

Design & Implementation of A Decentralized Peer-to-Peer Application on Ethereum

Mohammed Kamal Ahmed Gumma
Abdulrahman Ali Mohammed Daffallah

A thesis presented for the degree of
bachelor of engineering



Department
University of Khartoum
Sudan
October 2020

Abstract

The fully decentralized Ethereum Cryptocurrency and Blockchain were introduced in July 2015. In contrast to other Blockchains like Bitcoin, it not only supports the transfer of funds among participants but was designed to store and execute code in a distributed manner. This thesis is about the design and implementation of a Decentralized web-based application that uses the concept of smart contracts in Ethereum to facilitate lending and debt management directly between two parties on a peer-to-peer basis without relying on any Trusted Third Party (TTP) for the agreement or payment process. The smart contracts are developed in the Solidity Programming language. They use deposits to increase the compliance illustrated in the lending process between the parties, and all payments are conducted in the Ethereum native cryptocurrency ether. The contracts are deployed on a remote node that hosts an Ethereum client, which is connected over its HTTP-RPC interface with the web application on the user's device. The web client uses the JavaScript Web3js integration library to communicate with the smart contracts on the blockchain. A user-friendly interface is built to allow users to interact with the Decentralized Application's smart contracts, enabling them to lend each other money and create a debt contract that facilitates this process. The application was evaluated in terms of deployment- and transaction costs, scalability, security, and privacy. Although it still relies on a remote Ethereum client and does not provide perfect legal security, the application allows users to flexibly specify purchase and rent contracts, exchange personal user data and execute the monetary transactions directly on the blockchain.

Dedicatons

this is dedicated to those ...

Declarations

I declare this

Acknowledgments

I want to thank ...

Contents

1	Introduction	9
1.1	Motivation	9
1.2	Description of Work	10
1.3	Thesis Outline	11
2	Theoretical Background	12
2.1	Peer-to-Peer Communication	12
2.2	Evolution of the Internet	13
2.3	The Blockchain	13
2.3.1	Merkle Trees	14
2.3.2	Addresses	15
2.3.3	Cryptocurrencies	16
2.3.4	Transactions	16
2.4	Mining and Consensus Algorithms	17
2.4.1	Proof of Work	17
2.4.2	Proof of Stake	18
2.4.3	Consensus Attacks	18
2.5	Ethereum	19
2.5.1	Mining in Ethereum	19
2.5.2	Ethereum Accounts	20
2.5.3	Transactions and Messages	20
2.5.4	Smart Contracts	21
2.5.5	Decentralized Applications	22
2.5.6	Decentralized Applications Development Tools	23
3	Design & Implementation	25
3.1	Interfacing with Ethereum Networks	25
3.2	Ethereum Application Infrastructure	26
3.3	Smart Contract Design	27
3.3.1	Smart Contract Source Code	28
3.3.2	Compiling Solidity	30
3.3.3	Deployment to Rinkeby	31
3.3.4	Automated Testing	33
3.4	Building a Front-end Interface	36
3.4.1	Lending Interface	36
3.4.2	Debt Record Interface	37

4	Evaluation	41
4.1	Costs	41
4.1.1	Deployment Costs	41
4.1.2	Transaction Costs	42
4.2	Application Scalability	42
4.3	Security	42
4.4	Privacy	43
5	Summary & Conclusion	44
5.1	Summary	44
5.2	Conclusion	45
5.3	Limitations & Future Work	45
A	Installations Guideline	46
A.1	Installing and Using MetaMask	46
A.1.1	Installing MetaMask	46
A.1.2	Using MetaMask	46
A.2	Cloning the Project's GitHub Repository	48
A.2.1	Running Scripts	48
B	Front-End Interface Source Code	49
B.1	Lending Interface Source Code	49
B.2	Debt Record Interface Source Code	51

List of Figures

2.1	Server-based Computer Architecture vs. Peer-to-Peer Architecture . .	12
2.2	Blockchain Data-structure	14
2.3	The structure of a Merkle Tree	14
2.4	Validation of a transaction in a Merkle Tree	15
2.5	Outputs of transaction A and B as inputs of transaction C	17
2.6	Execution of a smart contract on the blockchain	22
2.7	Centralized Apps vs. Decentralized Apps	22
3.1	Ethereum Network Interfaces	25
3.2	Traditional App Architecture	26
3.3	Traditional App Architecture	27
3.4	Relationship between the Debt Factory and Debt Smart Contract . .	27
3.5	Peer-to-Peer Lending and Debt Settlement Process	28
3.6	Compiling Smart Contracts	30
3.7	Deployment to Rinkeby	32
3.8	The Debt contract and transaction receipt on Etherscan after scan- ning their QR Codes	39

Chapter 1

Introduction

1.1 Motivation

Almost all our modern-day applications are centralized. They're centralized in the sense that they all rely on a Trusted Third-Party (TTP), the platform owner, to operate the platform. This has many disadvantages for users. They need to sign up for each application separately and have to give away their private data to the application owner. This raises privacy concerns. Another problem is that on most applications, the possible payment methods are limited by the application owner. Users are forced to use a specific payment application and sometimes have to pay high transaction fees. A TTP also always is a single point of failure for all participants of the application. It can be attacked by a distributed denial of service (DDOS) attacks, or a central authority can shut it down. So although the Sharing Economy is often also referred to as the Peer-to-Peer (P2P) economy, its infrastructure is often not (yet) fully decentralized as in real P2P computer networks. Due to these issues, a new technology emerged for building internet-based apps called smart-contract applications or decentralized applications (DApps). The concept of smart contracts offers a solution to overcome the requirement of a TTP in a contractual agreement between two or more parties. A smart contract is a digital protocol that facilitates the agreement process between different parties by enforcing certain predefined rules. Nick Szabo first introduced the concept of a smart contract in 1997. In his article, he proposes to embed contractual clauses like property rights directly into hard- and software to make it expensive for a party to breach the protocol. He outlines a hypothetical digital security system for cars, where a cryptographic key represents the car's ownership and is transferred if the terms implemented in the protocol are fulfilled. In his example, the car owner can withdraw the key from a leaser who does not pay the monthly rent. At that time, smart contracts were not practically achievable, mainly because there existed no digital infrastructure that allowed the secure execution of the protocols without the need for a TTP at some point. This changed with the introduction of the cryptocurrency Bitcoin. Bitcoin is a digital payment system proposed by the pseudonym Satoshi Nakamoto in the year 2008. It allows the transfer of funds between participants without relying on a third party, like a bank. The ownership of currency tokens is maintained with the help of public-key cryptography, meaning that a token belongs to the user that holds its associated private key. To maintain the network's integrity, Bitcoin introduced the blockchain (section 2.1), a distributed ledger that stores all transactions

that were ever committed in a chain of distinct blocks. Participants have to solve a cryptographic puzzle that is used to verify a block. The main incentive for participants to participate in this process is a reward distributed when a block is proved to be valid. In this manner, new Bitcoin tokens are created. Although Bitcoin's functionality can be extended beyond the mere transferring of value tokens by writing special protocols, the scripting language used is not powerful enough to express complex contract logic like in the example envisioned by Nick Szabo. Ethereum, a cryptocurrency that was described first in late 2013 by Vitalik Buterin and went live on 30 July 2015, is the first cryptocurrency with full support for smart contracts. It stores the code and related data of a contract on the blockchain and executes code when participants or other contracts issue transactions. The code of a contract is stored in a byte code format and executed concurrently by all participants on the Ethereum Virtual Machine (EVM). Ethereum supports several scripting languages to write smart contracts and compiles them into byte code format. The most popular among them is Solidity, a contract oriented high-level language whose syntax is similar to JavaScript. Various types of applications can be implemented in Solidity, like Voting, Crowdfunding, Blind Auctions, Multi-Signature Wallets, and more.

1.2 Description of Work

Transactions play a vital role in any financial system. Major challenges for building payment based services are security and trust. Users must feel their funds are secure, and the administrator of the payment service doesn't steal their funds. In addition, unlike fiat currencies, cryptocurrencies have made it possible for anyone to own crypto assets and be able to perform secure financial transactions - bridging the geographical location and class boundaries. The decentralized lending and debt management we'll build will tackle both these issues and will allow users to lend each other crypto-assets and manage debt between them. This has the ability to condense the lengthy, complicated loan process by eliminating third-party intermediaries, certifications, and inherent delays in processing by creating an ecosystem of an interconnected web of lenders, and prospective buyers could also foster a system of uniformity and fairness in lending standards. This can improve the loan procedure entirely by reducing costs, developing incorruptible records, and promoting rapid settlement for all parties involved. This thesis addresses whether two private parties can lend each other money and manage debt between them without the need of a TTP for the agreement or the payment process. This thesis aims to design and implement a web-based smart contract application that enables the safe conclusion of debt contracts between two parties using the Ethereum blockchain to store the contract details and process the payments. The application allows a user to flexibly lend another party money and create a debt contract that facilitates the lending and debt management process between them. Further, the two parties can exchange their contract by using either their phone camera to scan a QR-code or by sharing the debt contract URI. The application also has to satisfy the following non-functional requirements:

- **Security and privacy:** The identity requirements that are used in a contract can be selected in a flexible manner. The application does not have any security breaches that allow for the loss of money or private data of the users.

- **Expandability:** The implementation of the application is as general as possible to integrate new components when the requirements for a contract change.
- **Reliability and scalability** The application is fully functional and can recover from basic failures (e.g., network errors)
- **Usability:** The application provides a user-friendly interface

1.3 Thesis Outline

Chapter 2: Explains how blockchain technologies and smart contracts work. It explains the cryptocurrencies Bitcoin and Ethereum in more detail and discusses their similarities and differences. It further explains how smart contracts work in Ethereum and how they can be programmed using the Solidity scripting language. It also discusses the vulnerabilities of Solidity and the Ethereum platform in general.

Chapter 3: Contains the design and implementation of the application. The first and second parts discuss how interfacing with Ethereum Networks is established and the Ethereum Application infrastructure. The third part goes into the design of the smart contracts and details of integrating the smart contract on the blockchain with JavaScript. In the last part, the actual Web client implementation that provides the user interface is shown.

Chapter 4: Evaluates the application according to the functional and non-functional requirements stated in the introduction chapter.

Chapter 5: Summarizes the results and also points out the limitations of the application and future work.

Chapter 2

Theoretical Background

2.1 Peer-to-Peer Communication

Peer-to-Peer (P2P) communication refers to the transmission of information between two computer peers across a network. P2P networks are a special kind of distributed systems where a set of individual computers (also called nodes) communicate with each other and share their computational resources (e.g., processing power, storage capacity, data, or network bandwidth) without having any central point of authority or intermediary for coordination. The nodes in the network are equal concerning their rights and roles in the system. Furthermore, all of them are both suppliers and consumers of resources [1].

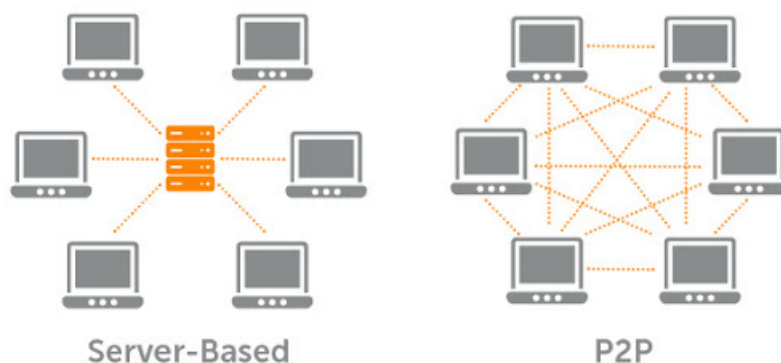


Figure 2.1: Server-based Computer Architecture vs. Peer-to-Peer Architecture

Peer-to-peer systems have interesting applications such as file sharing, content distribution, and privacy protection. Most of these applications utilize a simple but powerful idea: turning the users' computers into nodes that make up the whole distributed system. As a result, the more users or customers use the software; the more powerful the system becomes [1]. Two of the most popular P2P applications are Napster and BitTorrent. Napster is a P2P music sharing software that allows users to connect with other computers, search for music, play, and download any of them. It was initially released in 1999 but has dramatically changed over the years [1]. BitTorrent is a communication protocol used for P2P file sharing. It allows users to share all types of digital files with each other. In order for users to transfer or receive files, they have to use computer programs that implement the BitTorrent

protocol on their internet-connected devices. These programs are called clients, and one of the most popular BitTorrent clients is uTorrent [1].

2.2 Evolution of the Internet

The Internet and the way we interact with it over the decades have evolved. When the internet was initially made available across the globe, it was a dump of information in a very static way that you could only access the information with some images and text, with no personalized information or user-centered design or personal preferences. The era of the web we are using in today's world is known as web 2.0. it provides a very personalized approach to most of the websites today where users can change and filter things according to their preferences, along with the systems automatically detecting and devise the content and interaction sites to the users. The application of such implementation can be seen in social media sites. If a user searches for one particular type of content, the suggested or sponsored content may appear in their feed with the aid of Machine Learning algorithms allowing the internet to be more interactive engaging. It is also worth remembering that the internet is again moving towards Web 3.0, which will automate many aspects of today's systems and provide a whole new experience that will transform the user experience [2].

