographic hash functions, see 2.1.1 or elliptic curve cryptography[7][95]. Elliptic curve cryptography (ECC) is based on the discrete logarithm problem[44] and is the most common public-key algorithm in blockchain networks. In mathematical terms, an elliptic curve is a plane curve over all the points (x, y) that satisfy the following equation: $y^2 = x^3 + ax + b$

2.2 Ethereum

The Bitcoin[93][1] network and its set of technologies enabled for the first time a decentralized, permissionless, trustless, peer-to-peer digital currency system. The distributed computation system that bitcoin introduced: a proof-of-work algorithm to produce a consensus in a distributed system without a central trusted authority, in combination with the blockchain storage system, solved the double-spend problem[45] and the Byzantine generals problem[37], enabling other applicability beyond digital currencies. In 2014, Vitalik Butering introduced Ethereum[10], a platform that moved beyond cryptocurrency applications.

Similar to Bitcoin, Ethereum[10][3] is a distributed transaction-based state machine. However, it is not only a cryptocurrency system. Ethereum is a decentralized computing platform or a decentralized, pseudo-turning complete virtual machine 2.2.2, which runs smart contracts and stores its state changes in its blockchain. Like bitcoin, state changes in Ethereum are managed by a consensus mechanism, where everyone can participate in the validation of blocks or state changes.

2.2.1 Accounts

Ethereum[10][18] has an account-based model, where every account represents a state. An account is mapped with a 160 bits address (public key) and there are two types of accounts: *Externally owned account* (EOA), controlled by a private key and (smart) *contract account*, controlled by EVM 2.2.2 code. Depending on the account type, an account state is represented by the following four fields: **Nonce** is a counter that represents the number of transactions or contracts created by the account. **Balance** is the amount of Ether owned by the account, expressed in Wei 2.1. **Storage hash** is the root hash of a merkle tree that represents the contents of the account. **Code hash** is the hash that points to the EVM code that the account has.

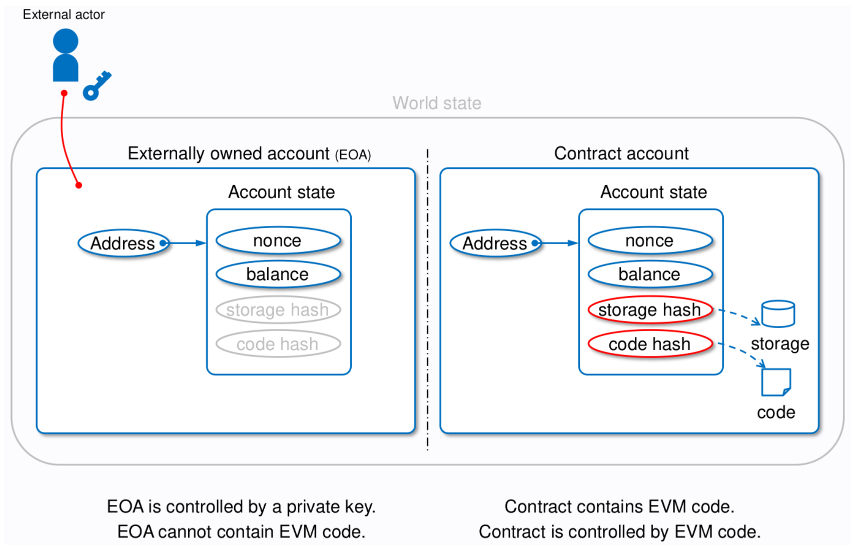


Figure 2.4: Ethereum accounts[94]

Transactions and Messages

A **transaction** is a cryptographically-signed message that can only be sent by an EOA. Transactions can trigger a state change on the blockchain or the execution of EVM code. Transactions can be divided into two groups:

- **Message calls**: The transactions between accounts.
- **Contract creations**: creation of a new account with EVM code in its data or code field.

Messages can be seen as internal transactions between contracts triggered by a message call. A message is like a transaction but produced by a contract. This way messages enable contracts to interact between them.

2.2.2 Ethereum Virtual Machine (EVM)

The EVM[18][4] is a quasi-Turing-complete machine; the quasi means that its computation is limited by the available **gas** 2.2.2 that the executed contract bytecode has, preventing that accidentally or malicious contracts execute forever. Thus, denial-of-service attacks are not possible on Ethereum. The EVM is a stack machine with 1024 elements. It is big-endian by design with 256-bit word size, which facilitates hashing and elliptic curve operations (Keccak-256 hashes and secp256k1 signatures). In addition to the **Stack**, the machine has two more spaces where operations can access and store data: **Memory**; volatile and initialized to zero. **Account storage**; non-volatile and maintained as part of the Ethereum state, also initialized to zero. Lastly, the EVM code is stored in a separate virtual read-only memory (**ROM**).

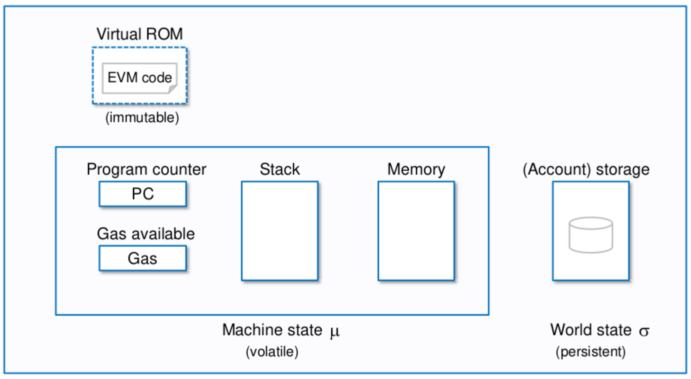


Figure 2.5: Ethereum Virtual Machine[94]

Ether and Gas

Ether (ETH) is the native cryptocurrency of Ethereum. Ether is used to pay for transaction fees or EVM computing power in the form of gas units. For example, the cost or gas limit for sending a transaction is 21000 gas units[19]. The price for a gas unit is represented in gwei, and is set by the participants of the consensus protocol (miners), based on the supply and demand of the network computational power[5]. For instance, with an average gas price of 15 gwei[51], sending a transaction at the moment of writing this thesis would cost: $21000 * 15 = 315000$ *Gwei* or 0.000315 *Ether*.

Unit Name	**Wei Value**
Ether	1^{18}
Gwei	1^{9}
Wei	1

Table 2.1: Important metrics of *Ether*

2.2.3 Smart Contracts

In the Ethereum network, a smart contract[6][57] is a computer program that is executed by the EVM. Since smart contacts are also a type of account, see 2.2.1, they have a balance and can send transactions in the form of messages. Furthermore, smart contracts have the following properties:

Immutability
When a smart contract is deployed to the Ethereum platform, the code can not be changed. If a modification (fixing a bug or adding new functionality) needs to be made, a new smart contract needs to be deployed.

Determinism
Given an input and the blockchain state, the result of the execution of the smart contract has to be the same for everyone who calls the contract.

Limitations
By design, smart contracts can not send $HTTP$ requests; this can be solved using decentralized oracles[66]. Furthermore, the context of execution is limited to the EVM, meaning that they can only access their own state and the transaction context of the transaction caller. Lastly, smart contracts can not exceed the size of $24KB$.

Lifecycle
Once deployed, smart contracts can not be modified. However, they can be deleted, if the smart contract was programmed with the `selfdestruct()` function.

Composability
Since smart contracts can call other smart contracts, functionality of different contracts can be combined. However, such interaction between contracts has to be triggered by a transaction sent by an EOA 2.2.1.

Permissionless
Anyone can deploy a smart contract to the Ethereum network.

Smart contracts can be written in EVM bytecode[48]. However, they are usually written in high-level programming languages such as *Solidity*, *Vyper*[88] and *Fe*[63], among others, being Solidity, the most widely used programming language for smart contracts in EVM-based blockchains.

Solidity

Solidity[6][75] is an object-oriented or contract-oriented high-level programming language; it was created by Dr Gavin Wood[52] and designed to target the EVM. The Solidity compiler *solc* converts Solidity contracts into EVM bytecode and also generates the contract application binary interface (ABI)[38], which enables the interaction with contracts in both directions; from outside the blockchain and from contract to contract.

2.2.4 Tokens

Tokens have a widely used description[102], in the context of Ethereum; a token[82] is a digital representation or abstraction of something that lives on the Blockchain. For instance, tokens can represent currencies, shares in a company, or even a virtual pet. Unlike Ether, which is managed by the Ethereum protocol, tokens are created and handled by smart contracts. Everyone can create a token. Nonetheless, because a smart contract can be programmed without any restriction, there are standards that need to be followed for the creation of tokens. The two most used standards are:

- The **ERC20** Token Standard[92], for the representation of equivalent and interchangeable tokens (fungible tokens). For instance, a token that represents a currency or a share in a company. As defined in the EIP[92], the ERC20 contract is composed of the following elements:

Must have methods	Optional methods	Must have events
`totalSupply()`	`name()`	`Transfer()`
`balanceOf()`	`symbol()`	`Approval()`
`transfer()`	`decimals()`	
`transferFrom()`		
`approve()`		
`allowance()`		

Table 2.2: Components of the ERC20 Standard.

