A Decentralized Approach to Messaging Using Blockchain Technology

A graduate thesis project submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

By
Mahzad Stearns

May 2019
The thesis of Mahzad Stearns is approved by:

Prof. Robert McIlhenny, Ph.D. ___________________________ Date

Prof. Sevada Isayan ___________________________ Date

Prof. Jeff Wiegley, Ph.D. (Chair) ___________________________ Date

California State University, Northridge
ACKNOWLEDGEMENT

I would like to sincerely thank Prof. Jeff Wiegley for accepting and supporting the topic of this thesis.

My sincere thanks to Prof. Robert McIlhenny and Prof. Sevada Isayan for their help and guidance throughout the program.

I’m incredibly grateful for my husband, Scott who patiently stood by me every step of the way and gave me strength. I would not be able to be who I am today without his love and his support.
# Table of Contents

SIGNATURE PAGE ......................................................................................................................... ii

ACKNOWLEDGEMENT .................................................................................................................... iii

LIST OF FIGURES ............................................................................................................................ vi

LIST OF TABLES ............................................................................................................................... vii

ABSTRACT ......................................................................................................................................... viii

CHAPTER 1 - INTRODUCTION ........................................................................................................ 1

1.1 INTRODUCTION ....................................................................................................................... 1

1.1.1 Background ........................................................................................................................... 1

1.1.2 Development Overview ......................................................................................................... 1

CHAPTER 2 – BACKGROUND AND HISTORY ............................................................................. 2

2.1 RELATED WORK ......................................................................................................................... 2

2.1.1 Bitcoin ................................................................................................................................ 2

2.1.2 Blockchain ............................................................................................................................. 3

2.1.3 Smart Contracts .................................................................................................................... 6

2.2 TECHNICAL STRUCTURE .......................................................................................................... 7

2.2.1 Cryptography ........................................................................................................................ 7

2.2.2 Public and Private Keys ......................................................................................................... 7

2.2.3 Nonce and Hash Function .................................................................................................... 7

2.2.4 Blocks .................................................................................................................................. 9

2.2.5 Block Structure ...................................................................................................................... 9

2.2.6 Forming a Chain .................................................................................................................. 10

2.2.7 Transactions ......................................................................................................................... 10

2.2.8 Timestamp Server ................................................................................................................ 11

2.2.9 Blockchain Consensus Protocols ......................................................................................... 12

2.2.10 Network ............................................................................................................................... 14

2.2.11 Distributed Ledger .............................................................................................................. 15

2.2.12 The Disadvantages ............................................................................................................. 16

CHAPTER 3 – ETHEREUM BLOCKCHAIN .................................................................................... 17

3.1 ETHEREUM PHILOSOPHY ........................................................................................................ 17

3.2 ETHEREUM AND BITCOIN COMPARISON .......................................................................... 18

3.3 GAS AND PAYMENT ................................................................................................................ 19

3.4 ETHEREUM VIRTUAL MACHINE (EVM) .............................................................................. 20

3.5 ETHEREUM SMART CONTRACTS .......................................................................................... 21

3.6 DECENTRALIZED APPS OR DAPPS ....................................................................................... 22

3.7 TRANSACTION EXECUTION .................................................................................................... 22

3.8 INTERACTING WITH ETHEREUM NETWORK ........................................................................ 23

3.9 THE COMPETITION ................................................................................................................ 24

CHAPTER 4 – TOOLS OF IMPLEMENTATION ON BLOCKCHAIN .................................................. 25

4.1 STRUCTURE OF SMART CONTRACTS ..................................................................................... 25

4.2 DEPENDENCIES AND DEVELOPER SETUP ......................................................................... 27

4.2.1 Node.js and Node Package Manager ................................................................................... 27

4.2.2 React .................................................................................................................................... 27

4.2.3 Web3 ..................................................................................................................................... 28

4.2.4 Application Binary Interface (ABI) ....................................................................................... 29

4.2.5 EthereumJS .......................................................................................................................... 29

4.2.6 Truffle Framework ............................................................................................................... 29
List of Figures