Web 3.0 or the Decentralized Web refers to the next generation of the internet powered by Blockchain Technology, Artificial Intelligence, and Big Data Analytics. Distributed Ledger Technologies like Blockchain are set to disrupt the internet through ledgers, decentralized protocols, and cryptography. A ledger is a list of transactions. A database is different from a ledger. In a ledger, we can only append new transactions, whereas, in a database, we can append, modify, and delete transactions. This enables us to reconfigure the internet into a distributed global computer where we're no longer dependent on web platforms and data centers of web 2.0 to run the internet but can now build and run applications on this shared global computing infrastructure [2].

2.3 The Blockchain

Blockchain or Distributed Ledger Technology (DLT) was first introduced after the 2008 financial crisis when a person(s) under the name of Satoshi Nakamoto published a white-paper that encompassed the technology and presented what's known now as the Bitcoin cryptocurrency [3]. The term "Blockchain" is used synonymously to refer to the Blockchain as a distributed network of nodes and the data structure that underlies the distributed system. The latter is defined as an append-only data structure consisting of a chain of blocks where every transaction executed in the network is permanently recorded. Every block on the blockchain contains the hash of its predecessor, along with many other attributes. This allows the blockchain to create a chain of blocks of data that reaches the first block created, the genesis block. Subsequent blocks depend on each other, making it computationally infeasible to alter any block on the blockchain without changing all of its previous blocks [3]. This solves the double-spending problem, which will be discussed later in (Section 2.3.3).

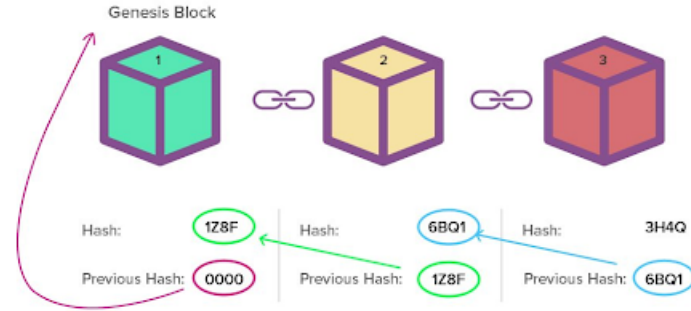


Figure 2.2: Blockchain Data-structure

Bitcoin and Ethereum are the most predominant blockchain networks. Bitcoin is the first implementation of the blockchain introduced by Satoshi Nakamoto, while Ethereum was proposed in late 2013 by Vitalik Buterin and launched in July 2015 [4].

Every peer has a unique address in a blockchain network and can send cryptocurrency in transactions to other peers in the network. A certain amount of cryptocurrency is transferred from peer A's address to peer B's address. Transactions that happen at a particular time frame are grouped in a block that gets appended to the chain of blocks. The blockchain's state, i.e., this distributed data structure as a chain of blocks is maintained and extended by some consensus mechanisms, the mechanism differs from one blockchain to another [5, 6].

The next sections discuss the most important pillar concepts of Blockchain technology in more depth.

2.3.1 Merkle Trees

The Merkle Tree is a data structure that stores a digital signature for the whole list of transactions in a block. It is used to verify the integrity of a transaction efficiently with the help of a Binary Hash Tree [7].

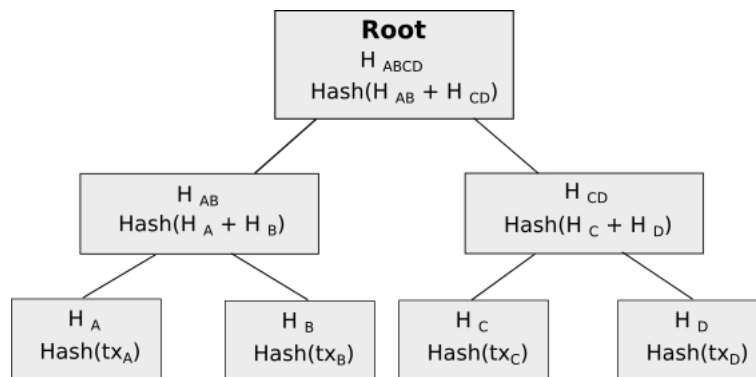


Figure 2.3: The structure of a Merkle Tree

Figure 2.3 illustrates the structure of the Merkle Tree. Its leaf nodes contain the hash value of the individual transactions of a block. Every non-leaf node hash value is the hash of its two children. This tree's resulting root node is then included in the block that contains the associated transactions [7]. The Merkle Tree allows a node only to download the header of a block and a small number of nodes from the Merkle Tree to validate a transaction. In the example illustrated in figure 2.3, only the nodes H_D , H_{AB} , and H_{EFGH} are required to prove the inclusion of the transaction H_C in the block. This verification takes on average $O(\log n)$ time, and at most $O(n)$ time because the structure is the same as in a binary search tree [7]. If an attacker smuggles in an invalid transaction somewhere at the bottom of the tree, the hash of that transaction propagates upward to the root node and makes the hash of the whole block invalid [5].

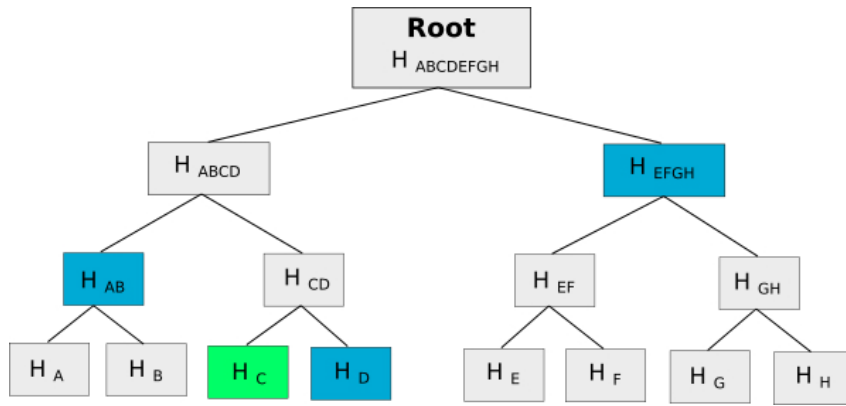


Figure 2.4: Validation of a transaction in a Merkle Tree

The ability to download and validate only a part of a block to validate individual transactions is essential for the network's sustainability. A full node storing and processing every block's transactions consumes approximately 75GB of disk space as of December 2019 and grows by 1GB per month. It is not feasible for a mobile device to store such amounts of data [5]. Simplified Payment Verification (SPV) protocol enables the so-called "light nodes" or SPV nodes. SPV nodes only download a blocks header and the Merkle Tree part relevant to prove the inclusion of a transaction. In a similar way, SPV nodes prove the inclusion of a block in the blockchain by validating the blocks header. The combination of these validation processes allows an SPV node to prove that a transaction belongs to the blockchain by downloading less than 1 kilobyte of data [7].

2.3.2 Addresses

Each participant in the Blockchain network has to create a wallet file that stores the digital private/public keys combination associated with the blockchain address used to signing and validating transactions before receiving or sending transactions between peers in the network [7].

The private key is just a randomly generated number.

$$K_{priv} \in \{0, 1\}^{256} \quad (2.1)$$

A public key is a 512-bit number generated from the private key using the Elliptic curve DSA(ECDSA) algorithm [7].

$$K_{pub} = ECDSA_{512}(K_{prev}) \quad (2.2)$$

When a participant in the network wants to send a transaction, a signature with their private key is created, and at the same time, their public key is with the network so that it's able to verify the authenticity of the transaction that's being sent [7].

The public key of the participant's address is shared only when it signs a transaction. To receive crypto assets, the public key's hash function is used as the recipient's address. This 160-bit hash is generated using the $RIPMED_{160}$ and SHA_{256} hash functions:

$$K_{address} = RIPMED_{160}(SHA_{256}(K_{pub})) \quad (2.3)$$

2.3.3 Cryptocurrencies

At the core of every blockchain lies a cryptocurrency. Cryptocurrencies govern the interactions between nodes and the network itself using completely decentralized protocols to achieve consensus among nodes in the blockchain compared to digital currencies that rely on a Trusted Third Party (TTP) for maintaining the state of the network [5]. As of January 2020, Bitcoin (BTC) and Ethereum (ETH) are the most popular cryptocurrencies with a market capitalization of over 131 and 20 Billion US-Dollar, respectively [8].

2.3.4 Transactions

In distributed networks, transactions are data-structures that are used to transfer assets between peers. Every transaction has several inputs and outputs. The inputs reference past transaction outputs called Unspent Transaction Outputs (UTXO), they're chunks of crypto that belong to a specific participant, and they're permanently recorded on the blockchain. In addition, UTXO that belongs to a specific owner is distributed across multiple transactions and blocks depending on the network's scale. Thus, the account's actual balance, including all the crypto-assets owned by the participant, must be calculated by the wallet service used and not stored in the network [7]. Since a participant's wallet balance consists of a number of UTXO with different values scattered across the network, it's often impossible to send a specific amount of crypto-assets to a recipient address. Instead, multiple UTXOs are spent as an input for the transaction. The output would be a new UTXO with the spent amount of UTXO and another UTXO with the participant's remaining credit balance. To demonstrate this, see figure 2.4.

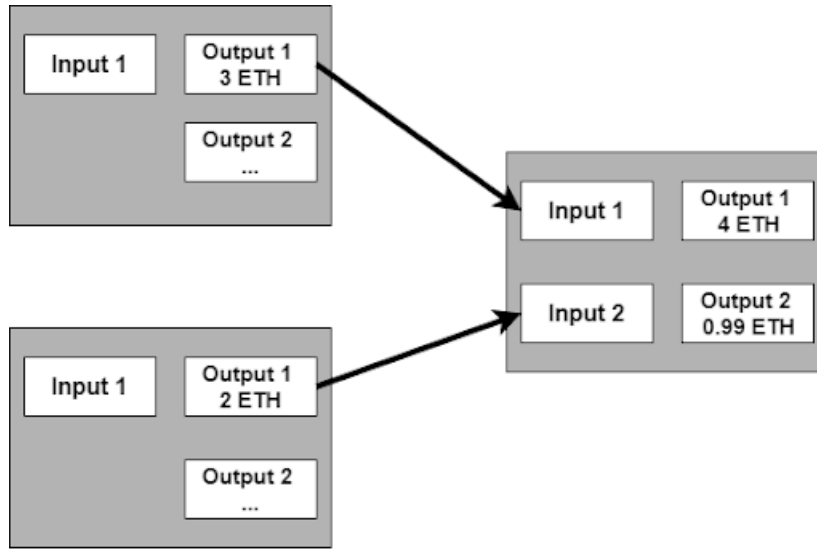


Figure 2.5: Outputs of transaction A and B as inputs of transaction C

In this case, the participant wants to send 4 Ether to a recipient. Two previous UTXOs from transaction A are referenced and taken as input to transaction B. The first UTXO has 2 Ether in value, and the other has 3 Ether in value, accumulating to 5 Ether. Transaction B is sent, and the output is two UTXOs; one is for the amount of Ether sent to the recipient, and the other is the remaining change directed back to the participant as one of their up-to-date UTXOs. The difference of the input and output values is a transaction fee collected by the miners, which includes this transaction in their blocks. Transaction fees give an incentive for miners to prioritize transactions over others or to include them at all. Transaction fees also prevent abusive transactions from destabilizing the network [7].

2.4 Mining and Consensus Algorithms

The creation of new cryptocurrency, along with the maintainability and extension of the Blockchain network is carried out by participants or nodes called Miners. Miners constantly try to solve a cryptographic puzzle called “Proof of Work” (PoW) to validate new blocks and reach an agreement or consensus about the blockchain state. When peers broadcast transactions to the network, miners include them in the blocks they generate. Miners that successfully validate new blocks are awarded the newly created cryptocurrency and transaction fees [6, 9].

2.4.1 Proof of Work

As mentioned before, miners in the network try to validate their blocks by solving the cryptographic puzzle to be awarded in cryptocurrency. In PoW, miners try to generate a random integer (nonce) until the hash of this integer, together with the block header, yields a smaller value than a predefined target by the puzzle. The target value is adjusted dynamically and depends on the total hashing power of the network. It is adjusted such that a block is created every 10 minutes on average. The mining process makes it hard for a single node to manipulate the network’s

state unless it controls the majority of the mining power [5, 10].

The first node that manages to compute the nonce is rewarded with the transaction fees and the coinbase reward. This coinbase reward was 50 bitcoins at the genesis node, but every 210,000 blocks, the reward is halved. The coinbase reward is the only source of new cryptocurrency, and thus, the supply of bitcoins is limited to 21 million [7]. In Bitcoin's case, the last bitcoin will be mined is estimated to be mined in 2140, when the block reward would drop below one satoshi (the smallest denomination of BTC) [11].

When a mining node finds a nonce, it adds it to the block header and broadcasts the block to all network nodes. Since the broadcasting process takes some time, two or more miners may find a solution for the same block and propagate it in the network. This creates a fork in the blockchain, and for a short time, multiple chains can coexist. When this happens, new blocks are added to the longest chain by honest miners [3]. Both Bitcoin and Ethereum 1.0 use this consensus method. This method's disadvantage is that it requires an enormous amount of energy because of the computationally intensive puzzle. A single transaction consumes 652.61 Kwh, which is equivalent to an average U.S. household's power consumption over 22.06 days [12].