- The **ERC721** Token Standard[103], for non-fungible tokens or representation of unique goods like an ID or collectibles.

2.2.5 Decentralized Applications (Dapps)

Unlike a web application, a decentralized application or Dapp[8][56][77] has its backend running on a decentralized peer-to-peer network such as Ethereum. For dapps running in an EVM-based blockchain, the backend is smart contracts executed by the EVM. This means, that a dapp has the same properties as smart contracts and the blockchain network where it lives.

Dapp architecture

The architecture of dapp is still been defined and depends, like any application, on the services that the application should provide. However, there are some main components that a dapp should have:

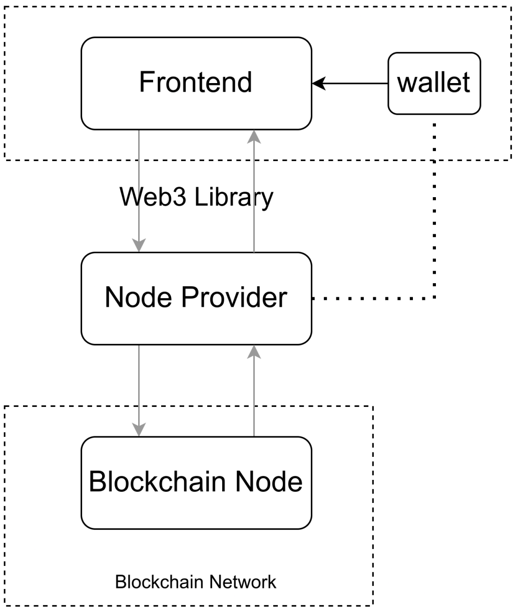


Figure 2.6: Basic dapp architecture.

Frontend: It enables the users to interact with the smart contracts through interactive graphical components such as buttons, and icons, among others. Like web applications, this user interface runs on any browser.

Wallet: A wallet is a tool or software application that manages the account's private key and addresses (public keys). Wallets don't have custody of the account crypto assets. A wallet is only an application that enables the account owner to sign transactions, therefore, interact with the blockchain and the smart contracts living in the network.

Node or Node Provider: Enables communication with the blockchain, i.e., reading data or sending transactions. A node will broadcast the transactions and miners will validate the new states that the transactions can produce. Everyone can run a node. Nonetheless, because of the difficulty and costs that running a node could have, Dapps usually rely on a third-party service or Node provider. Every Ethereum client or Node provider implements a JSON-RPC[72] Specification with a uniform set of methods.

The Blockchain Network: is the base layer, where the EVM executes the dapp smart contracts and stores the new states in the blockchain.

Furthermore, the Comunication with the Node provider is done using web3 libraries such as `ethers.js` 5.2.3 or `web3.py` 5.1.6.

Chapter 3

Decentralized Finance (DeFi)

Decentralized finance or DeFi[15][16] refers to a set of decentralized applications and protocols that are focused on financial services. The term finance[96] involves the creation, and management of money, in traditional finance systems this is done by financial institutions[97], which emit, buy and sell financial instruments[98] on financial markets[99], all regulated by Laws. In DeFi, these practices and processes are determined by protocols that rely on smart contracts, which means Defi is an open, permissionless, and composable stack of protocols built on top of a permissionless blockchain such as Ethereum 2.2.

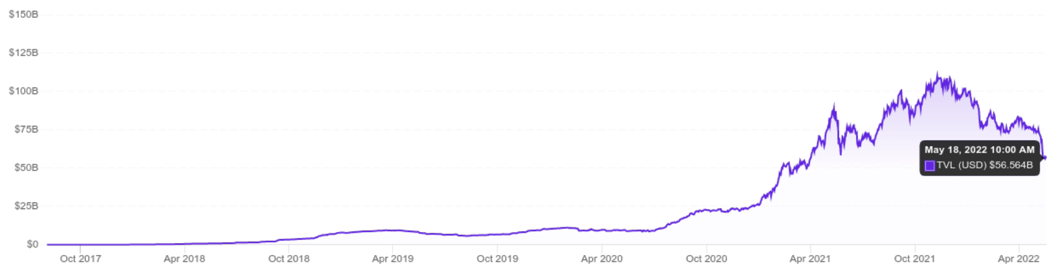


Figure 3.1: Total Value Locked (USD)[42]

DeFi offers a wide variety of financial services. For example, the lending and borrowing of crypto-assets, exchange of crypto assets, stable currencies, and investment opportunities, among others. Unlike traditional financial systems, it is open to anyone who has internet access; users have total control of their assets and markets are always open. Furthermore, DeFi has gained a lot of traction in the past years and according to Defipulse[42], at the moment of writing this thesis, the total locked value or TVL (held by smart contracts) in all the DeFi applications and protocols is $58.56B 3.1, being the Maker Protocol 3.1, the Aave Protocol 3.2, and the Uniswap Exchange[85], the three relevant protocols of the DeFi space with $14.52B, $8.38B and $7.04B of TVL respectively.

3.1 Maker Protocol

One essential piece of DeFi is stable coins or stable cryptocurrencies, and DAI is one of the most popular stable coins in the DeFi space with a market cap of $6.5B[83] at the moment of writing. Stable coins are cryptocurrencies pegged to a predetermined value, usually to the USD value. It enables DeFi users to access more traditional assets, avoiding the natural volatility of crypto assets at the current moment. In addition, due to the high demand for stable coins in the DeFi space, users can also earn interest in their stablecoins by providing liquidity into lending pools of different protocols such as Aave 3.2 or Compound[78].

The Maker Protocol[79] is an open-source project built on Ethereum that enables users to generate the stable coin DAI. Wich is collateral backed by different crypto assets authorized by the MakerDao, a decentralized autonomous organization (DAO)[41], made up of all the Maker governance token (MKR) holders. The MKR token is an ERC20 token 2.2.4 that enables dao participants to vote on different changes or improvements to the Maker protocol.

DAI

DAI[80] is the main product of the Maker Protocol. DAI is an ERC20 token 2.2.4, pegged to the USD value and fully backed by different crypto assets, approved by the MakerDao, such as ETH 2.2.2. DAI can be generated by depositing any accepted collateral crypto asset into the Maker Vaults. In short, the Maker Vaults are a set of smart contracts 2.2.3 that lock the deposited asset and generate DAI (let the user borrow DAI). The generated DAI is destroyed by the vault when the generated DAI is repaid to the Vault. Vaults are overcollateralized, for instance, at the time of writing, the Maker ETH-C Vault has a minimum collateral ratio of 170%[65], meaning that for every $170 of ETH deposit in the Vault a maximum of 100 DAI can be generated.

3.2 Aave Protocol

Aave[22][27][29] is a DeFi liquidity protocol that enables users to lend and borrow crypto assets. Aave was first deployed on the Ethereum network, however, at the moment of writing, Aave has expanded to other EVM-based blockchain and Layer 2 scaling solutions such as Polygon[35], Avalanche[40], Fanton[49], and Arbitrum[32], among many others.

Using Aave, users that supply liquidity to the protocol (lenders) earn interest on their deposited assets, and borrowers are able to borrow crypto assets after depositing collateral to the pool contract. The interest rate for users (borrowers and lenders) is decided algorithmically based on the reserves available in the pool. The **reserves**, see fig 3.2, are the multiple currencies deposited on the pool expressed in ETH value and defined as *total liquidity*. Lenders can deposit into these reserves to earn interests and Borrowers can borrow from the reserves after making a deposit, locking a greater value as collateral for the borrowed funds. Only specific low-risk crypto assets are configured as collateral.