Figure 1 - Centralized vs. Decentralized [27] ................................................................................................................. 3
Figure 2 - Blockchain Transactions [51] .......................................................................................................................... 4
Figure 3 - SHA256 Hash Function [54] .......................................................................................................................... 9
Figure 4 - Hashing a Block using SHA-256 ....................................................................................................................... 9
Figure 5 - How Blocks in the Blockchain Form [54] ......................................................................................................... 10
Figure 6 - How Transactions in blockchain work [5] .......................................................................................................... 11
Figure 7 - Timestamp server ............................................................................................................................................... 12
Figure 8 - Structure of a Blockchain [5] .......................................................................................................................... 15
Figure 9 - Merkle trees in Bitcoin [27] ............................................................................................................................ 18
Figure 10 - Ethereum Smart Contract [40].................................................................................................................... 21
Figure 11 - Instruction of a DApp [20] ............................................................................................................................ 22
Figure 12 - Interacting with Ethereum Network [38] ......................................................................................................... 23
Figure 13 - Ethereum and App Integration [50] ................................................................................................................. 23
Figure 14 - Running Migrations Using Truffle ................................................................................................................ 30
Figure 15 - Ganache Interface ........................................................................................................................................... 31
Figure 16 - MetaMask Extension for Chrome Browser .................................................................................................. 31
Figure 17 - Create custom RPC ................................................................................................................................-------- 32
Figure 18 - Reset the account ............................................................................................................................................. 32
Figure 19 - The localhost:9545 ....................................................................................................................................... 32
Figure 20 - Unlocked Accounts in Ganache Local Test Network [57] .............................................................................. 42
Figure 21 - The Testing Architecture [55] ....................................................................................................................... 43
Figure 22 - Mocha Structure [56] ..................................................................................................................................... 43
Figure 23 - Compile & Deploy to Rinkeby Structure ........................................................................................................ 45
Figure 24 - Rinkeby Faucet ................................................................................................................................................ 50
Figure 25 - Deployment Process to Main/Test Network [58] ........................................................................................... 51
Figure 26 - Signature Request from MetaMask .............................................................................................................. 58
List of Tables

TABLE 1 – UNIQUE FIXED SIZE HASH OUTPUT ............................................................................................................. 8
TABLE 2 - DENOMINATIONS OF UNIT IN ETHER [27] ................................................................................................ 19
Abstract

A Decentralized Approach to Messaging Using Blockchain Technology

By
Mahzad Stearns

Master of Science in Computer Science

People are increasingly relying on texting and messaging as opposed to talking in person to communicate. It is therefore important for individuals and groups with different identity and goals to have a secure and simple way to connect. However, there is rising privacy concerns for example with hacks and data sold on the web, denial of service to individuals, groups or organizations as mean of censorship, or data being mined for financial interests by advisors Originated from crypto-currencies, blockchain technology has been introduced to provide user access to numerous applications without relying on a third party. The implementation of the messaging DApps has the potential to bring several benefits to the society and the way user information or different services are handled. The main motivation to develop the ChatApp project was to increase message privacy. Many mainstream chat applications claim that, the applications are end to end encrypted. However, this claim is not verified. In this thesis, a new chatting application called ChatApp was developed. ChatApp is completely open source and uses end-to-end encryption. By using Ethereum platform, ChatApp does not require a centralized server and exists on Ethereum blockchain and on several thousand of nodes. Message encryption and decryption is done by a web app that sends the encrypted messages to the network. The smart contract does not store any messages or contacts, does not encrypt or decrypt messages, and is entirely done with web3. Since sending messages costs ether, an Ethereum wallet is required. In addition, there is a speed constraint with ChatApp since each transaction takes about 30 seconds to complete. Overall, a more secure chatting application is introduced that provides its users a higher level of privacy compared to commonly used messaging applications.
Chapter 1 - Introduction

1.1 Introduction

1.1.1 Background
Blockchain is one of the most promising emerging technologies. Beyond cryptocurrency, it's redefining how we store, update, and transfer data across networks. Blockchain and smart contracts are a completely new way to write and deploy applications. Blockchain has the potential and capacity to improve online security and user’s trust.