2.4.2 Proof of Stake

In the Proof of Stake (PoS) consensus algorithm, the next block's creator is awarded based on various random combinations based on wealth or age (i.e., stake). In contrast to PoW, there's no competition as the next block's creator is chosen based on their stake, and there's no block reward, the block creator only takes a transaction fee. Selection by stake would result in undesirable centralization, as the single wealthiest participant would have a permanent advantage over other peers in the network [13]. To bridge this, Randomized Block and Coin age-based Selection are introduced to add an element of randomness to the process of selection. In randomized block selection, the generator uses a formula that looks for the lowest hash value in combination with the size of the stake while in coin age-based selection, a number derived from the product of the number of coins at stake multiplied by the number of days the coins have been held is added to the generator formula. The advantage of this method is that it produces coins without consuming significant computational power. Ethereum 2.0 uses PoS, but it's still in beta [14, 15].

2.4.3 Consensus Attacks

- **51% Attack**

As we'll see in section (2.4.1), it's nearly impossible for one participant to alter the blockchain state by changing a block data after a certain time, because that would also require altering all the subsequent blocks. However, it's still possible for a miner with a high computational power to insert invalid transactions to blocks in the near future or deny specific participants of the networks if they own 51% or more of the overall computational power of the network [7]. In this attack, the attacker tries to insert transactions containing UTXO of their address multiple times. In this scenario, the attacker sends a UTXO in a transaction and waits until the transaction is included in a block. The

attacker then forks the blockchain below this block by mining a new block in which they spend the same UTXO again in another transaction to an address that is owned by her. They then try to create more blocks than the rest of the miners to create a longer chain than the chain containing the original transaction. If they manage to do that, the actual transaction is considered invalid because the miners will always work on the longest chain. To protect against such an attack, the recipient of a transaction should wait for at least six confirmations until they accept the payment. The longer they wait, the more unlikely it is that a 51 percent attack succeeds. This is known as the double-spending problem [5, 7].

- **Sybil Attack**

In this attack, an attacker can attempt to fill the network with regular nodes controlled by them; participants would then be very likely to connect only to the attacker nodes. Once they have connected to the attacker nodes, the attacker can refuse to relay blocks and transactions from everyone, thereby disconnecting them from the network. The attacker can relay only blocks that they create, thereby putting victim participants on a separate network, and so on [7].

2.5 Ethereum

Ethereum is a decentralized computing platform that allows us to deploy Decentralized Applications (DApps) by using blockchain to store not only the state of user accounts but also program code and its associated state. This program code is called Smart Contracts and will be discussed further in section (2.4.5). Ethereum was specially designed to allow anyone to write smart contracts and decentralized applications. It supports several scripting languages that can be compiled to bytecode executed on the Ethereum Virtual Machine (EVM). Ethereum uses the blockchain data structure and proof-of-work consensus protocol. A method of a smart contract can be invoked via a transaction or another method. There are two kinds of nodes in the network: regular nodes and miners. Regular nodes are the ones that just have a copy of the blockchain, whereas miners build the blockchain by mining blocks [5, 16].

The following subsections discuss the most critical aspects of Ethereum.

2.5.1 Mining in Ethereum

The Ethereum blockchain is very similar to Bitcoin. The most important difference is the fact that a block not only contains a list of transactions but also the whole state of the network. The state is stored in a data structure called “Patricia Tree” [5].

The Patricia Tree is a modified Merkle Tree that is optimized for the insertion and deletion of nodes. It stores the state of all contract and externally owned accounts. Every block stores a reference to the root of the tree and updates only the parts that changed because of the transactions’ effects in that block. This allows new nodes to only download the Patricia Tree instead of all blocks to retrieve all accounts’ states.

Therefore, it saves a considerable amount of disk space. It is estimated that if this concept is applied to Bitcoin, it would require a node to store between 5 and 20 times fewer data [5].

Ethereum also uses a different proof-of-work algorithm, called Ethash, which produces a block every 12 seconds on average compared to 10 minutes in Bitcoin. This has the advantage that transactions can be processed faster, and a recipient of a transaction does not have to wait long until she can consider a transaction to be safe. It also increases the interactivity of applications that interact with contracts on the blockchain. Further, Ethash is memory hard and, therefore, ASIC resistant [16].

The fast block time's negative effect is that the stale rate, the rate at which blocks that are not part of the main chain are produced, is increased. This is a security risk that can lead to centralization to mining pools since many miners will not be rewarded for their effort in mining new blocks. While in Bitcoin, such blocks are considered "orphan" and are no longer used, the GHOST protocol of Ethereum also allows for the inclusion of such "uncle" blocks and rewards the miners of them [5]. In contrast to Bitcoin, the mining reward for a block is static and exactly 5.0 ether. Successful miners also collect all gas that is used in the transactions of a block. Miners of "uncle" blocks receive 7/8 of the static block reward [16]. The total amount of ether issued in a year is statically bound to 1/3 of the pre-sale, which is approximately 18 million ether. It is estimated that about 1% of the total monetary base is lost every year due to key owners' death, loss of private keys, or transactions to empty addresses. Therefore the supply of ether grows at a disinflationary rate until the rate of annual loss and destruction of ether will balance the rate of issuance rate, and the currency no longer grows [5].

2.5.2 Ethereum Accounts

In Ethereum, an account is an object that stores a user's account balance or the state of a contract. The former is called an externally owned account (EOA) because it belongs to an external entity and is controlled by a private key. It can send messages by creating and signing transactions with its private key. The latter is called a contract account, and the contract code controls its state. When a contract account receives a message, its code is executed. It can also send messages to other contracts or create new contracts [5, 16].

2.5.3 Transactions and Messages

- **Transactions**

A transaction is a data package that is signed by an externally owned account. It contains the transaction's sender, the recipient of the transaction, the amount of ether sent, an optional data field, a gas limit, and a gas price field [5].

- **Messages**

A message is like a transaction that is sent from one contract to another contract. It behaves similarly to an ordinary function call in other programming languages. Every time the running code of a contract account executes the

“call” opcode, a message is generated and sent to the recipient contract or externally owned account. Both transactions and calls can create new contracts, invoke functions of a contract, or transfer ether to a contract or an externally owned account [5, 16].

- **Ether**

Ether is the currency token used by Ethereum to pay for transaction fees. Developers have to provide ether when they deploy contract code to the blockchain, and users have to spend or burn ether when they invoke transactions on a contract. Ether can be exchanged against other currencies via various cryptocurrency exchanges. Ether can be divided further into smaller units, the smallest among them is called Wei. One ether is equal to 10¹⁸ Wei [5, 16].

- **Gas**

Gas is used to pay transaction fees in Ethereum. Gas is not a currency, but an internal pricing unit that decouples the ether’s market price from the cost of transactions. One unit of gas represents the price for the most simple operation executed on the EVM. The price for one unit of gas, the gas price, can be dynamically adjusted to adapt to the ether currency’s fluctuating values. The originator of a transaction can define the gas price, and miners decide whether they want to include the transactions in their blocks or not [16].

The gas limit or start gas value of a transaction determines how many computational steps a transaction can execute. The more lines of code a contract executes, and the more memory and storage it uses, the higher the initial transaction’s gas limit needs to be [16].

Transaction fees and the gas limit help prevent the execution of faulty code, like infinite loops, and they help to save computational resources on the network. The gas limit also disincentivizes potential denial-of-service attacks on the network [5, 16].

2.5.4 Smart Contracts

Smart contracts are programs that have special accounts that store executable code together with its associated data and an account balance on the blockchain. Smart contracts have an address (a public key) and are created by transactions. Transactions are also used to interact with a contract on the blockchain by sending money to its account balance or executing code. To execute the code of a contract, a function call containing the functions name and its parameters is binary encoded and sent to the contract in the data field of the transaction [16, 17].

Figure 3.5 illustrates the interaction of externally owned accounts (users) with a contract. Every time a contract receives a message from another contract or a transaction from a user, it can receive Ether or execute a specified function in the data field. Similarly, the contract can send money from its balance to other accounts or execute functions on other contracts through the broadcasting of messages [17]. Execution of the code takes place on all mining nodes in the network concurrently, which reach consensus over the new state of the contract using a proof-of-work algorithm. The persistent variables of a contract are stored on the storage, a key-value store associated with the contract that is persisted on the blockchain. Access to the

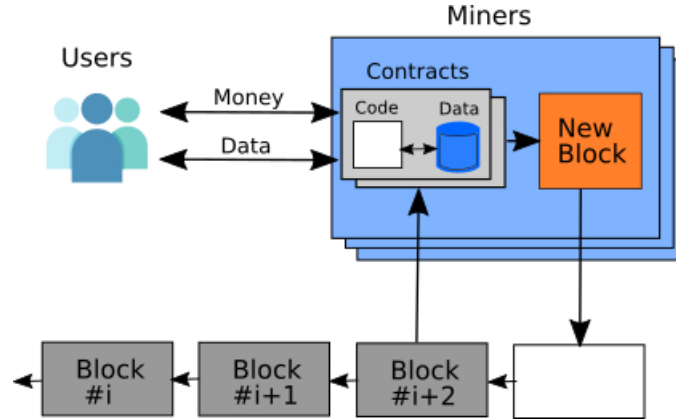


Figure 2.6: Execution of a smart contract on the blockchain

storage is costly (20000 units of gas per 256-bit word) because it has to be stored on every full node in the network. Intermediate results of computations are stored in the memory, a non-persistent byte-array. The state, as well as the code of a contract, are public, and the code of a contract cannot change retrospectively [5, 16].

2.5.5 Decentralized Applications

DApps are created using one or more smart contracts. Smart contracts are programs that run exactly as programmed without any possibility of downtime, censorship, fraud, or third party interface. In Ethereum, smart contracts can be written in several programming languages, including Solidity, LLL, and Serpent. Solidity is the most popular of those languages. Ethereum has an internal currency called ether. To deploy smart contracts or to call their methods we need ether. There can be multiple instances of a smart contract, just like any other DApp, and its unique address identifies each instance. Both user accounts and smart contracts can hold ether.

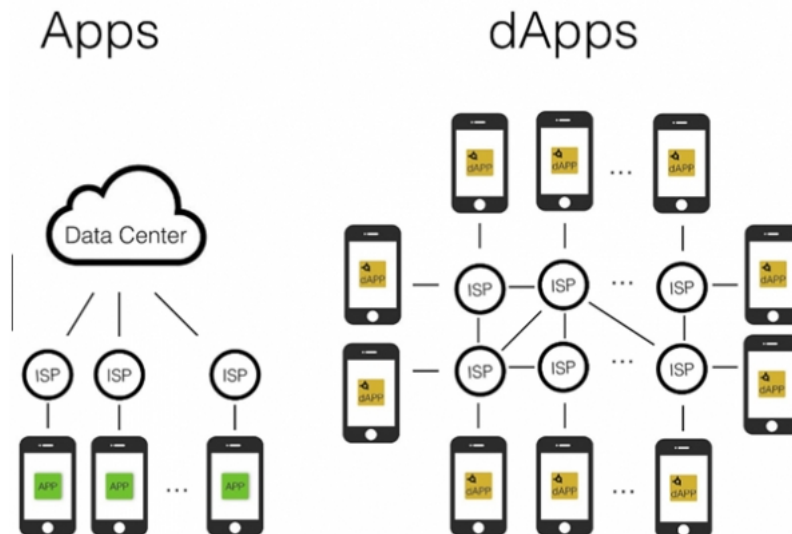


Figure 2.7: Centralized Apps vs. Decentralized Apps

2.5.6 Decentralized Applications Development Tools

Solidity

Solidity is an object-oriented programming language used to write smart contract programs. It's the primary language used in Ethereum DApps and other blockchains that compete with Ethereum [18].

Remix

Remix is an open-source Integrated Development Environment (IDE) tool that allows the deployment, debugging, and testing smart contracts. It's written in JavaScript and supports both usage in-browser and locally [19].

MetaMask

MetaMask is a browser extension wallet that allows users to access and interact with Ethereum powered DApps and Ethereum's accounts management [20].

Web3.js

Web3.js is an Ethereum JavaScript API that consists of a collection of libraries that allow developers to interact with a local or remote Ethereum node over HTTP or IPC connection [21].

Rinkeby

Rinkeby is an Ethereum test network that allows DApp Developers to deploy, test, and interact with DApps before releasing them on the main network [22].

Ganache

Ganache, or formerly known as Test-RPC, is a command-line tool (CLI) that allows developers to interact with the blockchain without the overheads of running an actual Ethereum node [23].

Infura

Infura is a hosted Ethereum node cluster that enables developers to run DApps and access IPFS networks without requiring them to set up their Ethereum node or wallet. It's the Ethereum provider that powers MetaMask [24].

Node.js

Node is an open-source, cross-platform, JavaScript runtime environment for easily building fast and scalable network applications. It also provides a rich library for various JavaScript modules, which simplify the development of web and mobile applications [25].

Mocha

Mocha is a JavaScript automated testing framework running on Node.js. It features browser support, asynchronous testing, test coverage reports, and the use of any assertion library [26].

React.js

React is a JavaScript for building interactive User-Interfaces for Web and Mobile applications. It's maintained mainly by Facebook and has a broad community for support [27].

Semantic-UI

Semantic UI is a modern front-end development framework, powered by LESS and jQuery. It has a sleek, subtle, and flat design look that provides a lightweight user experience [28].