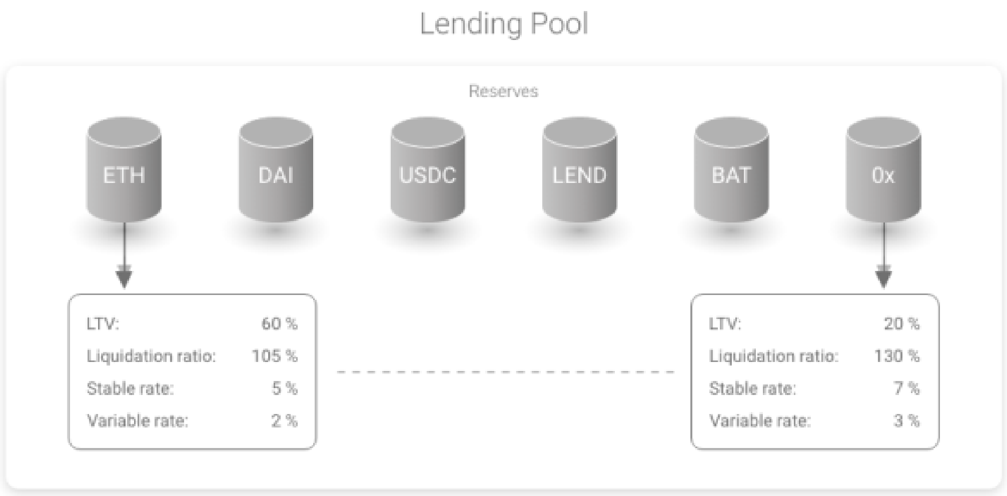


Figure 3.2: Pool Parameters[22]

3.2.1 Aave Rates and Risk Parameters

As mentioned in the last paragraph, the interest rate, for lenders and borrowers is determined by the status of the specific reserve. For **lenders**, the interest rate depends on the *current liquidity rate*:

$$R_t = R_o U$$

Where R_o is the overall borrow rate of the reserve defined as follows:

$$R_o = \begin{cases} 0 \ if \ \ B_t = 0 \\ \frac{B_v R_v + B_s R_{sa}}{B_t} \ \ if \ \ B_t > 0 \end{cases} \tag{3.1}$$

With the total amount of borrowed liquidity B_t (*total borrows*), expressed as the sum of the total stable borrows plus the total variable borrows :

$$B_t = B_s + B_v$$

And U is the *utilization rate*, which is the representation of the utilization of the deposited assets, defined as follows:

$$U = \begin{cases} 0 \ \ if \ \ L_t = 0 \\ \frac{B_t}{L_t} \ \ if \ \ L_t > 0 \end{cases} \tag{3.2}$$

Where L_t is the *total liquidity* available in the reserve.

Borrowers can borrow crypto assets with variable and stable interest rates. The R_v[23] is the *variable borrow rate* and R_{sa}[24] is the *average stable rate*.

Note that the mentioned definitions are from the V1[22] whitepaper held for the V2[27] and the recently V3[29] release.

Loan-To-Value (LTV)

A user will be able to borrow a maximum amount depending on the Loan-To-Value (LTV)[22][71] of the desired asset reserve. For instance, if the LTV value of a given asset is 60%, see fig 3.2, for every 1 ETH value of collateral deposited, a user can borrow a maximum of 0.6 ETH value of the desired asset reserve. The max LTV is defined as follows:

$$maxLTV = \frac{\sum Collateral_i \ ETH \ \times LTV_i}{Total \ Collateral \ in \ ETH}$$

Liquidations

[22][60] To maintain the solvency of the protocol, anyone can liquidate a borrow position i.e. buy a maximum of 50% of the borrow position. Every liquidated position has a liquidation bonus, which depends on the asset. For instance, if someone borrows 1 ETH worth of DAI and its H_f 3.2.1 drops below 1, anyone can liquidate the position (max 50%) for a bonus of 5% (105% liquidation ratio, see fig 3.2) at the movement of writing, the liquidator can claim up to 0.5 + 0.05 ETH by repaying 0.5 ETH worth of DAI.

Liquidation Threshold

Because of price fluctuations in the market, a borrow position can become close to the collateral position. The percentage at which a borrow position can be liquidated is called the liquidation threshold[22][71] . For instance, if the liquidation threshold of a given asset is 80% and the borrow position value surpasses 80% of the collateral value, the position can be liquidated. The liquidation threshold is defined as follows:

$$Liquidation\ Threshold = \frac{\sum Collateral_i\ in\ ETH\ \times LiquidationThreshold_i}{Total\ Collateral\ in\ ETH}$$

The $LiquidationThreshold_i$ for a given asset is defined and approved by the aave governance 3.2.4.

Health Factor

In order to maintain solvency in the protocol, every user or account has a health factor[22][71] , which indicates if the borrow position of the user can be liquidated. If $H_f < 1$, the borrow position can be liquidated.

$$H_f = \frac{\sum Collateral_i\ in\ ETH\ \times LiquidationThreshold_i}{Total\ Borrows\ in\ ETH}$$

Ray Math

To avoid rounding errors, Aave uses the Ray Math Concept[55][46], where a Ray is a unit with 27 decimals of precision. All the rates are expressed in Ray.

3.2.2 Tokenization: aTokens

Aave Tokens or aTokens[26][28] are ERC20 tokens that liquidity providers (lenders), and borrowers, receive once the deposit or borrow transaction is processed; aTokens maps 1:1, the deposited or borrowed asset, also known as the underlying asset. For instance, if a user deposits 100 DAI, 100 aDAI is sent to its account 2.2.1. For lenders, the balance of a specific aTokens grows depending on the interest rate 3.2.1 of the underlying asset.

3.2.3 Lending Pool

At the heart of the Aave protocol is the `lendingPool.sol` or `pool.sol`, renamed in the V3[29] release. The pool handles the most important actions of the protocol.

Users and developers can interact with the pool contract to deposit, borrow, and or repay a borrow position, among other actions, see fig 3.3.

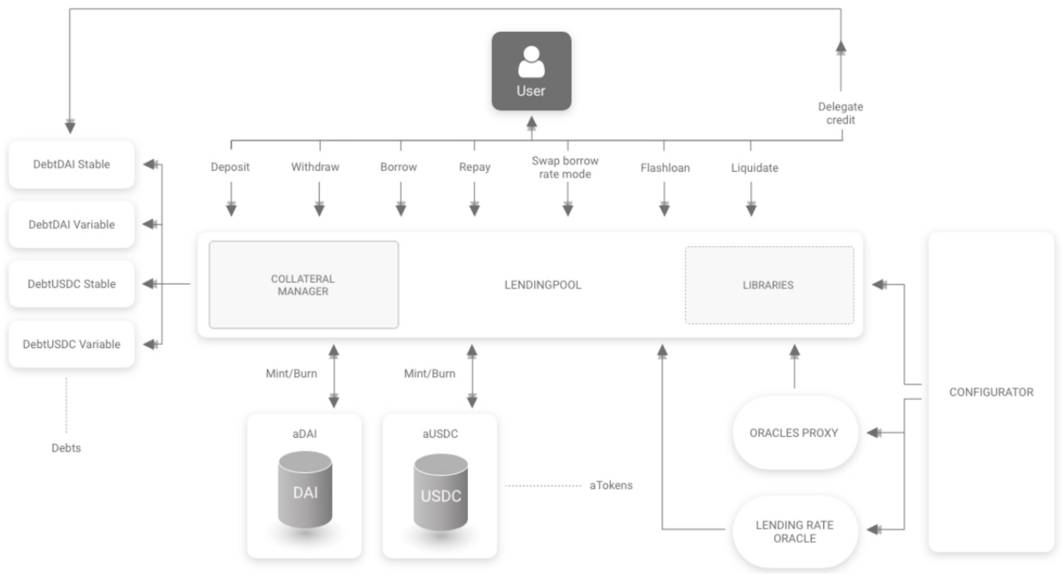


Figure 3.3: Aave V2 Protocol Architecture[27]

3.2.4 Aave Governance

Aave[25] uses a decentralized autonomous organization or DAO[41] to govern the protocol. The General Governance Process[53] at the time of writing is executed as follows:

1. **Governance Forum[54]:** The proposals are introduced. Anyone can create an Aave Request for Comments (ARC)[76] and submit it to the forum.
2. **Snapshot[74]:** The ACR is submitted to vote.
3. **AIP:** If the proposal passes the snapshot, it moves to the Aave Improvement Proposal (AIP), which is reviewed by developers, and security contributors, before any on-chain submission.
4. **Create Proposal:** When the AIP has been revised, it can be created on-chain; a user with enough proposition power can call the function `create()` [87] from the aave governance contract.
5. **Voting and Execution:** This will result in the proposal being succeeded or failed. If succeeded, the proposal can be executed or expired.

Chapter 4

Application Requirements

This chapter describes the application requirements based on ISO/IEC/IEEE 29148:2018[59] Software Requirements Specification standard.

Project: Thesis.

4.1 Introduction

4.1.1 Purpose

The main purpose of this chapter is to provide a detailed description of the decentralized application developed for this thesis.

4.1.2 Scope

Requirements for a decentralized application that can provide permissionless financial services using an evm based blockchain.

The minimum viable functionality that the application shall offer the users are:

- Sign in with the metamask wallet.
- See which tokens can be deposited.
- Using Aave: Deposit ETH, DAI to earn interest.
- Using Aave: Withdrawal of deposited assets.
- Using Aave: Borrow DAI.
- Using Aave: Repay the borrowed DAI.

The application runs online; its frontend may not be decentralized, but its backend runs on top of an EVM-based blockchain such as Ethereum.

4.1.3 Product Perspective

Interface

The application runs in the latest version of Chrome, Firefox, Brave browsers on Windows, Linux and Mac.

4.2 Requirements

4.2.1 Functional

Login

[Thesis-SRS-01] The application shall allow users to login using the Metamask 5.2.1 wallet.