Global information transmission has become increasingly accessible. Through the power of the default, consensus mechanisms and voluntary respect of the social contract that it is possible to use the internet to make a decentralized value-transfer system, shared across the world and virtually free to use. Ethereum is a project, which attempts to build the generalized technology, a technology on which all transaction-based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto un-explored compute paradigm in the mainstream: a trustful object messaging compute framework. [1]

1.1.2 Development Overview
ChatApp is a decentralized messenger, which allows users to chat, transact and earn without limitations on censorship or privacy concerns. ChatApp lets users connect directly to a desired contact without the third party interference of other messengers. Send and receive messages within the DApp and chat with others directly without any centralized authority to have access to the conversations.
Chapter 2 – Background and History

2.1 Related Work

2.1.1 Bitcoin

In January 2009, the first viable digital currency called Bitcoin was born on the internet. The idea was to develop a currency that has no need for traditional banks and does not have any central authority. Bitcoin does not have any governance and oversight. It is managed through the consensus of all users of the currency. The first mention of Bitcoin, a new form of peer-to-peer electronic cash, was in a paper written by Satoshi Nakamoto in 2008. However, this is not a real person. Speculation is that it was a group of writers who used a pseudonym [3].

Bitcoin created a form of digital currency that could pass from an individual to another individual without any other financial intermediary. The idea behind Bitcoin was that there is no need to trust a third party. We can verify the transactions instead. The transactions are irreversible, blocks keep the exact order of transactions and the computers are decentralized, therefore the whole payment system is robust. Everyone can see Bitcoin’s existence in history of a transaction, but the people involved in the transaction and on the making and receiving each individual transaction only know the contents. As a currency, Bitcoin does not exist in a physical form and it is purely digital and lives on the internet.

For anything to have value it must be relatively rare and also be accepted by public as a form of payment. All the valuable metals and stones are rare and expensive to mine. The maximum number of Bitcoin that can be mined is 21 million. The value of Bitcoin is evaluated based on its demand. Users must have an electronic wallet to spend and receive Bitcoin. A transaction that is initiated from the wallet is added to a queue, validated by all the other Bitcoin users, and, if approved, added as a block to the distributed ledger the blockchain [3]. Bitcoin is a Peer-to-peer payment system. Using blockchain technology, Bitcoin replaces the third party trust with verification. Transactions are irreversible and Nakamoto argues that irreversibility would protect sellers from fraud. As long as the majority of the control is in the hand of honest participants (nodes), it is safe from the
attackers and hackers. Bitcoin uses the SHA-256 hash algorithm to generate verifiably random numbers in a way that requires a predictable amount of CPU effort. Generating a SHA-256 hash with a value less than the current target solves a block and wins coins [4].

2.1.2 Blockchain

Blockchain or distributed ledger technology, DLT, is a new technology to store and manage data across the Internet and other computing networks. It was created as a result of the introduction of the Bitcoin cryptocurrency. Today, the application of blockchain and its potential far exceed its genesis in Bitcoin. It supports not just digital money and trusted data movement and storage, but the exchange of value, an Internet of value. [3]

The terminology blockchain reflects the logic of the mechanism. New transactions in a network are bundled into blocks, which are added to existing blocks - forming a chain with cryptographic signatures. These signatures security link the blocks to each other [6]. Since the cryptographic signature depends on the chain of all previous blocks, changing an existing block in the chain would invalidate all the following blocks in the eyes of the rest of the network [7]. As a result, when a new transaction is proposed no participant would accept transactions coming from a modified version of this chain: this constitutes blockchains groundbreaking fraud-detection system [8].

Blockchain is peer-to-peer network architecture. If one peer breaks down, the data is reachable through other peers (Figure 1). All participants are equal in their roles on the network. Blockchain has a flat topology and all participants are the same. Blockchain does not have hierarchy. Decentralization, transparency and immutability are the main three characteristics of blockchain.
Blockchain is a new and innovative type of database. It is not like other kinds of databases that are installed on one central server. The blockchain database is installed on individual computers. The participants who use the database use these computers. Same identical database is installed on individual computers who used the database; and that is why it is called distributed database. These participants are called nodes. All the nodes must agree every time a new entry is created on the database and consensus must be reached. For example, if we use the metaphor of payment, if a person attempts to make a payment from one user to another and does not have sufficient funds in their account the blockchain participants will not permit the transaction. In addition, there are participants in the network called miners who contribute processing power to the network to solve mathematical problems in exchange for a reward. That is used to ensure that only valid new transactions may be added to [3]. Blockchain also enables new computing platforms such as Ethereum, Tezos and Rootstock. In Figure 2 – is shown.