Chapter 3

Design & Implementation

3.1 Interfacing with Ethereum Networks

To interact with Ethereum is to interact with a network of computers. A network is formed by one or more nodes, and these networks are used to transfer money between different parties and store data on the blockchain. There are many different Ethereum networks; there's the main Ethereum network that hosts Decentralized Applications deployed to production, and test networks such as Rinkeby, Ropsten, and Kovan that are being used for testing and developing Decentralized Applications on top of Ethereum.

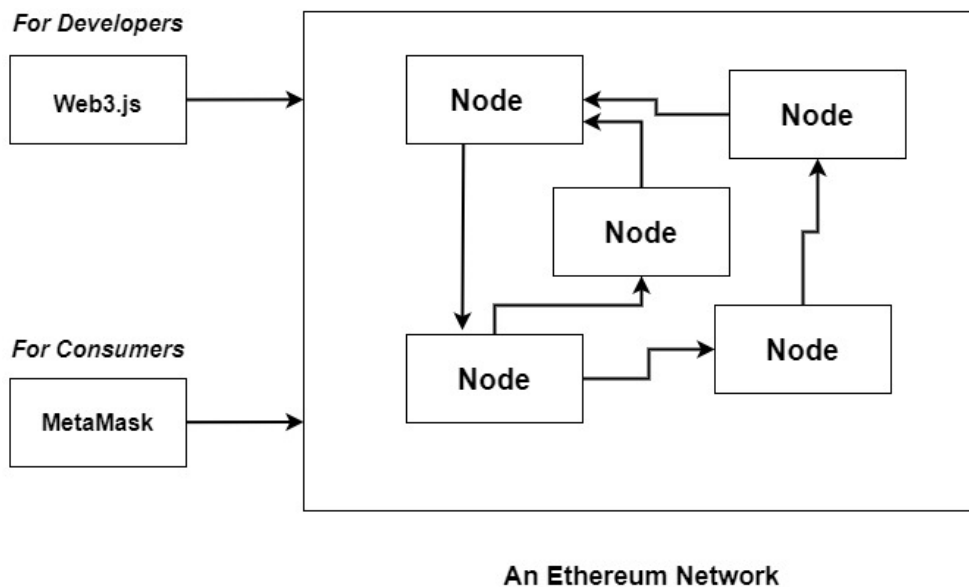


Figure 3.1: Ethereum Network Interfaces

In general, there are two groups of technologies that are going to be made use of for connecting to the network. The first is technologies that are used by Developers to allow them to build decentralized solutions on top of Ethereum. As illustrated in figure 3.1, a library called Web3.js will be made use of - it serves as a portal into the Ethereum network. It allows us to send transactions, store data, besides, deploying and interacting with smart-contracts. As for consumers, there's MetaMask.

MetaMask is a browser extension that allows people to interact with Decentralized Applications.

3.2 Ethereum Application Infrastructure

The vast majority of applications that exist now are traditional in the sense that they're not connected to Ethereum and aren't powered with Web3 functionalities. They consist of a server that's accessed by users, and the server responds with an HTML document and maybe some Javascript assets. Anytime users want to create, update, read, delete data in traditional applications, a request is sent to the server, and that data is modified in the database.

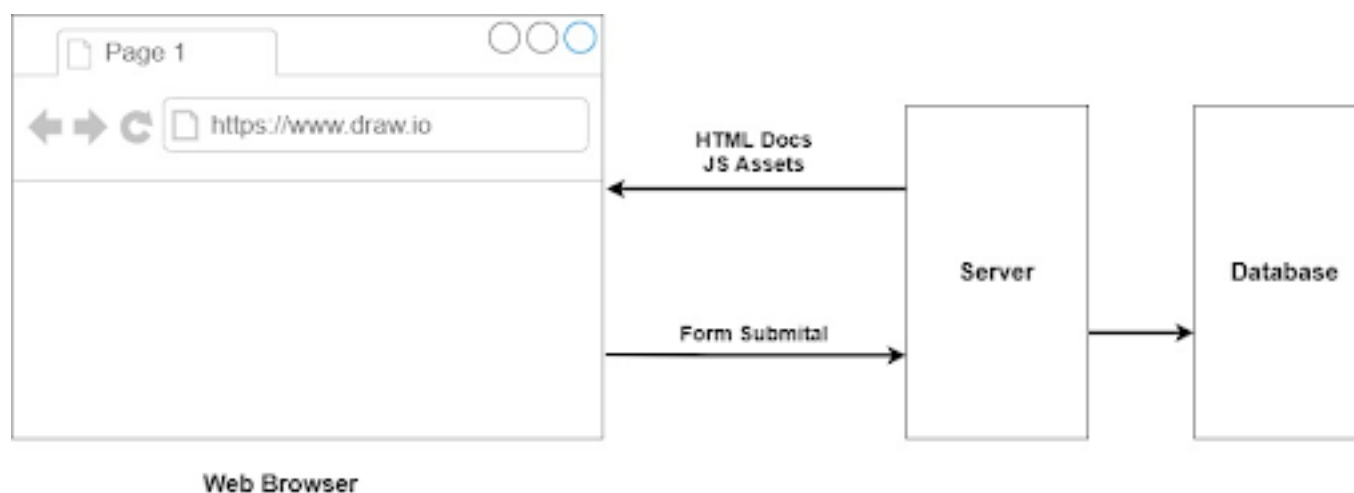


Figure 3.2: Traditional App Architecture

In an Ethereum application architecture, the server's role is dramatically diminished. It does far less work than it did in traditional architecture. It can still send an HTML document and some Javascript assets to the browser. When a user wants to send a transaction or store data in the blockchain, the server isn't involved in this process. However, it can read data from the blockchain and then render it as an HTML document and Javascript assets to the browser. To send transactions and store data in the blockchain, the Ethereum Application will make use of Web3, which communicates with MetaMask, MetaMask creates a transaction and signs it with the user's private key and then sends that transaction to the Ethereum network. The only way for a user to change data is by using their private and public keys that exist on the user's machine, which are isolated from the server and can't be shared under any circumstance.

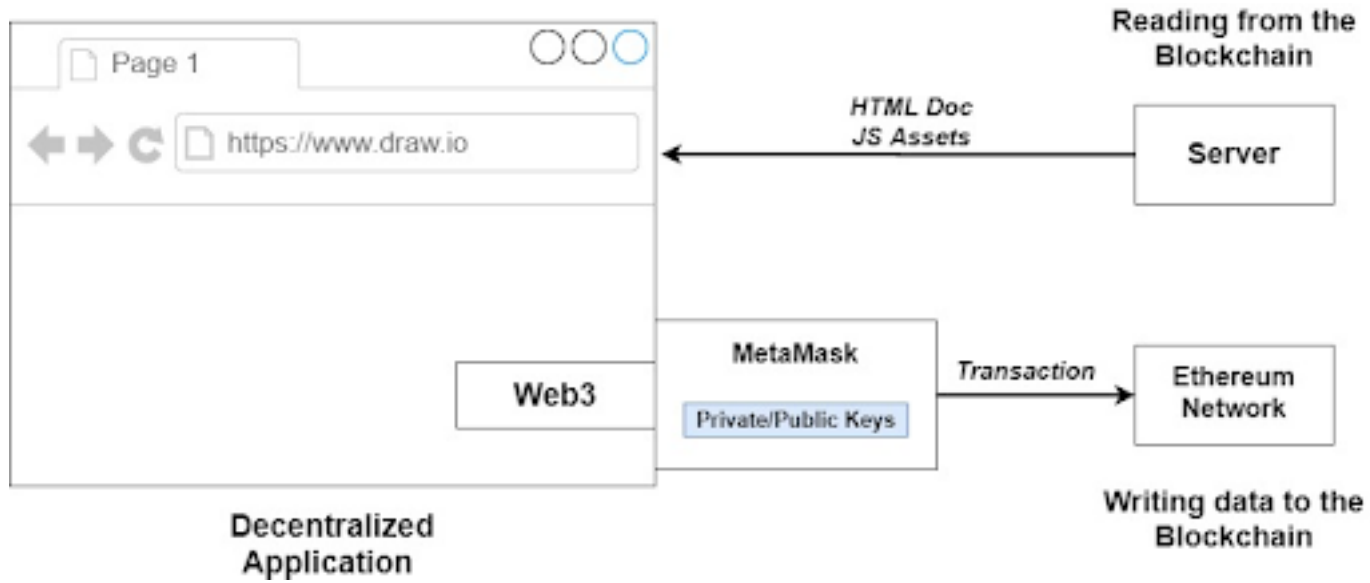


Figure 3.3: Traditional App Architecture

3.3 Smart Contract Design

Two smart contracts will be constructed, the debt factory smart contract that behaves as a parent Smart contract that will allow users to create instances of it. The other smart contract is the Debt smart contract, which behaves like an instance or child of the debt factory smart contract. In object-oriented programming terms, the debt factory is a class, and the Debt smart contract is an object of that class, and users can create as many objects as necessary.