[Thesis-SRS-02] If the wallet is set with another network, the application should inform the user to change to the application network.

[Thesis-SRS-03] The application shall display the user balance of the supported assets.

[Thesis-SRS-04] The application shall display the user balance of deposit assets and earnings.

[Thesis-SRS-05] The application shall display the interest rates for the supported assets.

Deposit

[Thesis-SRS-06] The application shall allow users to deposit ETH to earn interest.

[Thesis-SRS-07] The application shall allow users to deposit DAI to earn interest.

[Thesis-SRS-08] The application shall convert ETH to WETH, because the protocol only accepts ETH in the form of ERC20, i.e. WETH.

[Thesis-SRS-09] The application shall display the user’s balance after any deposit transaction.

Withdrawal

[Thesis-SRS-10] The application shall allow users to withdraw deposited cryptoassets.

[Thesis-SRS-11] The application shall display the user′s balance after any withdrawal transaction.

Borrow

[Thesis-SRS-12] The application shall set the maximum borrowable DAI at 95% to avoid liquidations and display it.

[Thesis-SRS-13] The application shall let users borrow DAI based on users′ deposited collateral.

[Thesis-SRS-14]The application shall display the user′s balance after any borrow transaction.

Repay

[Thesis-SRS-15] The application shall let users repay their partial or total borrowed DAI

[Thesis-SRS-16] The application shall display the user′s balance after any repay transaction.

4.2.2 Nonfunctional

[Thesis-SRS-17] The application should display a notification after any deposit transaction.

[Thesis-SRS-18] The application should display a notification after any withdrawal transaction.

[Thesis-SRS-19] The application should display a notification after any borrow transaction.

[Thesis-SRS-20] The application should display a notification after any repay transaction.

Performance

[Thesis-SRS-21] The application will be deployed in the Ethereum Rinkeby test network.

4.2.3 Use Cases

The following use case diagram 4.1 describes the minimum functionality that the application shall have. The thesis-dapp is the subject (system) that enables EOA accounts 2.2.1 or users to access decentralized financial services. The user (primary actor) uses the thesis-dapp to perform the following use cases:

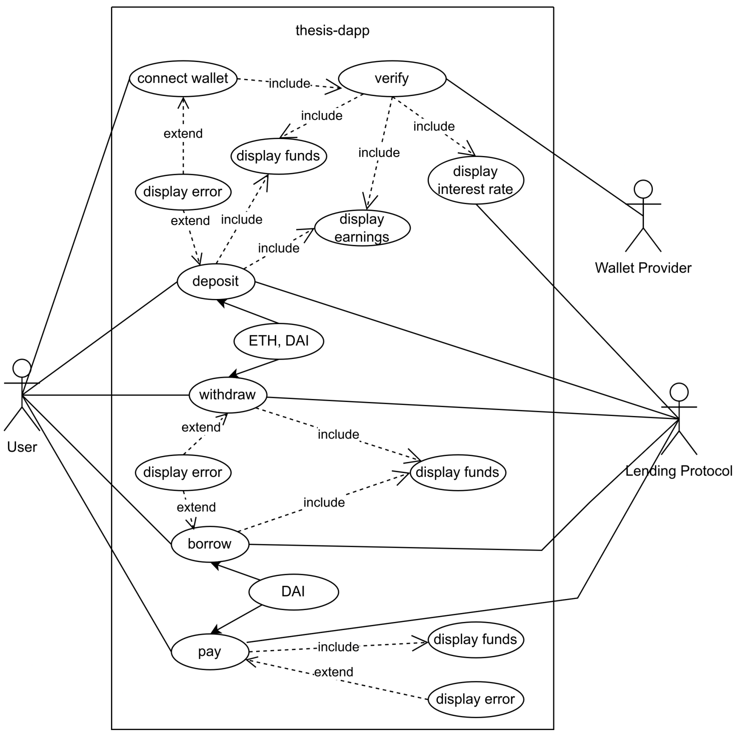


Figure 4.1: Use Case Diagram: Minimum Viable Product.

- **Connect wallet** use case describes the authentification of the user. Connect wallet involves the wallet provider (secondary actor), and it's necessary to perform the other use cases (deposit, withdraw, borrow, pay). The wallet allows users to sign transactions, i.e., enabling users to submit the mentioned use cases.
- **Deposit** use cases describe the cases in which the user performs a deposit transaction to the Lending Protocol (secondary actor).

- **Withdraw** use case describes cases when a user, who already has deposited into the lending pool, needs to withdraw their deposited ETH or DAI.
- **Borrow** use case describes the scenario when a user, who has deposited collateral, aims to borrow an amount of DAI.
- **Pay** use case describes the situation when a user wants to pay its borrowed DAI position.

Chapter 5

Implementation

5.1 Interaction with the Smart Contracts

The main purpose of the application is to be able to access the different functionalities, such as deposit and borrow, of `pool.sol` smart contract within the Aave Protocol 3.2. In order to test the interaction process with smart contracts, the Brownie framework[36] is been used. Brownie is a python-based framework for the development and testing of EVM smart contracts. Brownie enables developers the compilation, test, and deployment of smart contracts with just a few commands, it also uses the ganache-cli[84], which is a built-in local blockchain (test RPC blockchain node) hosted on 127.0.0.1:8545 for fast contract tests. Brownie also lets developers manage accounts and network configurations in an easy and safe way. A new brownie project for this thesis was created using the `brownie init` command.

Under its permissionless characteristic, anyone can interact with a smart contract. However, from a developer's perspective, to be able to interact with a smart contract, the following three items are needed:

1. An Account: Needed to sign the transactions. Brownie provides developers with a set of test accounts. However, for this project, a new EOA Account 2.2.1 was created using Metamask 5.2.1, and imported to the brownie project using its private key 2.1.4.

2. The Application Binary Interface (ABI) 2.2.3, which is generated in the compilation process, `brownie compile` command.

3. The Smart Contract Address: This is the address of the contract account 2.2.1.

This thesis uses the smart contracts from the V3 release of the Aave Protocol. All the contracts can be found in the Aave GitHub repository[30]. Since all the contracts are also deployed in different EVM blockchains, the contracts can also be found in their respective blockchains explorers. For instance, the `pool.sol` deployment in the Arbitrum Network can be found here[33].

The following figure 5.1 gives an overview of how the account and the smart contracts (used for this thesis) interact with each other.

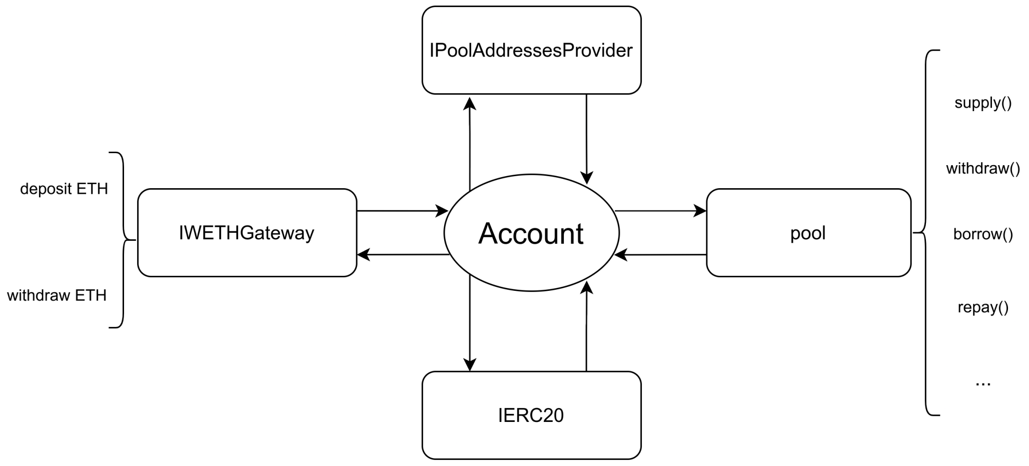


Figure 5.1: Smart contract interactions for the project of this thesis.

5.1.1 The Pool Contract

The `pool.sol`[67] contract enables users and other contracts to interact with the main functionality of the Aave Protocol, more specifically, the `IPool.sol`, which is the interface of the `pool.sol`. The `IPool.sol` is placed on top of the pool.sol (inherit) and contains the functions declarations that the `pool.sol` shall implement. Functions in the `IPool.sol` are declared using the keyword `external`, enabling other contracts the interaction with these functions. This interface construction facilitates the interaction between dapps or contracts. The functions of the `pool.sol` contract used for this thesis are:

- **Supply:** The `supply()` function, also known as the deposit function in Aavve V1 and V2, enables accounts to deposit a given amount of a supported crypto asset into its reserve 3.2. If the supply transaction succeeds, the account address, set in the parameter `onBehalOf`, will receive the equivalent amount of aTokens. For instance, if an account supplies 100 DAI, 100 aDAI will be sent back to the account address set in the `onBehalOf` parameter. Once the supply transaction is done, interest 3.2.1 is generated according to the status of the reserve of the deposited crypto asset.