![Blockchain Transactions](image)

Figure 2 – Blockchain Transactions [51]

The other negative point of traditional database is the central authority. In many cases that is exactly what is expected from a system; an e-commerce website is an example of that. Having this central power has the flaw of single point of failure. If the central authority is compromised, the database can be compromised as well. In most cases, any organization that creates a database has the right to that database because a central
authority holds the control. That organization is in charge of deciding who has access and what type of access they have, what is stored in them, what and when data gets deleted, and what is backed and archived.

In most cases, people remain the final arbiter of the validity of a transaction. We see this in contract work. A contract between two entities completed over the Internet still requires one or more central authorities to validate data. For example, with a mortgage, banks must validate savings and approve loans. Title companies must validate properties and legal professionals must validate signatures and other contractual requirements. [3] Each one of these central authorities has unique power that levies considerable overhead in a mortgage transaction. The transactions in the varied databases all take time to process, cost money, are vulnerable to hacking, provide limited participation from those involved, require special skills and can be error prone. Up until now we've generally been okay with this [3]. Blockchain is the solution all the problems mentioned above.

Blockchain database is installed on every computer that is used by the people that use the database. We call them nodes. A copy of the same database is installed on every node. This way the database is spread out among several nodes. There are no database servers. When a transaction happens in the distributed database. The transaction happens in one of the nodes then the transaction is transmitted to every identical database on the distributed network. All the nodes agree that this change can be done and the transaction is completed. All the distributed databases know the rules and when a change can be done. This is equivalent of consensus-based permission since all the nodes on the network have to agree on a change before it can happen.

Blockchain has chain of blocks, which have identical information on all of the databases. The blockchain database will not allow hacks because each block in the database relies on the previous block. Blocks are only added and never get deleted. Every change in the database just creates a new block not remove or edit, the previous block. Since blockchain is an immutable database, it is called characteristic immutability. Blockchain keeps tracks of changes and because of that sometimes it is called distributed ledger. In
order for a hacker to successfully hack the blockchain database, it needs to change the
data on all the nodes on the blockchain, which can be as high as thousands. A hacker
would need to fake all the processing and electrical power of the miners to fraudulently
add a block to the blockchain. This is the only way to get the authority to change the
database. Blockchain database is installed on every node. Every single node must verify
every transaction in the database. The system is secure as long as honest nodes
collectively control more CPU power than any cooperating group of attacker nodes [5].

Not only can the blockchain transfer and store money, but it can also replace all processes
and business models, which rely on charging a small fee for a transaction, or any other
transaction between two parties. Even recent entrants like Uber and AirBnB are
threatened by blockchain technology. All that is required is to encode the transactional
information for a car ride or an overnight stay, and again you have a perfectly safe way
that disrupts the business model of the companies, which have just begun to challenge the
traditional economy. We are not just cutting out the fee-processing middleman; we are
also eliminating the need for the matchmaking platform [9]. Smart contracts, the
sharing economy, crowd funding, governance, supply chain auditing, file storage,
prediction markets, protection of intellectual property, internet of things,
neighborhood micro grids, identity management, anti money laundry, know your
customer, data management, land title registration, and stock trading are some of
the examples of applications of blockchain.

2.1.3 Smart Contracts
A smart contract is a code or an automated process that handles the transaction in
between two parties without a third party. Contracts are very much like classes in object-
oriented languages. Contracts may have state variables, functions, events, and struct
types. Inside the contract we may have modifiers. Once the smart contract governing
the organization is deployed on the blockchain network, it become independent of
its developers and cannot be influenced by any outside entity. Its rules, financial
records and transactions are controlled and maintained by the blockchain and therefore eliminate the need of a middleman. [44]

2.2 Technical Structure

2.2.1 Cryptography

Cryptography is a Greek word, and it means secret writing. Using this technique we are able to send messages in cryptic way, and nobody will be able to edit it. This process is called encryption. The message will be decrypted at the final destination. Cryptography involves the use of code and protocols to establish secure communications [10].