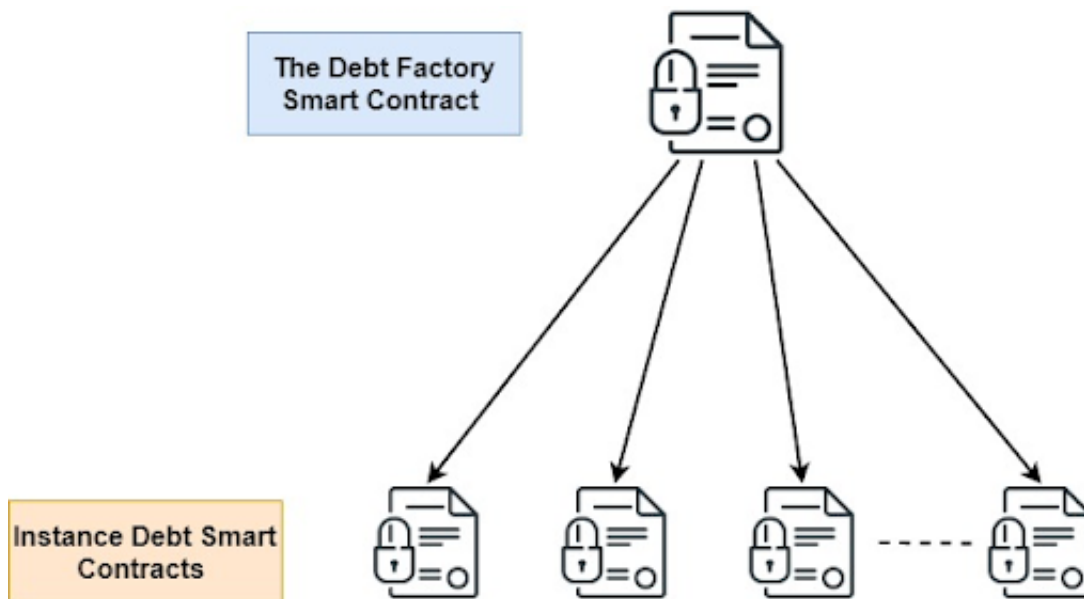


Figure 3.4: Relationship between the Debt Factory and Debt Smart Contract

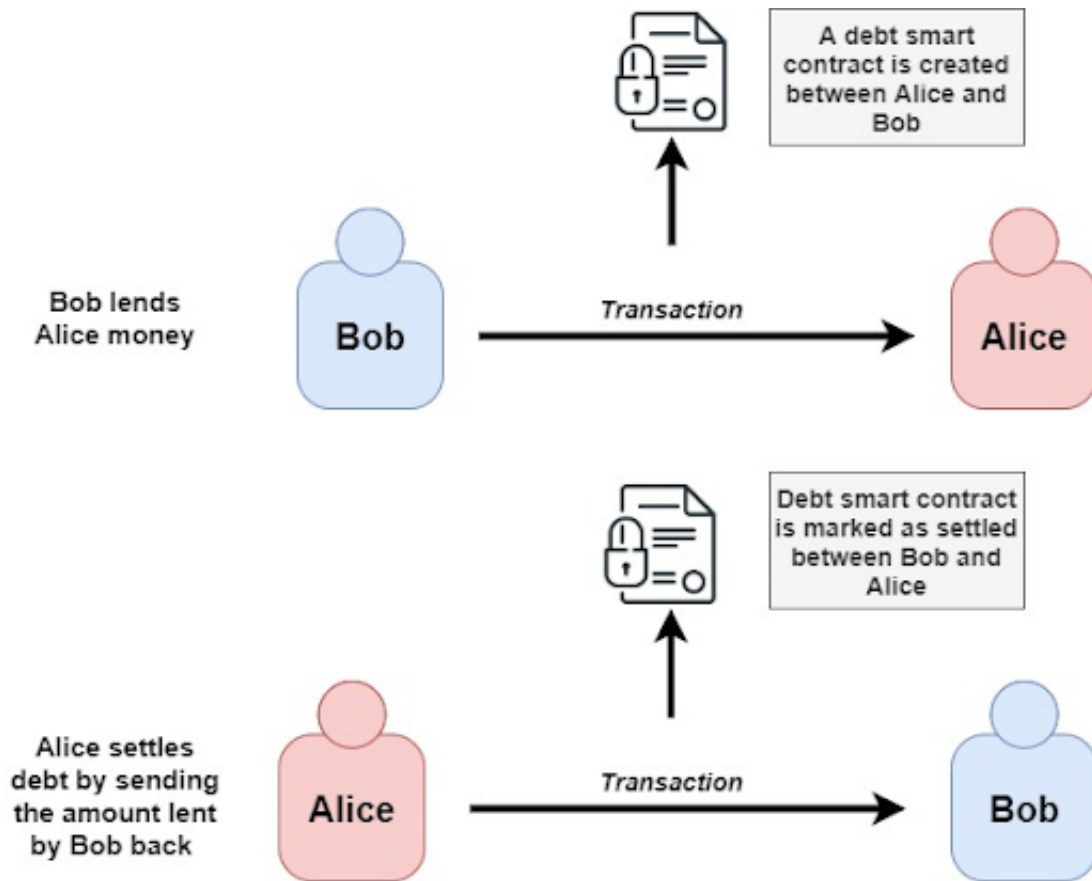


Figure 3.5: Peer-to-Peer Lending and Debt Settlement Process

3.3.1 Smart Contract Source Code

```
1 pragma solidity ^0.4.19;
2
3 //the Debt Factory contract
4 contract DebtFactory {
5     //Debt Factory state variable
6     address[] public debts;
7
8     //event definition
9     event ContractCreated(address newAddress);
10
11     //functions
12     function createDebt(uint _amount, address _borrower,
13 string _description, string _txHash) public returns (address){
14         require (msg.sender != _borrower);
15         address newDebt = new Debt(_amount, msg.sender, _borrower,
16 _description, _txHash);
17         debts.push(newDebt);
18         ContractCreated(newDebt);
19         return newDebt;
20     }
21
22     function getDeployedDebts() public view returns (address[]){
23         return debts;
24     }
25 }
```

```
25
26 //the Debt instance contract
27 contract Debt {
28     //state variables
29     uint public amount;
30     address public borrower;
31     address public lender;
32     string public description;
33     bool public is_settled;
34     string public txHash;
35
36     //constructor
37     function Debt(uint _amount, address _lender, address _borrower,
38         string _description, string _txHash) public {
39         lender = _lender;
40         amount = _amount;
41         borrower = _borrower;
42         description = _description;
43         txHash = _txHash;
44         is_settled = false;
45     }
46     //functions
47     function settleDebt() public onlyBorrower{
48         is_settled = true;
49     }
50
51     function getDetails() public view returns (
52         address, address, uint, string, bool, string
53     ) {
54         return (
55             lender,
56             borrower,
57             amount,
58             description,
59             is_settled,
60             txHash
61         );
62     }
63     //function modifier
64     modifier onlyBorrower() {
65         require(msg.sender == borrower);
66         _;
67     }
68 }
```

Here's how the preceding code works:

- In the first line, the Solidity version is specified to 0.4.19, then the smart contracts, state variables, functions, events, and function modifiers are declared.
- The first contract declared is the debt factory contract. It has the debt state variable, which is an array of addresses and is declared public. This variable contains an array of all the deployed instances of the debt contract. Then, we have an event called **ContractCreated**, which will be used for capturing the addresses of newly created smart contracts. This event will be kept in the blockchain.
- Then, a function called **createDebt()**, which takes four arguments is declared,

these arguments are the debt contract state variables. This function allows users to create instances of the debt contract or debt records and returns the address of the newly created debt smart contract.

- Then, a function called **getDeployedDebts()** that returns all the deployed debt contracts is declared.
- Then, a constructor function that initializes our state variables to local variables that can be passed as an argument to functions is declared.
- Then, a **settleDebt()** function that changes a debt contract's state to settled by setting the **is_settled** state variable to true is declared. This function has an **onlyBorrower** function modifier that makes sure that no one other than the borrower can settle the debt.
- Then, there's a **getDetails()** function that returns information about any given smart contract.
- Finally, the **onlyBorrower** function modifier is declared at the bottom of the code, which is a common practice.

3.3.2 Compiling Solidity

The smart contract written in Solidity programming language will be passed to the solidity compiler, which will output the ABI representing the javascript interpretation layer of the smart contract and the contract's byte code, which will be deployed to the Rinkeby test network.

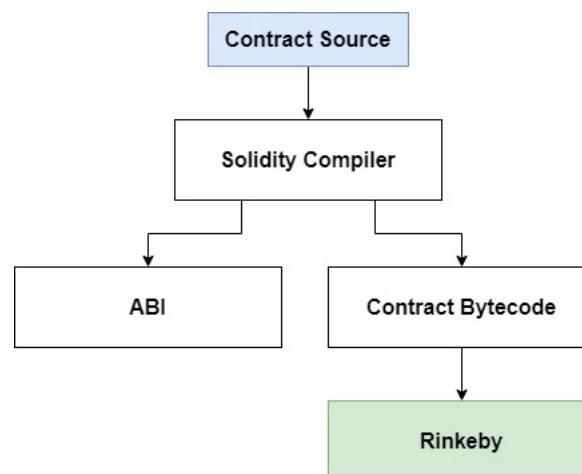


Figure 3.6: Compiling Smart Contracts

Compile Script

```
1 const path = require('path')
2 const solc = require('solc')
3 const fs = require('fs-extra')
```

```
4
5 const buildPath = path.resolve(__dirname, 'build')
6 fs.removeSync(buildPath)
7
8 const debtPath = path.resolve(__dirname, 'contracts', 'Debt.sol')
9 const source = fs.readFileSync(debtPath, 'utf8')
10
11 const output = solc.compile(source, 1).contracts
12 fs.ensureDirSync(buildPath)
13
14 for (let contract in output) {
15   fs.outputJsonSync(
16     path.resolve(buildPath, contract.replace(':', '')) + '.json',
17     output[contract]
18   )
19 }
```

Here's how the preceding code works:

- In the first three lines, the required standard library node modules are imported.
- The **path** module is going to build a path from our current directory. The reason why the path module is being used is that it guarantees cross-platform compatibility. In other words, if this compile script is run on a Windows or Linux based system, it'll always generate a valid build path.
- Then, a path for our solidity contract source is generated using the **path.resolve()** method that takes three arguments, the first is the current directory, and the other two are the contract source name and its parent directory.
- Now that the path is specified, the contents of the file will be read using the **fs.readFileSync()** method that takes two arguments, first is the path of our contract source and the second is the type of encoding used which in this case would be "UTF-8".
- Then, the **fs.compile()** method will be used to compile the contract source and store it in an output constant. The output would have the contract source bytecode and ABI.
- Finally, the for-loop takes the ABI from output and stores it in a JSON file to later be used with javascript, which will allow users to invoke methods and interact with the smart contract.

3.3.3 Deployment to Rinkeby

To deploy the smart contracts into Ethereum's Rinkeby test network, the Web3 instance will be instructed to communicate with MetaMask, which will serve as the wallet provider. The deployed contract is signed using the MetaMask external account used to deploy its mnemonic phrase. (see Appendix A, Installation guideline, installing MetaMask)

Deployment Script

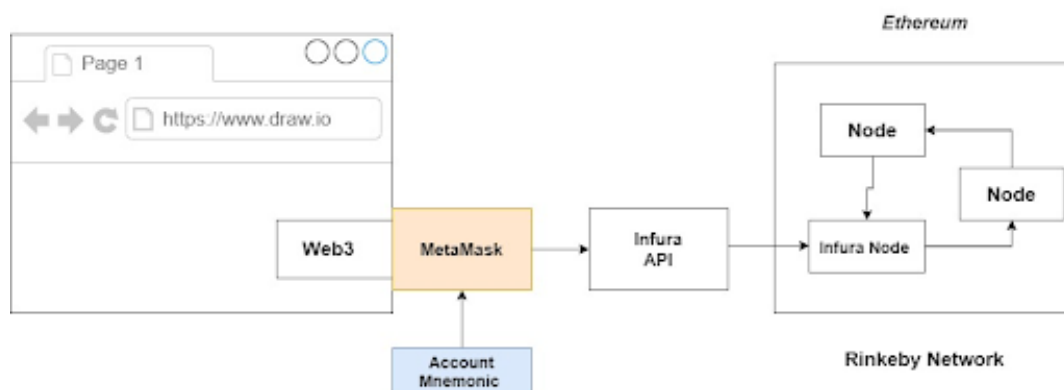


Figure 3.7: Deployment to Rinkeby

```
1 const HDWalletProvider = require("truffle-hdwallet-provider");
2 const Web3 = require("web3");
3 const DebtFactory = require("../build/DebtFactory.json");
4
5 const mnu =
6   "record rice genuine entire pact castle lightening typical
7   wrestle trust assist glimpse";
7 const network = "https://rinkeby.infura.io/v3/315
8   df566c9244aa2b9608a31ba46bc8a";
9
9 const provider = new HDWalletProvider(mnu, network);
10
11 const web3 = new Web3(provider);
12
13 const deploy = async () => {
14   const accounts = await web3.eth.getAccounts();
15
16   console.log("Attempting to deploy from account", accounts[0]);
17
18   const result = await new web3.eth.Contract(
19     JSON.parse(DebtFactory.interface)
20   )
21     .deploy({ data: DebtFactory.bytecode })
22     .send({ from: accounts[0], gas: "1000000" });
23
24   console.log("Contract deployed to", result.options.address);
25 };
26 deploy();
```

Here's how the preceding code works:

- At the very top, the node modules are imported. First, the MetaMask HD Wallet Provider, then there's the Web3 constructor that'll allow us to perform our function calls.
- Then, the compiled contract's source ABI and bytecode are imported.
- Next, the HDWalletProvider, which will tell the Infura node which account will be used to deploy our smart contract, is set up by passing the account's mnemonic and the Infura node used to deploy the smart contract into Ethereum's endpoint as arguments.

- Then, there's the deploy asynchronous function that will deploy our smart contract and then output the smart contract's address once it's deployed on the Network.
- Finally, the contract is deployed by calling the **deploy()** function.

Running the deployment script will result in the following output in the terminal:

```
metro@VBOX:/media/metro/bb204815-ba20-426b-8c86-0e9a71bf1e3d/home/metro/Workspac
e/Ethereum/DC/ethereum$ node deploy
Attempting to deploy from account 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
Contract deployed to 0xA315ba0D55b4602E763314f1aa18e6F63A658aC4
```

The deployed contract's address will be used later in building the Front-end inter-
face to allow users to interact with the smart contract on the Ethereum Blockchain.

3.3.4 Automated Testing

To perform asynchronous testing on the smart contract, Mocha's testing framework will allow us to test the smart contract based on certain testing conditions.

Testing conditions:

1. Deployment of smart contracts instances
2. Allowing users to lend each money
3. The creation of a debt record between two parties
4. Allowing borrowers to settle debts

End-to-End Testing

End-to-End testing means combining all of the testing conditions into one condition that's executed sequentially.

Testing Script

```
1 const assert = require('assert')
2 const ganache = require('ganache-cli')
3 const provider = ganache.provider()
4 const Web3 = require('web3')
5 const web3 = new Web3(provider)
6
7 const compiledFactory = require('../ethereum/build/DebtFactory.json')
8 const compiledDebt = require('../ethereum/build/Debt.json')
9
10 let accounts;
11 let debtAddress;
12 let debt;
13 let factory;
14
15 beforeEach(async () => {
16     accounts = await web3.eth.getAccounts()
17     amount = 1000000
```

```
18     description = "test-debt"
19     Bob = accounts[0]
20     Alice = accounts[1]
21
22     factory = await new web3.eth.Contract(JSON.parse(
23     compiledFactory.interface))
24     .deploy({ data: compiledFactory.bytecode })
25     .send({ from: caller, gas: '1000000' })
26
27     await factory.methods.createDebt(amount, accounts[1],
28     description, "random txHash").send({
29     from: caller,
30     gas: '1000000'
31     })
32
33     deployedDebts = await factory.methods.getDeployedDebts().call()
34     ;
35
36     debt = await new web3.eth.Contract(JSON.parse(compiledDebt.
37     interface),
38     deployedDebts[0]);
39 });
40
41 describe("P2P Lending & Debt Management DApp", async () => {
42     it("deploys contracts", () => {
43         assert.ok(factory.options.address)
44         assert.ok(debt.options.address)
45     });
46     it("allows users to lend each other money", async () => {
47         aliceBalance = await web3.eth.getBalance(Alice)
48         await web3.eth.sendTransaction({
49             from: Bob,
50             to: Alice,
51             value: amount
52         });
53         aliceNewBalance = await web3.eth.getBalance(Alice)
54         assert(aliceNewBalance==aliceBalance+amount)
55     });
56     it("creates a debt record between two parties", async () => {
57         tx = await web3.eth.sendTransaction({
58             from: Bob,
59             to: Alice,
60             value: amount
61         });
62         _txHash = tx.transactionHash
63         await factory.methods.createDebt(amount, Alice, description
64         , _txHash).send({
65             from: Bob,
66             gas: '1000000'
67         });
68         debts = await factory.methods.getDeployedDebts().call();
69         debt = await new web3.eth.Contract(JSON.parse(compiledDebt.
70         interface),
71         debts[1]);
72         debtDetails = await debt.methods.getDetails().call();
73         assert(debtDetails[5]==_txHash)
74     });
75     it("allows borrowers to settle debt", async () => {
```

```

70     tx = await web3.eth.sendTransaction({
71         from: Bob,
72         to: Alice,
73         value: amount
74     });
75     _txHash = tx.transactionHash
76     await factory.methods.createDebt(amount, Alice, description
77 , _txHash).send({
78         from: Bob,
79         gas: '1000000'
80     });
81     debts = await factory.methods.getDeployedDebts().call();
82     debt = await new web3.eth.Contract(JSON.parse(compiledDebt.
83 interface),
84     debts[1]);
85     await debt.methods.settleDebt().send({
86         from: Alice,
87         gas: '1000000'
88     });
89     });
90     it("end-to-end testing", async()=> {
91         tx = await web3.eth.sendTransaction({
92             from: Bob,
93             to: Alice,
94             value: amount
95         });
96         _txHash = tx.transactionHash
97         await factory.methods.createDebt(amount, Alice, description
98 , _txHash).send({
99             from: Bob,
100             gas: '1000000'
101         });
102         debts = await factory.methods.getDeployedDebts().call();
103         debt = await new web3.eth.Contract(JSON.parse(compiledDebt.
104 interface),
105         debts[1]);
106         debtDetails = await debt.methods.getDetails().call();
107         await debt.methods.settleDebt().send({
108             from: Alice,
109             gas: '1000000'
110         });
111         assert(debtDetails[5]==_txHash)
112     });
113 })

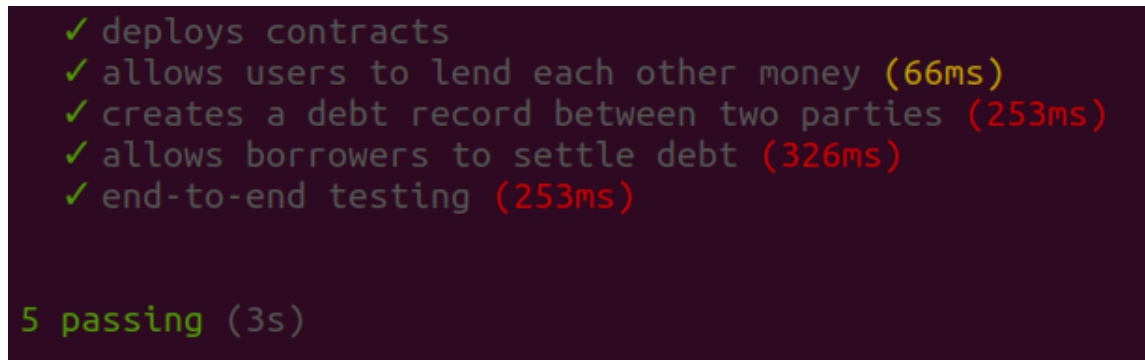
```