- **Withdraw:** The `withdraw()` function enables accounts to get the deposited asset (underlying asset) from the reserve by destroying or burning the generated aToken for the deposited asset. Only accounts holding the aToken will be allowed to withdraw the underlying asset from the reserve. The caller of the withdraw function can decide the receiver of the underlying asset by passing the desired account address to the parameter `to`.
- **Borrow:** This function allows accounts to borrow a maximum amount of a given asset depending on the borrowing power of the account. The borrowing power of the account depends on how much collateral the account has deposited into the collateral reserve. Borrowers can decide the interest rate by setting `theinterestRateMode` parameter to stable or variable 3.2.1. The account address set in the `onBehalfOf` parameter will receive the specific amount of debt tokens.
- **Repay:** The `repay()` function allows accounts to pay back borrowed assets by burning the specific debt tokens, reducing or removing the debt tokens of the account address passed in the `onBehalfOf` parameter.
- **Reserve Data:** The `getReserveData()` function returns the detailed reserve state for a given underlying asset. The asset address is passed as a parameter of the function. Using this function, the liquidity rate R_t, stable borrow rate R_s 3.2.1, and other data of the reserve can be retrieved.
- **User Account Data:** The `getUserAccountData()` function returns important data of the reserves for the passed account address. For instance, the health factor H_f 3.2.1, and other relevant data can be retrieved using this function.

5.1.2 The Address Provider Contract

As mentioned before 5.1, in order to interact with smart contracts, the address of the contract account is needed. One of the characteristics of a smart contract is its immutability; once a contract is deployed to the blockchain network, it can not be changed. However, to be able to implement new features approved by the Aave Governance 3.2.4, or adjust the contract to the deployed market (blockchain network), the pool contract implements the Proxy Upgrade Pattern[69], based on *delegatecall*[58], where the code is executed in the context of the caller account address. For this reason, to obtain the address of the pool contract, or IPool contract, the `IPoolAddressesProvider.sol`[68] contract is needed. This contract provides the function `getPool()`, which returns the address of the pool proxy contract.

5.1.3 The IERC20 Contract

Since ERC20 follows a standard 2.2.4, the `IERC20.sol` allow accounts to interact with any deployed ERC20 token. The project of this thesis needs to interact with ERC20 tokens, for instance, getting the balance of a given token or approving tokens to be deposited into the pool contract. Such actions can be accomplished by calling the `balanceOf()` and `approve()` functions of the IERC20 contract.

5.1.4 The IWETH Contract

Because the pool contract can only handle ERC20 tokens, to be able to use ETH within the protocol, it needs to be converted to an ERC20 token. The `IWETH.sol`[90] allow accounts to wrap ETH into an ERC20 token. Accounts that call the `deposit()` method of the IWETH contract will receive the exact amount of the deposited ETH in the form of WETH, which is an ER20 token that represents EHT. Accounts that desire to deposit ETH to the pool contract need to convert their ETH to WETH first. Another option, used in this project, is to use the `IWETHGateway.sol`[91] contract provided by Aave. This contract takes care of wrapping and unwrapping the deposited ETH.

5.1.5 Node Provider

As noted earlier 2.2.5, to communicate with the blockchain, more precisely to a node of the network, where the EVM and smart contracts live, a node or node provider service is needed. For this project, the Alchemy[34] and Infura[47] nodes were used. Both Services offer developers an endpoint (an API KEY) to communicate with the desired blockchain network, allowing developers to interact with the network without the need to run a node.

5.1.6 Web3.py and Brownie

To send transactions to the blockchain (interact with the smart contracts), a connection to the node provider is needed. This connection can be done using well-known web3 libraries such as `web3.py`[89]. For instance, using `web3.py` the connection to the node provider can be done using the following code:

```
w3 = Web3(Web3.HTTPProvider(
    'https://rinkeby.infura.io/v3/$WEB3_INFURA_PROJECT_ID'))
```

The `WEB3_INFURA_PROJECT_ID` is the API KEY from the node provider, and the `w3` instance enables communication with the blockchain.

The brownie framework[36] uses the `web3.py` library and enables developers to easily configure the targeted blockchain node. The configuration options are placed in the `.env` and `.yaml` files of the project. Once the configuration is done, python scripts on the targeted network can be performed, passing the network name on the `network` flag. For instance, running a python script that targets the rinkeby test network, can be done as follows:

```
brownie run scripts/my_script.py --network rinkeby
```

As an example, using brownie and the Alchemy node provider, a python script that performs the deposit action, for a given amount of DAI, can be described as follows:

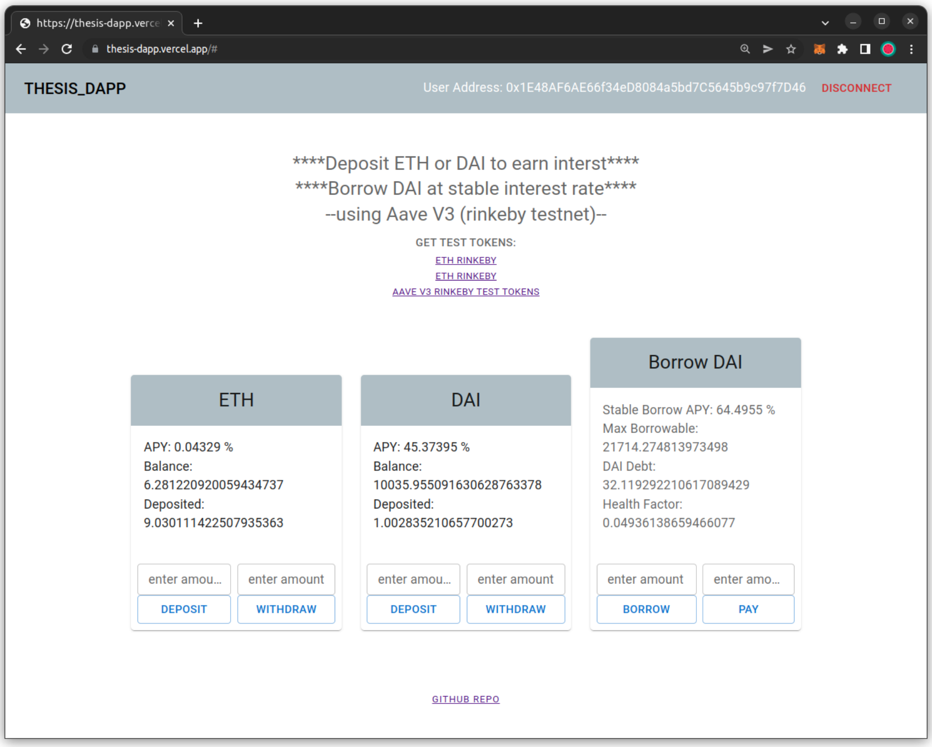
1.- Compile all the contracts to generate the ABIs.
2.- Get the IPool address using the IPool address provider contract.
3.- Using the IPool ABI and the IPool address, get the IPool contract object.
4.- Using the IERC20 ABI and the IERC20 address, get the IERC20 object.
5.- Get the DAI balance of the account using the IERC20 contract object.
6.- If the desired amount to deposit is less than the account DAI balance, approve the desired amount using the IERC20 contract object, setting the `spender` parameter to the address of the pool.
7.- Using IPool contract object, call the `supply()` function with the desired amount of DAI.

The first interaction with the smart contracts of Aave, for the application of this thesis, was done using the brownie framework. However, in order to let other accounts (users) interact with these smart contracts, a frontend with a wallet 2.2.5 is needed.

5.2 Interaction with the users

To enable any EOA account 2.2.1 (users) to interact with the smart contracts, a frontend and a wallet 2.2.5 are needed. The frontend or user interface enables users to enter the specific amount to deposit, withdraw or borrow, and to perform these operations using buttons that will perform the interaction with the smart contracts. The frontend shall also display the interest rates of the lending pool. For the frontend implementation of the thesis application the web development framework Next.js[86] and the Material UI[61] is used. On the other hand, the wallet allows users to sign transactions and see their token balances. The thesis dapp uses the Metamask 5.2.1 wallet provider.

Fig. 5.2 shows the user interface when a user is connected. In the top right, the user's account address is displayed. The three boxes with their input controls allow users



THESIS_DAPP

User Address: 0x1E48AF6AE66f34eD8084a5bd7C5645b9c97f7D46

****Deposit ETH or DAI to earn interst****
****Borrow DAI at stable interest rate****
--using Aave V3 (rinkeby testnet)--

GET TEST TOKENS:
ETH RINKEBY
ETH RINKEBY
AAVE V3 RINKEBY TEST TOKENS

ETH

APY: 0.04329 %
Balance:
6.281220920059434737
Deposited:
9.030111422507935363

DAI

APY: 45.37395 %
Balance:
10035.955091630628763378
Deposited:
1.002835210657700273

Borrow DAI

Stable Borrow APY: 64.4955 %
Max Borrowable:
21714.274813973498
DAI Debt:
32.119292210617089429
Health Factor:
0.04936138659466077

Figure 5.2: Thesis dapp user interface when a user is logged in.

to perform the deposit, withdraw, borrow and pay actions to the given asset. The boxes also display the user account info 5.2.3: balances, APYs, debt, borrow power, and Hf 3.2.1.