2.2.2 Public and Private Keys

Public Key Infrastructure or PKI is one form of cryptography. It requires two keys, a public and a private key. Public key is visible to everyone. Private key is only visible by the authorized user. Since blockchain technology aims to provide a distributed and unalterable ledger of information, it has qualities considered highly suitable for the storage and management of public keys [11]. Private key is a random mixture of numbers and letters thoroughly. When a private key is generated its pair is also created which is the public key at the same time. Creating public key from private key is easy, however the opposite is extremely difficult. It is almost impossible to figure out the private key from public key. When an electronic message is sent out, it’s public key is sent out with it, and it is signed with a private key. The recipient has the public key. The authentication of the message can be verified by checking that the sender created the signature with the private key pair. The recipient opens the message with the private key. Unlocking the message without the private key is impossible.

2.2.3 Nonce and Hash Function

A nonce is a number that is used once and for a specific purpose. It will not be used again. There is always a chance that data entered in the database may have the same identifier. Adding a nonce to identifiers will make it extra unique. Nonce is essential for
adding blocks to the blockchain. The miner will mine the block to find. The valid nonce and the data create a specific hash.

Hash is a fingerprint of a digital data. It is data of a fixed-size. A hash function is a mathematical process that takes data of any size. Hash function will perform on it, and returns a hash. A hash algorithm turns an arbitrarily large amount of data into a fixed-length hash. The same hash will always result from the same data, but modifying the data by even one bit will completely change the hash. Like all computer data, hashes are large numbers, and are usually written as hexadecimal [4]. It is difficult to find another hash value for the same string.

No matter the size of the message, hash output size is always the same size, as shown in table 1. One thing to notice is that the hash created is always the same for the same input. Hashing is one directional, therefore can be use in encryption. Hashing can keep the database small if only the hash part and not the original data are kept in the database. Only the lightweight digital footprint of the hash needs to be stored in the database since the hash is mathematically associated with the original picture and data. Another reason for hashing is that it creates an almost unique identifier. A hash with 64 characters of numbers and letters creates an astronomical number of combinations and there's little chance of duplication [3].

<table>
<thead>
<tr>
<th>INPUT</th>
<th>HASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi</td>
<td>54F8E907639DA4C67213F5628FD980EB54390CBF3648A038347D23DF09384BA</td>
</tr>
<tr>
<td>This is a ChatApp with blockchain approach</td>
<td>9A0562DF09854F8FD9E907639F80EB543380C4BABF3648347D233DA4C67218</td>
</tr>
</tbody>
</table>

Table 1 –Unique Fixed Size Hash Output

In the blockchain, hashes are used as identifiers for blocks, transactions, and addresses. The hashing algorithm used in the blockchain is called SHA-256. SHA, which stands for Secure Hash Algorithm, generates an almost unique fixed-size 256-bit hash. [3]
2.2.4 Blocks

Blocks are files that permanently record and hold transaction data. Blocks are linear sequences. Miners constantly process new blocks. These blocks are added to end of the chain of the blockchain. As more blocks are added to the blockchain, it gets harder to change, edit or remove the blocks that were added earlier and are deeper in the chain. This is how the transactions become irreversible. Each block in blockchain contains ledgers and transaction data. The block data is hashed by cryptographic hash functions.

2.2.5 Block Structure

A block on the blockchain consists of block number, data (transactions and ledger), nonce, hash of the current block, and the hash of previous block. Previous hash is passed to the cryptic hash function and gives us the hash of the current block. The hash of this block will be used as a previous hash of the next block (Figure 4).
2.2.6 Forming a Chain

The first block in the blockchain is a *Genesis* Block. After the genesis block, each block will be validated and then the validated block will be added to the blockchain. Each block contains a pointer to the previous hash. A hash pointer is similar to a pointer. Hash pointer contains the address of the previous block and the hash of the data inside the previous block. The chain of the blocks contains the ledger of all the systems.

![Blockchain Blocks](image)

**Figure 5 – How Blocks in the Blockchain Form [54]**

2.2.7 Transactions

Traditional blockchain application like Bitcoin, consist of transactions that represent an exchange of money between two entities (or users). Each valid transaction is recorded in a block, which can contain multiple transactions, for efficiency. Immutability is achieved by leveraging strong cryptographic properties such as hashing. [13] An electronic coin is a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership. The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint,
that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank [5].