Here's how the preceding code works:

- At the very top, the required node modules are imported - the assert library will allow us to make assertions in testing. Ganache (formerly known as Test-RPC), which will enable us to interact with compiled smart contracts and gives us testing accounts will be used. Finally, the Web3 library.
- Then, the JSON files outputted from compiling the smart contract are required.
- Finally, the testing conditions are constructed to test the specified smart con-

tract functionalities.

Running the testing script will result in the following output illustrated in the image below:



```
✓ deploys contracts
✓ allows users to lend each other money (66ms)
✓ creates a debt record between two parties (253ms)
✓ allows borrowers to settle debt (326ms)
✓ end-to-end testing (253ms)

5 passing (3s)
```

3.4 Building a Front-end Interface

Two interactive user-interfaces are built for the Decentralized Application front-end. The goal of building a front-end interface is to allow users to interact with the Decentralized Application through a user-friendly interactive interface. The two interfaces are the lending interface and the debt record interface. React Front-end JavaScript framework along with Semantic-UI are used in building the Front-end Interface. React is a front-end framework that breaks Web and Mobile Applications into components. Semantic-UI has ready-made HTML components like buttons, banners, etc that can be used in React instead of building those components from scratch.

The full code used to build the Front-end Interfaces can be found in Appendix B.

3.4.1 Lending Interface

In this interface, users will be able to lend each other money. At the top, there's a note that instructs users to connect to MetaMask or any wallet provider and to ensure that they have sufficient funds before proceeding with the lending process. A user enters a description that explains why they're lending the other person, the borrower's address, and the amount to be borrowed. After filling all these pieces of information, the user hits "send".

After hitting "send", a loading spinner appears. After the information entered by the user is validated, a MetaMask popup notification appears, telling the user how much ether will be sent, the gas fee, and in which network this transaction is going through.

Finally, the user hits "confirm" and waits for the transaction to go through. After

Note
Make sure you're connected to [MetaMask](#) or any wallet provider and have sufficient funds.

Description
New IKEA table

Borrower
0xb8220F02BfDAB556123F638c454aFaA4B243d7E6

Amount
0.001

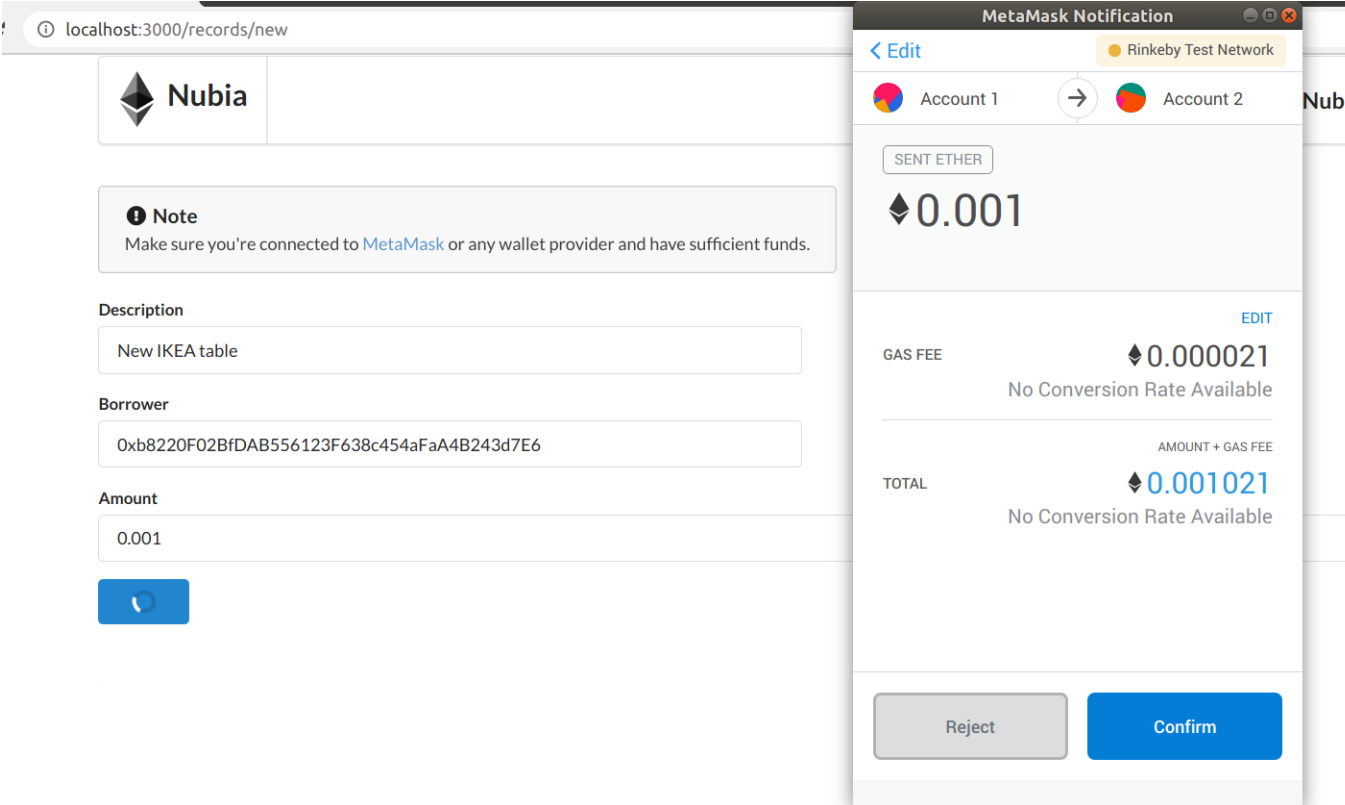
Send

the transaction goes through, a Debt smart contract is created between the user (lender) and the borrower, and the user is redirected to the debt record interface.

3.4.2 Debt Record Interface

In the debt contract, details are displayed along with the transaction receipt that generated the debt contract between the two parties. Both the debt contract and the transaction receipt have a QR code that redirects to the Ethereum Explorer (Etherscan) to show the proof of existence in the Ethereum Blockchain once it's scanned. As illustrated in the below screenshot, this debt is marked as unsettled, but there's no settle button mainly because the lender cannot settle the debt; only the borrower can settle the debt.

When the borrower visits the debt contract, the settle button appears, allowing them to settle the debt by paying the borrower back.



Settling Debt

When the borrower wants to settle a debt, they hit the “settle” button, and a MetaMask pop-up notification will appear. The notification tells them how much Ether will be sent, the gas fee, and where this transaction is going to take place.

The borrower then can confirm and wait for the transaction to go through. After the transaction goes through, the interface refreshes, and the debt contract is marked as “settled”.

Debt Contract Details

[0xaB2770cF02291ccAA1067794a2d677381247C000](#)
[See in Etherscan](#)

Debt Amount: 0.001 ETH
Description: New IKEA table
Lender: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
borrower: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Settled: 



Transaction Receipt

[0x77386bec1753d217be82f6da62e1e01f4b0a3ae08e1cc188c44b224ab74c2d04](#)

Thu, 19 Mar 2020 19:39:12 GMT

From: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
To: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Amount Paid: 0.001 ETH

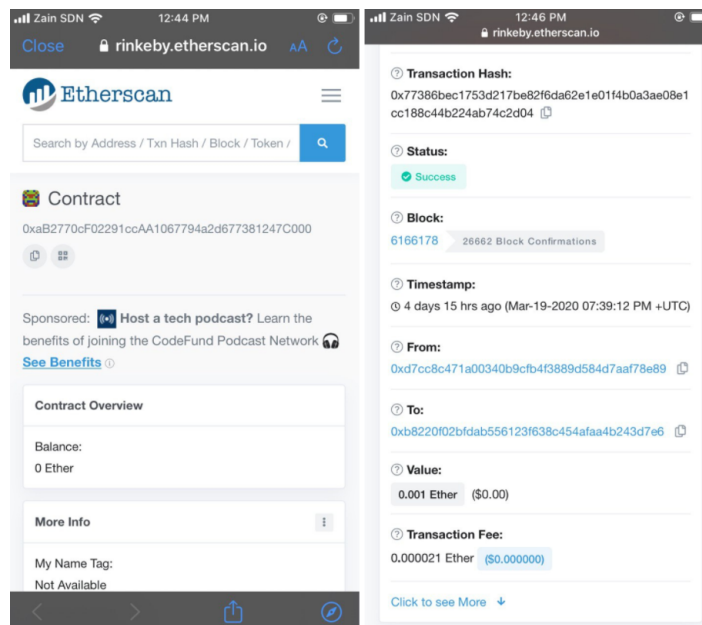


Figure 3.8: The Debt contract and transaction receipt on Etherscan after scanning their QR Codes

Debt Contract Details

[0x76B68EBCc6AA30c029D31CF1445bF7A44B1c3e1e](#)
[See in Etherscan](#)

Debt Amount: 0.000001 ETH
Description: New IKEA couch
Lender: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
borrower: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Settled: 



Settle

Transaction Receipt

[0xed679c1d71707de2e3e223f8966f1e701b96aca779f9024322a69beab313a688](#)
Sun, 26 Jan 2020 12:49:13 GMT


From: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
To: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Amount Paid: 0.000001 ETH



localhost:3000/records/0xaB2770cF02291ccAA1067794a2d677381247C000

Debt Contract Details

[0xaB2770cF02291ccAA1067794a2d677381247C000](#)
[See in Etherscan](#)

Debt Amount: 0.001 ETH
Description: New IKEA table
Lender: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
borrower: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Settled: 

Settle

Transaction Receipt

[0x77386bec1753d217be82f6da62e1e01f4b0a3ae08e1cc188c44b224ab74c2d04](#)
Thu, 19 Mar 2020 19:39:12 GMT

From: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
To: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Amount Paid: 0.001 ETH

MetaMask Notification

Rinkeby Test Network

Account 2 → Account 1

SENT ETHER

0.001

EDIT

GAS FEE 0.000021 No Conversion Rate Available


AMOUNT + GAS FEE


TOTAL 0.001021 No Conversion Rate Available

Reject Confirm

Debt Contract Details

[0xaB2770cF02291ccAA1067794a2d677381247C000](#)
[See in Etherscan](#)


Debt Amount: 0.001 ETH
Description: New IKEA table
Lender: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
borrower: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Settled: 



Transaction Receipt

[0x77386bec1753d217be82f6da62e1e01f4b0a3ae08e1cc188c44b224ab74c2d04](#)
Thu, 19 Mar 2020 19:39:12 GMT

From: 0xD7cC8c471a00340b9CFb4f3889d584D7aaF78E89
To: 0xb8220F02BfDAB556123F638c454aFaA4B243d7E6
Amount Paid: 0.001 ETH



Chapter 4

Evaluation

4.1 Costs

As mention in chapter two, the deployment of Ethereum smart contracts and the interaction with these contracts consume gas and have to be paid by the sender of a transaction. This section discusses the evaluation of the debt factory and the debt instance smart contract discussed in chapter three in terms of deployment and interaction costs.

4.1.1 Deployment Costs

According to the gas costs spreadsheet published in the official Solidity documentation, the main costs for deploying a contract are the costs associated with storing the contract code (“CREATEDATA”) with costs of 200 gas per byte and the costs for storing additional data on the storage of the contract (“STORAGEADD”) with 20000 gas per 256-bit word. Further, in addition to the 21000 gas for a normal contract transaction, 32000 gas have to be paid for a transaction that creates a contract.