5.2.1 Wallet Provider: Metamask

The metamask[100][62] wallet allows users to create and manage their accounts safely. This wallet uses the Infura node provider to communicate to the blockchain network and also allows the connection to any blockchain network that implements the JSON-RPC 2.2.5 specification. For developers, metamask provides the `ethereum` API to target web3 users using the metamask wallet. For instance, detecting if a user has metamask installed in its browser, can be done as follows:

```
if (typeof window.ethereum !== 'undefined') {...}
```

Metamask allows users and developers to communicate with the blockchain by facilitating: a node provider (Provider) that enables communication with the blockchain node, and the management of the private key to sign transactions (Signer).

5.2.2 Web3-react

Web3-react[64] is a library aiming to help developers to keep the users (account) relevant data up-to-date using React Context[70] to store the data and inject it where needed. To connect with metamask, one can import the `InjectedConnector` object, configure it to the application needs, and use this object, along with the `activate`, and `deactivate` modules from the `useWeb3React()` component, to connect with metamask. Once the user is connected, the interaction with the wallet and the node provider can be done using the `useWeb3React()` component. This component can be destructured into the following modules:
`{ activate, deactivate, active, chainId, account, provider }`

- `activate`, `deactivate` allows to connect and disconnect from the wallet.

- `active` is a boolean value that indicates whether the user is connected or not.

- `chainId` is a number that represents the network where the user is connected.

- `account` is the address of the connected account.

- `provider` is used to send transactions, it also contains the signer, which is needed to sign transactions. For instance, actions that only read from the blockchain will only require the provider, and transactions that imply a state change will require to sign the transaction: `provider.getSigner()`

Since this application is using metamask, the Infura provider will be used to interact with the blockchain network.

5.2.3 Web3 Library: ethers

Given that communication with the frontend is required, and that brownie because of its python nature, is not capable of this, the ethers library is used to communicate with the users and the node provider. Using ethers, communication between users, and smart contracts can be achieved by taking care of the following abstractions:

- **Contract**: Enables the interaction with the specific smart contract. As noted above 5.1, to generate the contract object, the contract address and ABI is needed, and to send a transaction using the contract object, a provider or signer is required, through web3-react 5.2.2, the provider and the signer from metamask can be accessed. To sum up, creating a contract instance that enables the interaction with the smart contract on the blockchain can be achieved as follows:
  `new ethers.Contract( address , abi , signerOrProvider )`

- **Provider**: Enables read-only interactions with the blockchain. For instance, getting the balance of a given asset.
- **Signer**: Enables users to sign transactions, hence changing the state of the blockchain. For instance, deposit a given asset to the Aaave lending pool.

The application can be implemented by using the concepts previously outlined, which enables users accounts to interact with the Aave protocol. The interaction is done through the User Interface components, which trigger the methods that interact with the smart contracts, see 5.2. Once the user account is connected to the

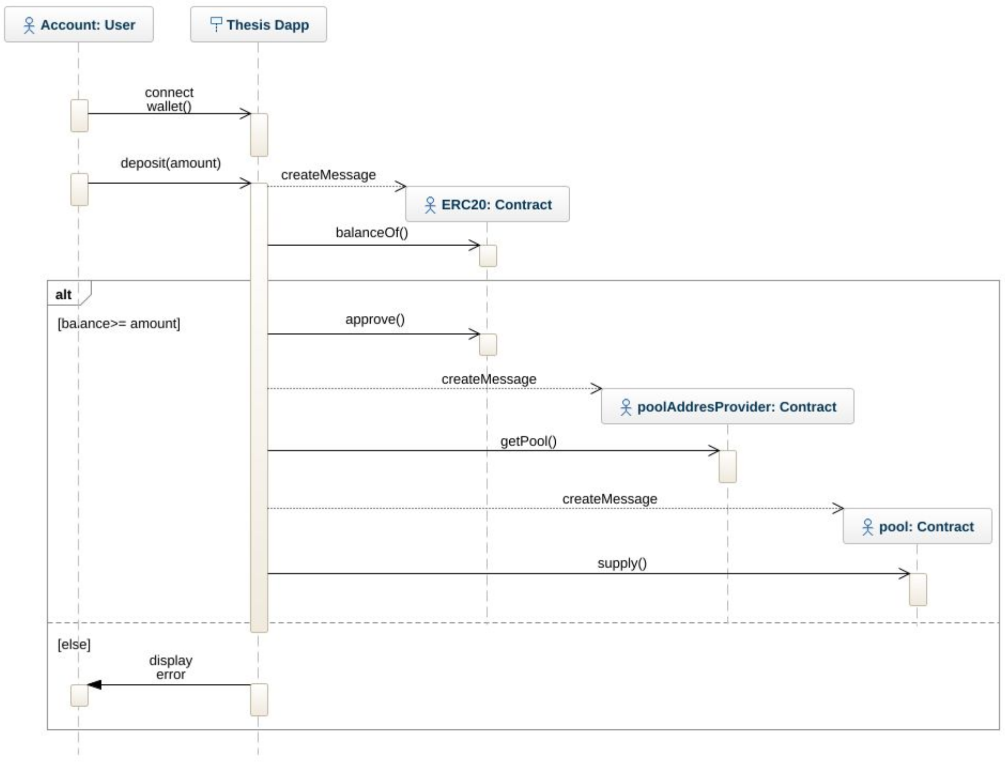


Figure 5.3: Deposit Scenario

Rinkeby testnet network (which is done using the activate and deactivate modules of web3-react), a user can deposit, withdraw, and borrow DAI if the user account has enough collateral. For instance, the interaction between the user interface, the application methods, and the Smart contracts for depositing DAI can be seen in the 5.3 sequence diagram (see 5.1 for implementation code).

The deposit button will trigger the deposit method, which creates the ERC20 contract and the pool contract instances; gives DAI allowance to the pool contract, waits for the user to sign the transaction, and the transaction to be confirmed (by the consensus protocol 2.1.3). When the approve transaction is processed and confirmed, the deposit transaction can be started. Furthermore, if the supply or deposit transaction is processed and confirmed, the account address will receive the exact amount of the deposited asset, in the form of aTokens 3.2.2 (aDAI).

Listing 5.1: Deposit Procedure

```
  ...
// create pool & erc20 contract instances:
const pool = await getPoolContractWrite(provider);
const erc20Contract = await getERC20ContractWrite(assetAddr,
    provider)
// approve the amount of tokens to be deposited into the pool:
const tx_approve = await erc20Contract.approve(pool.address,
    amount)
await tx_approve.wait();
// deposit tokens into the pool
const tx_deposit = await pool.supply(asset_addr, amount, account,
    referralCode)
await tx_deposit.wait();
  ...
```

Note that the amount is passed to the contract instances in the form of Wei 2.2.2 using ethers 5.2.3. Ultimately, the withdraw, borrow, and repay functionalities follow the same pattern. For instance, the Pay Scenario 5.4.

Deposit APY, Stable Borrow APY, and other User Account Info

As shown in Figure 5.2, the application also displays the deposit and borrow Annual Percentage Yield (APY). Following the Aave documentation[31], the APY (compounded per second) for the deposited and borrowed assets is calculated as follows:

$$APY = (1 + (APR/secondsPerYear))^{secondsPerYear} - 1$$

Where the APR is the *currentLiquidityRate* or *currentStableBorrowRate* obtained from the `getReserveData()` 5.1.1 function and divided by 10^{27} (Ray Math 3.2.1). For instance, getting the deposit APY is done this way:

```
depositAPY = ((1 + (depositAPR / SECONDS_PER_YEAR)) **
    SECONDS_PER_YEAR) - 1
```

User's ERC20 token balances are fetched using the ERC20 contract instance and the corresponding token address (DAI, aDAI, aWeth, aDebtToken). The ETH balance