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced, and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received [5]. The solution is a timestamp server.

2.2.8 Timestamp Server

Timestamp server takes the hash of a block of items and time stamped it. It will widely publish the hash. The timestamp is a way of proving that the data must have existed at the time. Each timestamp includes the previous timestamp in its hash. This will form a chain. Next timestamp will reinforce the ones before it as shown in figure 7.
2.2.9 Blockchain Consensus Protocols

In order for the blockchains to function globally, a practical, efficient and secure consensus algorithm is required for a shared public ledger. All blockchain-based applications use distributed consensus algorithm. Research on consensus mechanisms has proposed a large range of systems, from proof-of-work to proof-of-stake systems to the hybrid systems in between [14].

2.2.9.1 Proof of work

The mining proof-of-work (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the determination of some token value. It is utilized to enforce the blockchain security by giving meaning and credence to the notion of difficulty (and, by extension, total difficulty). However, since mining new blocks comes with an attached reward, the proof-of-work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism [30]. Proof of work is accessible to as many nodes as possible. There is minimum need for specialized or uncommon hardware. The goal is to make the mining from electricity at the same rate for any node in the globe. A proof of work is used to implement a distributed timestamp server on a peer-to-peer basis. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash [5].
The proof-of-work is established by incrementing a nonce in the block. This process is continued till a value that gives the hash of the block the required zero bit. A computer is going to solve the mathematical problems. The processing capability is applied in order to solve it. The harder it gets to solve the problem, the more computing power, is consumed. Once the proof-of-work is established, the block cannot be changed without redoing the work. The act of mining and creating a winning hash with success will implement the proof of work. It also solves the big problem of double spending. Mining Bitcoin takes about 10 minutes. This is the time required to solve a given hash and generate proof of work, no block will be accepted as authentic, until the mining is complete. The miners provide the proof of work by completing the math problem and therefore give permission for block to be added to the blockchain. Miners compete to solve the problems for each block transaction. The winning miner is provided with a specific amount of coin and the transaction fee. The protocol is fair in the sense that a miner with \( p \) fraction of the total computational power can win the reward and create a block with the probability \( p \). An attacker is required to solve the same tasks as the rest of the network; i.e., an attack on the network will only be successful if the attacker can bring to bear significant computational resources [14]. Proof of work requires enormous amount of computational energy, which will create scalability issues. Also most of the mining can be centralized in the areas with cheap electricity, and that is why other alternative consensuses have been developed since Bitcoin.

### 2.2.9.2 Proof of Stake

Proof of stake is the most common alternative to proof of work. The validators (stakeholders) invest in the coins instead of investing in computational equipment and energy. All the coins of the system exist on day one and no mining is required. The chance of the validators to be picked for next block depends on the fraction of the coins they own in the system. The decentralized consensus mechanism has advanced security measures by eliminating the need for a trusted third party in any interaction [49]. One possible decentralized ledger implementation with security not based on expensive computations relies on proof of stake algorithms. The idea behind proof of stake is simple: instead of mining power, the probability to create a block and receive the
associated reward is proportional to a user’s ownership stake in the system. An individual stakeholder who has $p$ fraction of the total number of coins in circulation creates a new block with $p$ probability [14]. The nodes with the highest number of stakes in the network have the most interest to keep a secure network. If the network has successfully attacked the value of the cryptocurrency would drop due to the attacks. The attackers have to acquire most of the network (currency) in order to implement a successful attack. Doing so is very expensive and discourages the attackers.

### 2.2.9.3 Proof of Capacity

Proof of capacity is also known as proof of space. The nodes on the network prove that they have enough *storage* to solve a computational problem. Proof of capacity algorithm targets computational problems such as hard-to-pebble graphs that need large amount of memory storage to solve the problem. [15]

### 2.2.10 Network

In a network, the new transactions are broadcasted to all the nodes. The new transactions are collected into a block by the nodes. Every node will work really hard to solve the problem, therefore provide the proof-of-work for its block. The first node that finds the proof-of-work will broadcast it to all the nodes. All the other nodes will only approve the new block only if all the transactions in it are valid, also not already spent. Once the nodes accept the new block, it will be added to the chain.