The arbitrary deployment costs for a contract dependent heavily on the size of the contract code and the number of bytes that it designates to the storage in its constructor. Since both the debt factory and the debt instance contract don’t perform any calculation-intensive work in their constructors, their deployment costs in gas can be estimated using the following formula:

$$C_{gas} = (53000 + 200 * N_{bytes} + 20000 * N_{words}) \quad (4.1)$$

Where C_{gas} are the total transaction costs in gas, N_{bytes} is the contract size in bytes and N_{words} is the number of 256 bit words that are initialized in the constructor. To calculate the deployment price in US-Dollar, the gas usage has to be multiplied with the the gas price P_{gas} and the US-Dollar exchange rate for the ether $P_{exchange}$:

$$C_{dollar} = C_{gas} * P_{gas} * P_{exchange} * 10^{-18} \quad (4.2)$$

Empirical Results:

4.1.2 Transaction Costs

The transaction costs for the debt factory and debt instance contracts are dominated by the fix transaction costs (“GTX”) of 21000 gas, the costs for adding a 256 bit word to the storage (“STORAGEADD”) of 20000 gas, the costs for modifying a word on the storage (“STORAGEMOD”) of 5000 gas and the costs for making a call from the contract that contains ether (“GCALLVALUETRANSFER”) of 9000 gas. Other costs are not significant since no mayor computation is done in any of the transaction functions. Therefore, to estimate the transaction costs, the following formula can be used:

$$C_{gas} = 21000 + 20000 * N_{words_{add}} + 5000 * N_{words_{mod}} + 9000 * N_{tx} \quad (4.3)$$

Where C_{gas} are the total costs in gas, $N_{words_{add}}$ are the number of 256 bit words added to the storage in the transaction, $N_{words_{mod}}$ are the number of word that are modified in the transaction and N_{tx} are the number of messages with ether that are sent in the transaction function (e.g. for refunding ether to the buyer or seller). The total costs in US-Dollar can be estimated by equation 4.2.

4.2 Application Scalability

At the moment the Web application relies on an external server that hosts an Ethereum client to interact with contracts on the blockchain (chapter 3). This configuration has scalability issues when multiple Web clients connect to the same server. Eventually, the network interface will be a bottleneck, especially when many contracts have to be synchronized concurrently. Since the current application is only a prototypical implementation and not intended to be used in a production environment, these scalability issues are not further addressed.

4.3 Security

As explained in section 3.1, user accounts are managed on MetaMask’s browser extension to interact with the smart contracts. MetaMask hasn’t suffered any major hacks. It uses HD backup settings and has a strong community of developers updating its open-source code. However, the wallet is online so it’s more at risk than hardware wallets and other forms of cold storage. The most common risk facing the MetaMask wallet are phishing attacks. A phishing attack is a scam hackers use to steal personal information like usernames and passwords.

In addition to phishing attacks on MetaMask, there are only two scenarios in which an attacker could obtain or use the private key of an account:

- The attacker has root access on the operating system and can intercept the password from the user by using a key logger.
- The attacker has physical access to the device and the account is still unlocked. In this case, the attacker could use the account to sign debt contracts and transfer money to their own account.

4.4 Privacy

This section discusses how privacy issues that arise from storing and exchanging personal user data are addressed in the application.

In this application, users interact on an address to address basis without having to reveal their true identity, thus their user data remains anonymous as long as their addresses are anonymous.

However, A party that knows a person that belongs to a particular address (e.g. when it exchanged contract or user data with that person in the past) could look up details of other contracts that were signed by that person.

In addition, Storing contract details in plain text on a smart contract can cause privacy issues because the addresses of the seller and the buyer are also stored on the contract and are therefore publicly accessible.

Chapter 5

Summary & Conclusion

5.1 Summary

The introduction chapter posed multiple questions that were addressed in different chapters of the thesis.

Chapter two discussed the literature review that made this project feasible. It addresses the evolution that shapes the world wide web as well as the building blocks of the Blockchain technology and Ethereum.

Chapter three discussed the design and implementation of a Web application to conclude the Debt smart contract that uses the Ethereum blockchain to store the contractual details, enforce compliance and process the lending and debt management.

Section 3.1 shows how interfacing with Ethereum networks is achieved for both Developers and consumers and the technologies that make this possible.

Section 3.2 discusses Ethereum's application infrastructure and outlines the similarities and major differences between it and the traditional application infrastructure.

Section 3.3 discusses the design and implementation of the smart contracts and the relationship between them. The following subsections show how the smart contracts are implemented in the solidity programming language, their compilation into Bytecode and ABI, deployment and automated testing as well as their integration with a web application client that's powered by JavaScript and nodeJS.

Section 3.4 discusses how a user-friendly interface is built to allow users to interact with the Decentralized Application's smart contracts, enabling them to lend each other money and creating a debt contract that facilitates the financial transactions and debt between individuals.

In chapter four, the web-based Decentralized application was evaluated in terms of costs, scalability, security, and privacy. The specific deployment and transaction costs were measured and a simple equation was derived that can account for most of the costs for the smart contracts.

It was concluded that the application offers a high degree of security and privacy because it signs transactions on the local user's device, stores contracts, and personal information only on the blockchain which is secured by hash power and never sends this information unencrypted over the network.

5.2 Conclusion

The implemented Decentralized web application satisfies the functional and non-functional requirements stated at the beginning of this thesis. It allows two parties to conclude valid electronic debt smart contracts. It is completely independent of any TTP and uses the Ethereum blockchain to store the contract details, enforce the contractual clauses and facilitate the payments made in the lending process in ether. Users specify the contract details including the amount to be lent, the borrower, a textual description and deploy the contract on the blockchain. The contract details can be exchanged either by scanning the QR-code of the contract or by sharing the link of the contract. The application signs contracts on the user's local device and protects personal user data by storing it only on their device and by encrypting it when it is sent over the network.

5.3 Limitations & Future Work

Appendix A

Installations Guideline

A.1 Installing and Using MetaMask

In order to connect and interact with the Ethereum Blockchain. Users will need to install a browser extension called MetaMask. Which is a wallet provider that will allow them to store and secure their crypto assets as well as allow them to interact with Decentralized Applications.

A.1.1 Installing MetaMask

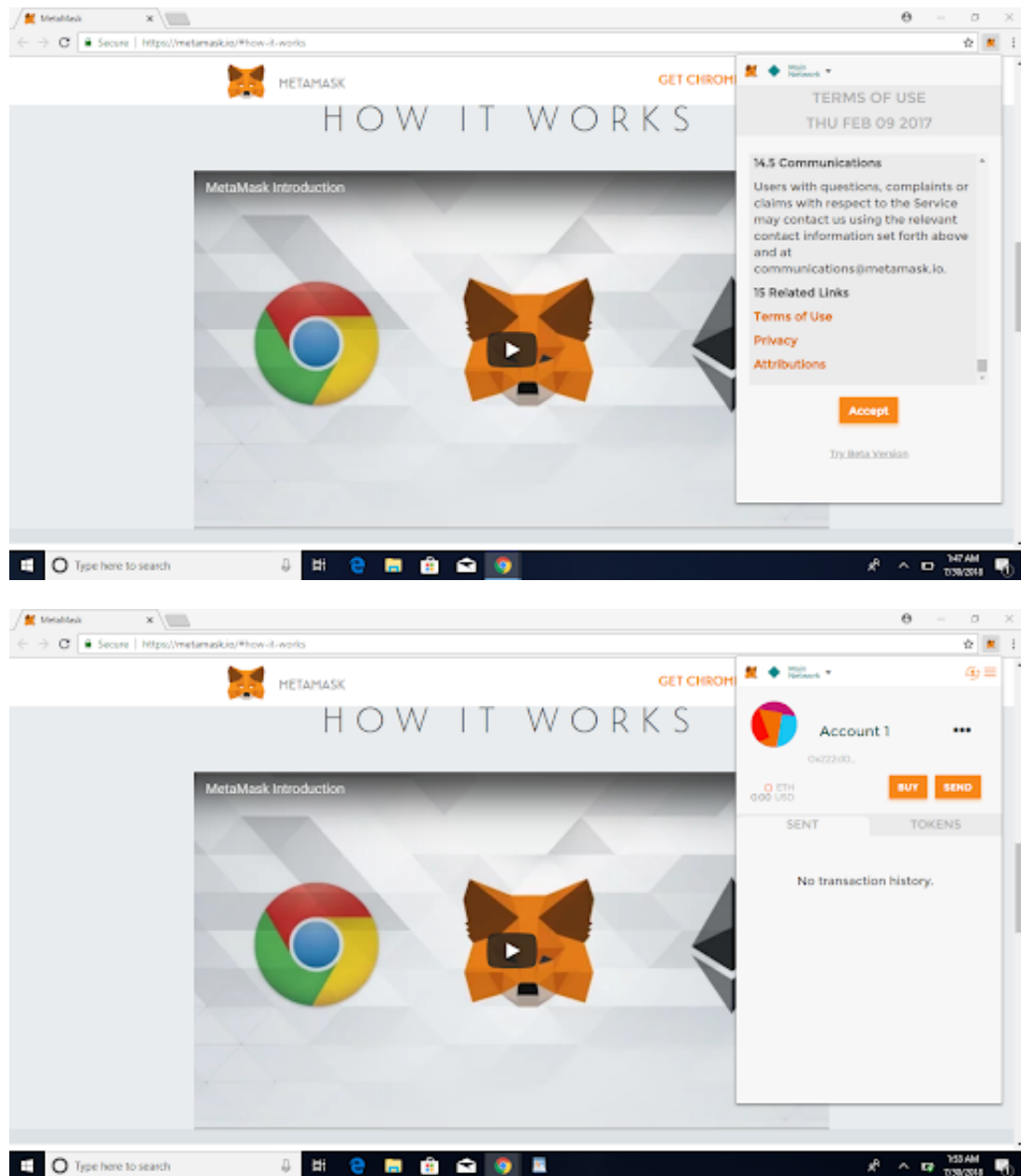
- Step 1.** Go to the Metamask website.
- Step 2.** Click “Get Chrome Extension” to install Metamask.
- Step 3.** Click “Add to Chrome” in the upper right.
- Step 4.** Click “Add Extension” to complete the installation.

You will know Metamask has been installed when you see the fox logo on the upper right-hand corner of your browser.



A.1.2 Using MetaMask

- Step 1.** Click on the Metamask logo in the upper right-hand corner of your Google Chrome browser.
- Step 2.** Read and agree to the terms and conditions. You may have to agree to 2 to 3 pages worth of terms.
- Step 3.** Enter a password and click “Create” to create your wallet.
- Step 4.** You will see a set of 12 “seed words” called a mnemonic for your vault. Click “Save Seed Words as File” and copy the “MetaMask Seed Words” file that is downloaded to a safe place. You will need it to access your vault.
- Step 5.** Click “I’ve Copied It Somewhere Safe” once your seed words file has been secured. You’ll be taken into your Metamask wallet!



You are now in the Ethereum mainnet network. To start experimenting with Meta-mask, you can switch to one of the testnet networks by clicking “Main Network” in the left-hand corner of the wallet pop up screen, and selecting one of the testnets such as Ropsten Test Network or Kovan Test Network.

A.2 Cloning the Project's GitHub Repository

Cloning the Project's GitHub Repository To clone the project's GitHub repository, Git or a Git client needs to be installed in the system.

After installing git or the git client, we enter the following commands in the terminal:

```
git clone https://www.github.com/kemoszn/OpenDL.git
```

After the cloning the project, we enter the following command to install the dependencies:

```
npm install
```

To run the project, open the clone directory and navigate to OpenDL/ and run the following command:

```
npm run dev
```

After running the project, go to **localhost:3000** in any browser that has MetaMask installed and the project should be up and running.