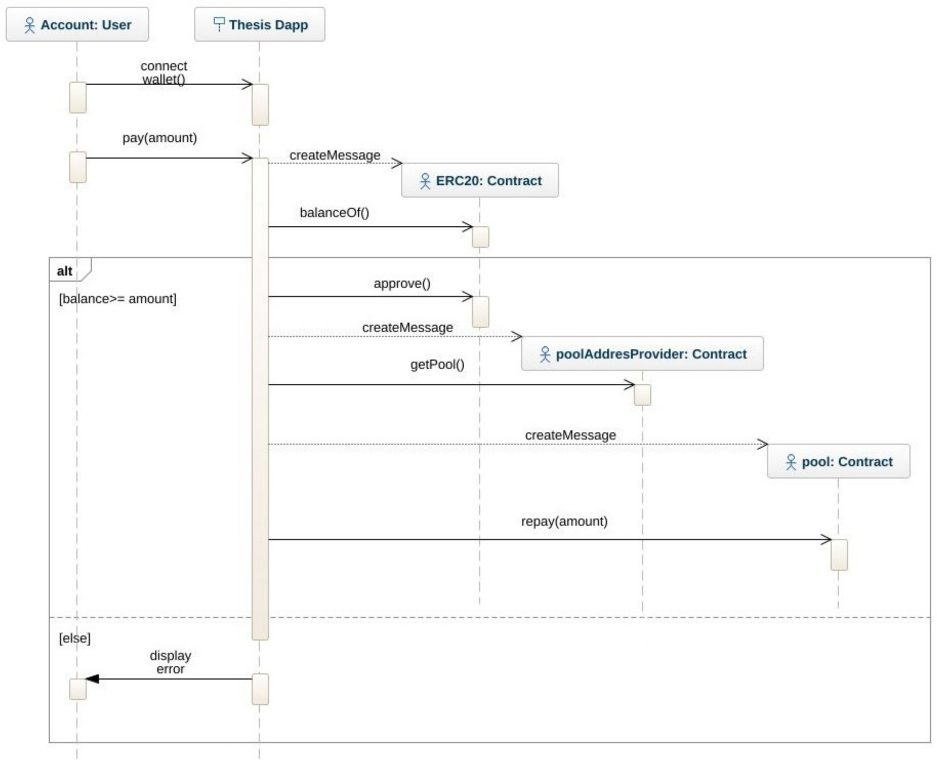


Figure 5.4: Pay Scenario

can be retrieved by using the provider and account modules 5.2.2: `provider.getBalance(account)`. The health factor and the borrowing power are both retrieved from the `getUserAccountData()` 5.1.1 function. And borrowing power (max borrowable) is set to 95% to avoid liquidations 3.2.1. More implementation details and the code for the thesis dapp can be found in its GitHub repository[81].

Chapter 6

Conclusion

The aim of this chapter is to provide a comprehensive overview of the conclusions obtained during the development of this thesis.

6.0.1 Achieved Goals

The main goals of this thesis were to study the technology behind decentralized applications and implement a decentralized application that provides financial services such as lending and borrowing crypto assets.

To reach these goals, chapter 2 describes the set of technologies that compose a blockchain network, as well as the Ethereum network. It also outlines why this blockchain platform is so important in the space of decentralized applications. Finally, it presents the architecture of dapps. The requirements in chapter 4 were brought up for developing the thesis dapp. After struggling with the decision of the protocols for the dapp, the Aave protocol was chosen. The main reasons to use Aave for the thesis app development were its maturity in the Defi space and the well-produced developer documentation. Once studying the Aave protocol and its core functionalities in chapter 3, along with the different web3 tools, the implementation was possible, as disclosed in chapter 5.

Challenges

While developing the thesis, a few challenges had to be resolved. Since blockchain technology and its whole ecosystem are still in their early stages, there were not that many resources to work with. For instance, during the development process of the thesis dapp, I had to rely only on the official documentation. There was no extensive literature on best practices or common errors to avoid, such as there is for other technologies. However, the community was welcoming and I was quickly able to find help in forums such as the Ethereum Stack Exchange[21] or even on Discord[43] servers. I found the appropriate answers to my questions during the development process on the Aave developer's Discord server.

6.0.2 Future work

The thesis dapp is a DeFi dapp (running on the rinkeby testnet) that enables users to deposit ETH and DAI and earn interest on it. It also enables users to borrow DAI at a stable interest rate. The dapp relies on the Aave protocol to provide such financial services. However, it is just a small part of the DeFi products available today. The future work that can be implemented is:

- Improve the User Interface.
- Continue to study the Aave protocol and add new features such as flash loans[50].
- Thanks to the composability nature of dapps, the thesis dapp could add more protocols to interact with. For instance, creating a yield strategy to get the best APY combining different protocols such as Compound[78] or Curve finance[39].
- Rigorous unit and integration test and add support to an EVM mainnet.

Thanks to decentralized networks, such as Ethereum 2.2, and the different DeFi protocols, such as Maker 3.1 or Aave 3.2, anyone can access more stable currencies and other financial services that may not be possible in their traditional financial infrastructures. This could allow people in regions that may suffer from inflation to have an option to protect their assets using these emerging technologies. Moreover, any developer can use this set of technologies to build a new dapp, like the dapp developed for this thesis, that enables users to access these protocols.

Bibliography

[1] Andreas M. Antonopoulos. *Mastering Bitcoin*. O'Reilly Media, Inc., first edition, December 2014.

[2] Andreas M. Antonopoulos. *Mastering Bitcoin*, pages 137–139. O'Reilly Media, Inc., first edition, December 2014.

[3] Andreas M. Antonopoulos and Dr. Gavin Wood. *Mastering Ethereum*, pages 1–6. O'Reilly Media, Inc., November 2018.

[4] Andreas M. Antonopoulos and Dr. Gavin Wood. *Mastering Ethereum*, pages 297–314. O'Reilly Media, Inc., November 2018.

[5] Andreas M. Antonopoulos and Dr. Gavin Wood. *Mastering Ethereum*, pages 314–317. O'Reilly Media, Inc., November 2018.

[6] Andreas M. Antonopoulos and Dr. Gavin Wood. *Mastering Ethereum*, pages 127–134. O'Reilly Media, Inc., November 2018.

[7] M. Poongodi Bharat S. Rawal, Gunasekaran Manogaran. *Implementing and Leveraging Blockchain Programming*, pages 12–17. Springer Singapore, January 2022.

[8] M. Poongodi Bharat S. Rawal, Gunasekaran Manogaran. *Implementing and Leveraging Blockchain Programming*, page 107. Springer Singapore, January 2022.

[9] M. Poongodi Bharat S. Rawal, Gunasekaran Manogaran. *Implementing and Leveraging Blockchain Programming*, pages 26–27. Springer Singapore, January 2022.

[10] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. 2014. available at https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[11] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, pages 99–111, 1991. available at https://link.springer.com/article/10.1007/BF00196791.

[12] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, pages 321–325. CRC Press., 1997.

[13] Ralph C. Merkle. Protocols for public key cryptosystems. pages 122–122, 1980.

[14] Pratyusa Mukherjee and Chittaranjan Pradhan. *Blockchain 1.0 to Blockchain 4.0—The Evolutionary Transformation of Blockchain Technology*, pages 29–49. Springer International Publishing, Cham, 2021.

[15] Kaihua Qin, Liyi Zhou, Yaroslav Afonin, Ludovico Lazzaretti, and Arthur Gervais. Cefi vs. defi – comparing centralized to decentralized finance, 2021.

[16] Fabian Schär. Decentralized finance: On blockchain- and smart contract-based financial markets. Federal Reserve Bank of St. Louis Review, Second Quarter 2021, pp. 153-74.

[17] Rajeev Sobti and Geetha Ganesan. Cryptographic hash functions: A review. *International Journal of Computer Science Issues, ISSN (Online): 1694-0814*, Vol 9:461 – 479, 03 2012.

[18] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Berlin Version 934279c.

[19] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Berlin Version 934279c, Apendix G. Fee Schedule.

[20] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. 06 2017.

Online Sources

[21] Ethereum Stack Exchange. https://ethereum.stackexchange.com/. [Accessed June 13, 2022.].

[22] aave v1 whitepaper. https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf. [Accessed May 23, 2022.].

[23] aave v1 whitepaper. https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf. [page 3, Accessed May 25, 2022.].

[24] aave v1 whitepaper. https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf. [page 18, Accessed May 25, 2022.].

[25] aave v1 whitepaper. https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf. [page 7, Accessed May 27, 2022.].

[26] aave v1 whitepaper. https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf. [page 16, Accessed May 27, 2022.].

[27] aave v2 whitepaper. https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf. [Accessed May 23, 2022.].

[28] aave v2 whitepaper. https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf. [pages 3-5, Accessed May 23, 2022.].

[29] aave v3 technical paper. https://github.com/aave/aave-v3-core/blob/master/techpaper/Aave_V3_Technical_Paper.pdf. [Accessed May 23, 2022.].

[30] aave/aave-v3-core: This repository contains the core smart contracts of the Aave V3 protocol. https://github.com/aave/aave-v3-core. [Accessed May 28, 2022.].

[31] APY and APR - Developers. https://docs.aave.com/developers/v/2.0/guides/apy-and-apr#apr-greater-than-apy. [Accessed June 7, 2022].

[32] Arbitrum. https://arbitrum.io/. [Accessed May 23, 2022.].

[33] Arbitrum (ETH) Blockchain Explorer. https://arbiscan.io/address/0x794a61358D6845594F94dc1DB02A252b5b4814aD#code. [Accessed May 28, 2022.].

[34] Blockchain APIs and Node Infrastructure. https://www.alchemy.com/. [Accessed May 30, 2022.].

[35] Bring the World to Ethereum | Polygon - Polygon. https://polygon.technology. [Accessed May 23, 2022.].

[36] Brownie. https://eth-brownie.readthedocs.io/en/stable/toctree.html. [Accessed May 28, 2022.].

[37] Byzantine fault. https://en.wikipedia.org/wiki/Byzantine_fault. [Accessed June 13, 2022.].

[38] Contract ABI Specification. https://docs.soliditylang.org/en/latest/abi-spec.html. [Accessed 18 Apr. 2022.].

[39] Curve Finance: Stats, Charts and Guides: DeFi Pulse. https://www.defipulse.com/projects/curve-finance. [Accessed June 13, 2022.].

[40] Dapps Platform. https://www.avax.network/. [Accessed May 23, 2022.].

[41] Decentralized autonomous organization. https://en.wikipedia.org/wiki/Decentralized_autonomous_organization. [Accessed May 23, 2022.].

[42] DeFi Pulse - The Decentralized Finance Leaderboard: Stats, Charts and Guides: DeFi Pulse. https://www.defipulse.com/. [Accessed May 18, 2022.].

[43] Discord, Your Place to Talk and Hang Out. https://discord.com/. [Accessed June 13, 2022.].

[44] Discrete logarithm. https://en.wikipedia.org/wiki/Discrete_logarithm. [Accessed June 13, 2022.].

[45] Double-spending. https://en.wikipedia.org/wiki/Double-spending. [Accessed June 13, 2022.].

[46] DS-Math. https://dappsys.readthedocs.io/en/latest/ds_math.html. [Accessed May 26, 2022.].

[47] Ethereum API: IPFS API Gateway: ETH Nodes as a Service. https://infura.io/. [Accessed May 30, 2022.].

[48] Ethereum virtual machine opcodes. https://ethervm.io/. [Accessed 18 Apr. 2022.].

[49] Fantom. https://www.fantom.foundation/. [Accessed May 23, 2022.].

[50] Flash Loans - FAQ. https://docs.aave.com/faq/flash-loans. [Accessed June 13, 2022.].

[51] Gas tracker. https://etherscan.io/gastracker. [Accessed 17 Apr. 2022.].

[52] Gavin Wood. https://en.wikipedia.org/wiki/Gavin_Wood. [Accessed 18 Apr. 2022.].

[53] General Governance Process. https://docs.aave.com/developers/guides/governance-guide. [Accessed May 27, 2022.].

[54] Governance Forum. https://governance.aave.com/. [Accessed May 27, 2022.].

[55] Important considerations - Developers. https://docs.aave.com/developers/v/1.0/developing-on-aave/important-considerations. [Accessed May 26, 2022.].

[56] Introduction to dapps. https://ethereum.org/en/developers/docs/dapps/. [Accessed 5 Apr. 2022.].

[57] Introduction to smart contracs. https://ethereum.org/en/developers/docs/smart-contracts/. [Accessed 18 Apr. 2022.].

[58] Introduction to Smart Contracts - Solidity 0.8.14 documentation. https://docs.soliditylang.org/en/v0.8.14/introduction-to-smart-contracts.html?highlight=delegatecall#delegatecall-callcode-and-libraries. [Accessed May 30, 2022.].

[59] ISO. https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:29148:ed-2:v1:en. [Accessed 21 Apr. 2022.].

[60] Liquidations - FAQ. https://docs.aave.com/faq/liquidations. [Accessed May 27, 2022.].

[61] Material UI. https://mui.com/material-ui/getting-started/overview/. [Accessed May June 1, 2022].

[62] MetaMask Docs. https://docs.metamask.io/guide/#why-metamask. [Accessed June 1, 2022.].

[63] A next generation, statically typed, future-proof smart contract language for the ethereum virtual machine. https://fe-lang.org/. [Accessed 18 Apr. 2022.].

[64] NoahZinsmeister/web3-react at v6. https://github.com/NoahZinsmeister/web3-react/tree/v6. [Accessed June 1, 2022].

[65] Oasis.app. https://oasis.app/borrow. [Accessed May 23, 2022.].

[66] Oracles. https://ethereum.org/en/developers/docs/oracles/. [Accessed 18 Apr. 2022.].

[67] Pool - Developers. https://docs.aave.com/developers/core-contracts/pool. [Accessed May 30, 2022.].

[68] PoolAddressesProvider. https://docs.aave.com/developers/core-contracts/pooladdressesprovider. [Accessed May 30, 2022.].

[69] Proxy Upgrade Pattern. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies. [Accessed May 30, 2022.].

[70] React - Context. https://reactjs.org/docs/context.html. [Accessed June 1, 2022].

[71] Risk Parameters - Risk. https://docs.aave.com/risk/asset-risk/risk-parameters. [Accessed May 27, 2022.].

[72] RPC 2.0 Specification. https://www.jsonrpc.org/specification#conventions. [Accessed May 16, 2022.].

[73] SHA-256. https://en.bitcoin.it/wiki/SHA-256. [Accessed June 21, 2022.].

[74] Snapshot. https://snapshot.org/#/aave.eth. [Accessed May 27, 2022.].

[75] Solidity. https://docs.soliditylang.org/en/v0.8.13/index.html. [Accessed 18 Apr. 2022.].

[76] Submitting an ARC. https://docs.aave.com/governance/arcs. [Accessed May 27, 2022.].

[77] The Architecture of a Web 3.0 application. https://www.preethikasireddy.com/post/the-architecture-of-a-web-3-0-application. [Accessed May 16, 2022.].

[78] The Compound protocol. https://compound.finance/markets. [Accessed May 21, 2022.].

[79] The Maker Protocol White Paper: Feb 2020. https://makerdao.com/en/whitepaper/. [Accessed May 23, 2022.].

[80] The Maker Protocol White Paper: Feb 2020. https://makerdao.com/en/whitepaper#the-dai-stablecoin. [Accessed May 23, 2022.].

[81] Thesis Dapp GitHub Repository. https://github.com/cito-lito/thesis_dapp. [Accessed June 7, 2022].

[82] Tokens. https://docs.openzeppelin.com/contracts/4.x/tokens. [Accessed 18 Apr. 2022.].

[83] Top Stablecoin Tokens by Market Capitalization. https://coinmarketcap.com/view/stablecoin/. [Accessed May 24, 2022.].

[84] trufflesuite/ganache: A tool for creating a local blockchain for fast Ethereum development. https://github.com/trufflesuite/ganache. [Accessed May 28, 2022.].

[85] Uniswap Info. https://info.uniswap.org/#/. [Accessed May 19, 2022.].

[86] vercel/next.js: The React Framework. https://github.com/vercel/next.js. [Accessed May 31, 2022.].

[87] Voting Governance - Developers. https://docs.aave.com/developers/v/2.0/protocol-governance/governance#create. [Accessed May 27, 2022.].

[88] Vyper. https://vyper.readthedocs.io/en/latest/. [Accessed 18 Apr. 2022.].

[89] Web3.py 5.28.0 documentation. https://web3py.readthedocs.io/en/stable/. [Accessed May 16, 2022.].

[90] wETH. https://weth.io/. [Accessed June 13, 2022.].

[91] WETHGateway. https://docs.aave.com/developers/periphery-contracts/wethgateway. [Accessed May 30, 2022.].

[92] Vitalik Buterin Fabian Vogelsteller. EIP-20: Token standard. https://eips.ethereum.org/EIPS/eip-20, 2015. [Accessed 18 Apr. 2022.].

[93] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf. [Accessed 5 Apr. 2022.].

[94] Takenobu T. Ethereum evm illustrated. https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf. [accessed 15-April-2022].

[95] Wikipedia. Elliptic-curve cryptography — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Elliptic-curve%20cryptography&oldid=1081874678, 2022. [accessed 13-April-2022].

[96] Wikipedia. Finance — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Finance&oldid=1088464612, 2022. [accessed 18-May-2022].

[97] Wikipedia. Financial institution — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Financial%20institution&oldid=1074029593, 2022. [accessed 18-May-2022].

[98] Wikipedia. Financial instrument — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Financial%20instrument&oldid=1071762966, 2022. [accessed 18-May-2022].

[99] Wikipedia. Financial market — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Financial%20market&oldid=1083352146, 2022. [accessed 18-May-2022].

[100] Wikipedia. MetaMask — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=MetaMask&oldid=1083935548, 2022. [accessed 22-April-2022].

[101] Wikipedia. Peer-to-peer — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Peer-to-peer&oldid=1084324791, 2022. [accessed 16-May-2022].

[102] Wikipedia. Token — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Token&oldid=1066372445, 2022. [Accessed 18 Apr. 2022.].

[103] Dieter Shirley William Entriken (@fulldecent). EIP-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721, 2018. [Accessed 18 Apr. 2022.].