A.2.1 Running Scripts

For the compile and deploy, navigate to OpenDL/ethereum To run the compile script, the following command has been used:

```
node compile
```

To run the deploy script, the following command has been used:

```
node deploy
```

To run the automated tests, use the following command:

```
npm run test
```


Appendix B

Front-End Interface Source Code

B.1 Lending Interface Source Code

```
1 import React, {Component} from 'react';
2 import Layout from '../components/Layout';
3 import { Form, Button, Input, Message, Icon } from 'semantic-ui-
  react';
4 import factory from '../ethereum/factory';
5 import web3 from '../ethereum/web3';
6 import { Router } from '../routes';
7
8 class NewRecord extends Component {
9   state = {
10     description: "",
11     borrower: "",
12     amount: "",
13     errorMessage: "",
14     loading: false,
15     redirectedAddress: "",
16     txHash: ""
17   }
18
19   onSubmit = async (event)=> {
20     event.preventDefault();
21
22     this.setState({ loading: true, errorMessage: '' });
23     try {
24       const accounts = await web3.eth.getAccounts();
25
26       await web3.eth.sendTransaction({
27         from: accounts[0],
28         to: this.state.borrower,
29         value: web3.utils.toWei(this.state.amount, 'ether')
30       }).then((receipt) => this.setState({ txHash: receipt.
transactionHash}));
31       console.log(this.state.txHash);
32       await factory.methods
33         .createDebt(web3.utils.toWei(this.state.amount, '
ether'),
34         this.state.borrower, this.state.description, this.
state.txHash)
35         .send({from: accounts[0]})
```

```
36         .then((receipt) => this.setState({
37           redirectedAddress: receipt.events.ContractCreated.returnValues
38             [0] })
39         );
40         Router.pushRoute(`/records/${this.state.
41           redirectedAddress}`);
42         } catch(err) {
43           this.setState({ loading: false, errorMessage: err.
44             message });
45         }
46       }
47
48       render(){
49         return (
50           <Layout>
51             <br/>
52             <div>
53               <Message compact="true">
54                 <Message.Header><Icon name="exclamation circle
55                   "></Icon>Note</Message.Header>
56                 <p>
57                   Make sure you're connected to <a href="
58                     https://www.metamask.io">MetaMask</a> or any wallet provider and
59                   have sufficient funds.
60                 </p>
61               </Message>
62             </div>
63             <br/>
64             <Form widths="equal" onSubmit={this.onSubmit} error={!!
65               this.state.errorMessage}>
66               <Form.Field>
67                 <label> Description</label>
68                 <Form.Input width={8} value={this.state.
69                   description}>
70                   <input type="text" value={this.state.description}
71                     onChange={event => this.setState ({
72                       description: event.target.value}) }/>
73                 </Form.Field>
74                 <Form.Field>
75                   <label> Borrower</label>
76                   <Form.Input width={8} value={this.state.
77                     borrower}>
78                     <input type="text" value={this.state.borrower}
79                       onChange={event => this.setState ({
80                         borrower: event.target.value}) }/>
81                   </Form.Field>
82                   <Form.Field>
83                     <label> Amount</label>
84                     <Input fluid="false" label="ETH"
85                       labelPosition="right"
86                       value={this.state.amount}
87                       onChange={event => this.setState ({ amount:
88                         event.target.value}) }/>
89                   </Form.Field>
90                   <Message error header="Oops!" content={this.state.
91                     errorMessage}/>
92                   <Button loading={this.state.loading} primary> Send
93                   </Button>
94                 </Form>
95             </div>
96           </Layout>
97         );
98       }
99     }
100   }
101 }
102
103 export default App;
```

```
79         </Layout>
80     );
81 }
82 }
83
84 export default NewRecord;
```

Here's how the preceding code works:

- At the very top, node dependencies and user-interface components are imported. First, the React framework is imported. Then, there's the layout component that's used to render the navigation menu. Then, HTML components are imported from the Semantic-UI library. Then, The factory contract's JavaScript representation and web3 are imported. At last, Router is imported which helps in navigating between pages in React.
- Then, the input form is constructed as a class-based component that extends React's component library.
- First, we declare the debt smart contract state variables to be blank. Then, we have the **onSubmit()** function that's invoked whenever a user hits "send". This function sends the transaction from the lender to the borrower, creates a debt record between them based on the transaction's hash and redirects the lender to the debt record's interface.
- Finally, the HTML user interface components are implemented and the React component is exported.

B.2 Debt Record Interface Source Code

```
1 import React, { Component } from 'react';
2 import QRCode from 'qrcode.react'
3 import Layout from '../components/Layout';
4 import Debt from '../ethereum/debt';
5 import web3 from '../ethereum/web3';
6 import { Form, Card, Button, Icon, Message, Grid } from 'semantic-
  ui-react';
7 import { Router } from '../routes';
8
9 class Detail extends Component {
10     state = {
11         loading: false,
12         errorMessage: "",
13         userAccount: "",
14         from: "",
15         to: "",
16         value: "",
17         blockNumber: "",
18         timestamp: "",
19     }
20     async componentDidMount(){
21         const accounts = await web3.eth.getAccounts();
22         this.setState ({ userAccount: accounts[0], str: this.props.
  address });
23         await web3.eth.getTransaction(this.props.txHash)
```

```
24     .then((data)=> this.setState({from: data.from, to: data.to,
25     value: web3.utils.fromWei(data.value, "ether"),
26     blockNumber: data.blockNumber }));
27     const timestamp = await web3.eth.getBlock(this.state.
blockNumber);
28     const d = new Date(timestamp.timestamp * 1000);
29     const s = d.toUTCString();
30     this.setState({ timestamp: s });
31     const strLink1 = 'https://rinkeby.etherscan.io/address/' +
this.props.address;
32     document.getElementById("link1").setAttribute("href",
strLink1)
33   }
34
35   static async getInitialProps(props) {
36     const debt = Debt(props.query.address);
37
38
39     const details = await debt.methods.getDetails().call();
40     return {
41       address: props.query.address,
42       lender: details[0],
43       borrower: details[1],
44       amount: web3.utils.fromWei(details[2], 'ether'),
45       description: details[3],
46       isSettled: details[4],
47       txHash: details[5]
48     };
49   }
50
51   onSettle = async (event) => {
52     event.preventDefault();
53
54     this.setState({ loading: true, errorMessage: ''});
55     try {
56       const accounts = await web3.eth.getAccounts();
57       const debt = await Debt(this.props.address);
58       await web3.eth.sendTransaction({
59         from: accounts[0],
60         to: this.props.lender,
61         value: web3.utils.toWei(this.props.amount, 'ether')
62       });
63       await debt.methods.settleDebt().send({from: accounts[0]});
64
65       Router.replaceRoute(`/records/${this.props.address}`);
66       this.setState({ loading: false });
67     } catch (err) {
68       this.setState({ loading: false, errorMessage: err.message
});
69     }
70   }
71
72   renderDetails() {
73     const {
74       address,
75       lender,
76       borrower,
77       amount,
```

```

78     description,
79     isSettled,
80   } = this.props;
81   let isSettledString = isSettled;
82   if (isSettled === false){
83     isSettledString = 'times circle'
84   } else { isSettledString = 'check circle' }
85   const addressQR = 'https://rinkeby.etherscan.io/address/' +
address;
86   const items = [
87     {
88       header: (<div><p style={{color: "#2185d0" }}><b>{address}</b></p> </div> ),
89       meta: (<div><a id="link1">See in Etherscan <Icon name="
external alternate"></Icon></a></div>),
90       description:
91         (<div>
92           <Grid coloumns={2}>
93             <Grid.Row>
94               <Grid.Column width={13}> <br/><br/>
95               <b>Debt Amount: </b>{amount} ETH <br/>
96               <b>Description: </b>{description} <br/>
97               <b> Lender:</b> {lender} <br/>
98               <b> borrower: </b>{borrower}
99               <br/> <b> Settled: </b> <Icon name={isSettledString} />
100             </Grid.Column>
101             <Grid.Column width={3}>
102               <QRCode size={150} value={addressQR} />
103             </Grid.Column>
104             </Grid.Row>
105             </Grid>
106             { borrower==this.state.userAccount && !isSettled &&
107             <div>
108               <hr/>
109               <Form onSubmit={this.onSettle} error={!this.state.
errorMessage}>
110                 <Message error header="Oops!" content={this.state.
errorMessage}/>
111                 <Button loading={this.state.loading} primary size="small
"> Settle </Button>
112               </Form></div>}
113               </div>),
114               fluid: true
115             }]}
116
117   return <Card.Group items={items}/>;
118
119   }
120
121   renderTransaction() {
122     const { txHash } = this.props;
123     const txQR = 'https://rinkeby.etherscan.io/tx/' + txHash;
124     const items = [
125       {
126         header: <p style={{color: "#2185d0" }}><b>{txHash}</b></p>
>,
127         description: (<div>
128           <Grid coloumns={2}>

```

```

129         <Grid.Row>
130         <Grid.Column width={13}> <br/>
131         <b> From: </b> {this.state.from} <br/>
132         <b>To:</b> {this.state.to}
133         <br/><b>Amount Paid: </b>{this.state.value} ETH <br/>
134         </Grid.Column>
135         <Grid.Column width={3}>
136         <QRCode size={100} value={txQR} />
137         </Grid.Column>
138         </Grid.Row>
139         </Grid>
140         </div>),
141         meta: `${this.state.timestamp}` ,
142         fluid: true
143     ]}
144     return <Card.Group items={items}/>;
145 }
146
147 render() {
148     return (
149         <Layout>
150         <div> <br/>
151         <h3><Icon name="address card outline"></Icon>Debt
Contract Details </h3>
152         {this.renderDetails()}
153         <h3><Icon name="file alternate outline"></Icon>
Transaction Receipt</h3>
154         {this.renderTransaction()}
155         </div>
156         </Layout>
157     )
158 }
159 }
160
161 export default Detail;

```

Here's how the preceding code works:

- At the very top, node dependencies and user-interface components are imported. First, the React framework is imported. Then, a QRCode component which will be used to generate QRCodes for the debt contract and the transaction receipt is imported. Then, there's the layout component that's used to render the navigation menu. Then, the debt contract's JavaScript representation and web3 are imported. At least, HTML components and Router are imported.
- Then, the detail class-based component that extends React's component library is constructed. In this interface, the web3 library will be used to read from the blockchain and only to write if the borrower desires to settle the debt.
- First, state variables are declared - the state variables declared are used to retrieve relevant information about the debt contract and the transaction receipt.
- Then, the **componetDidMount()** lifecycle method is constructed. This method will be invoked every time users visit the page. In this method, information about the debt contract and transaction receipt is retrieved using

Web3 API calls such as the transaction's timestamp and then converting it from an epoch format to UTC.

- Then, there's the **getInitialProps()** asynchronous method which will be invoked before the page renders for users. In this method, the **getDetails()** method is called on the debt contract to retrieve its detail information.
- Then, an **onSettle()** function is constructed which is invoked only when the borrower wishes to settle the debt.
- At last, HTML components are rendered and the Detail component is exported.

Bibliography

- [1] Wojciech Galuba and Sarunas Girdzijauskas. *Peer-to-Peer System*. 2009.
- [2] *How the Internet is Changing for the better*. URL: <https://blockdelta.io/the-decentralized-web-3-0-how-the-internet-is-changing-for-the-better/>. (Last visited: 16.08.2020).
- [3] *Bitcoin, Blockchain*. URL: <https://en.bitcoin.it/wiki/Blockchain>. (Last visited: 16.08.2020).
- [4] Don Tapscott and Alex Tapscott. *Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world*. Penguin, 2016.
- [5] Vitalik Buterin et al. “A next-generation smart contract and decentralized application platform”. In: *white paper* 3.37 (2014).
- [6] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Manubot, 2019.
- [7] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. ” O’Reilly Media, Inc.”, 2014.
- [8] *Cryptocurrency Market Capitalization*. URL: <https://coinmarketcap.com/>. (Last visited: 16.08.2020).
- [9] E Bucher. “Zustandekommen des Vertrages”. In: URL: http://www.eugen-bucher.ch/pdf_files/Bucher_ORAT_10.pdf, Last visit May 9 (2017).
- [10] *Bitcoin, Proof of Work*. URL: https://en.bitcoin.it/wiki/Proof_of_work. (Last visited: 16.08.2020).
- [11] *When Will the Last Bitcoin be Mined?* URL: <https://coincodex.com/article/2401/when-will-the-last-bitcoin-be-mined/>. (Last visited: 16.08.2020).
- [12] *Bitcoin Energy Consumption Index*. URL: <https://digiconomist.net/bitcoin-energy-consumption>. (Last visited: 16.08.2020).
- [13] *Proof of Stake FAQ*. URL: <https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ>. (Last visited: 16.08.2020).
- [14] *NXT Whitepaper*. URL: <https://web.archive.org/web/20150203012031/http://wiki.nxtcrypto.org/wiki/Whitepaper:Nxt#Blocks>. (Last visited: 16.08.2020).
- [15] Sunny King and Scott Nadal. “Ppcoin: Peer-to-peer crypto-currency with proof-of-stake”. In: *self-published paper*, August 19 (2012), p. 1.
- [16] *Ethereum Homestead Documentation*. URL: <https://ethdocs.org/en/latest/>. (Last visited: 16.08.2020).

- [17] Kevin Delmolino et al. “Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab”. In: *International conference on financial cryptography and data security*. Springer. 2016, pp. 79–94.
- [18] *Solidity Documentation*. URL: <https://solidity.readthedocs.io>. (Last visited: 16.08.2020).
- [19] *Remix Online IDE*. URL: <https://remix.ethereum.org/>. (Last visited: 16.08.2020).
- [20] *MetaMask*. URL: <https://metamask.io>. (Last visited: 16.08.2020).
- [21] *Web3 Documentation*. URL: web3js.readthedocs.io/en/v1.2.6/. (Last visited: 16.08.2020).
- [22] *TESTNET Rinkeby (ETH) Explorer*. URL: <https://rinkeby.etherscan.io/>. (Last visited: 16.08.2020).
- [23] *Ganache fast RPC Client*. URL: <https://github.com/trufflesuite/ganache-cli>. (Last visited: 16.08.2020).
- [24] *Infura: Ethereum API*. URL: <https://infura.io>. (Last visited: 16.08.2020).
- [25] *Node.js Documentation*. URL: <https://nodejs.org/en/>. (Last visited: 16.08.2020).
- [26] *Mocha - JavaScript Test Framework*. URL: <https://mochajs.org/>. (Last visited: 16.08.2020).
- [27] *React - A Javascript Library for building Client Interfaces*. URL: <https://reactjs.org>. (Last visited: 16.08.2020).
- [28] *Semantic-UI Documentation*. URL: <https://semantic-ui.com>. (Last visited: 16.08.2020).