The longest chain is the correct chain, and accepted by the nodes. This is the chain that other nodes will add blocks to it. There are times that the two nodes broadcast the next block at the same time. As the result the blocks will not be the same. Some of the participating nodes will receive one or another first. The nodes will work on the first block they received. The other branch will be saved in case it becomes long. If one of the branches become longer and longer, all the nodes will move on to the longer node, and abandon the shorter one. The tie will be broken. The new transactions do not necessarily be broadcast to all modes. Block broadcasts are also ok with dropped messages. The
node that did not receive a block will request it when it receives the next block. That is when the node realizes that it missed it. Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle tree, with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored [5].

![Figure 8 – Structure of a Blockchain [5]](image)

### 2.2.11 Distributed Ledger

A distributed ledger is an asset database, and a form of decentralized database. A massive network of participating nodes hold distributed ledger. These nodes have a full copy of the ledger. The nodes can be virtual or real. The distributed ledger runs on the blockchain technology.

New transactions in a network are bundled into *blocks*, which are added to the existing blocks, therefore forming a *chain* with cryptographic signatures. These signatures securely link the blocks to each other. Since the cryptographic signature depends on the chain of all previous blocks, changing an existing block in the chain would invalidate all the following blocks in the eyes of the rest of the network. As a result, when a new transaction is proposed no participant would accept transactions coming from a modified version of this chain: this constitutes blockchains groundbreaking fraud-detection system [16].
2.2.12 The Disadvantages

The disadvantages of Decentralized applications revolve around the fact that fixing bugs in decentralized application can be very difficult due to a massive number of users having to update the software of the node and the fact that when a smart contract is deployed on the blockchain network it cannot be modified. Verification can be an issue with some applications but not all. Verification of a user’s identity due to the lack of central authority can become an issue during the development of decentralized application. Being deployed on the blockchain means that the decentralized application cannot be taken down from a network unless the entire network is terminated. The decentralized applications can be very difficult to develop due to the protocols being more complex. Later additions of features are very difficult to add on while maintaining scalability. [44]
Chapter 3 – Ethereum Blockchain

Ethereum is a blockchain that any one can download and run the processing software. It is an internet service platform that will provide a guaranteed environment for computation. Ethereum is a programmable blockchain and is a peer-to-peer network. It provides a set of useful integral features for the developers. Ethereum is a decentralized Heroku.

Ethereum features:

- User authentication, via seamless integration of cryptographic signatures
- Fully customizable payment logic without any reliance on third parties
- Resistant to distributed denial of service (ddos) resistant up-time
- Unlimited storage
- Ultimate interoperability, everything in the Ethereum ecosystem can trivially interact with everything else, from reputation to custom currencies
- Server free zone, the whole application can be deployed on the blockchain meaning no need for setting up or maintaining servers; users pay for the cost of using the service. [23]

3.1 Ethereum Philosophy

Simplicity is one of the key design philosophies in the Ethereum protocol. An average programmer should be able to learn and program in the Ethereum environment. Other core philosophy is universality and this means that Ethereum isn't going to have features, it's going to be a Turing complete scripting language, which anyone can use to develop anything, but it's not going to have specific apps built into it, that's up to the programmer. Insofar as possible, they will make it so that anyone can use this to do anything, but they're not going to focus on developing domain specific application. [20]

Ethereum is modular and therefore capable of using different elements of Ethereum network as necessary. The idea is that the Ethereum development environment should
always be maximized to develop the whole ecosystem. Ethereum is designed to be agile so as time evolves the platform evolves as well. Ethereum has a strong policy of non-discrimination and non-censorship. Ethereum is open to anyone.

3.2 Ethereum and Bitcoin Comparison

Bitcoin was mainly designed to be a store value. It does have some simple programming functionality. Ethereum was written to handle programming and store data for programming. It offers a full Turing complete programming environment.

![Figure 9 - Merkle trees in Bitcoin](image)

There are some slight differences in implementation. One main difference is Bitcoin uses a Merkle Tree structure. Merkle tree is essentially a way of compiling each of the transaction blocks and then including those blocks, moves up into a tree of blocks and encrypting a hash key for each one. [27]. A tree constructed by hashing paired data (the leaves), then pairing and hashing the results until a single hash remains, the Merkle root. In Bitcoin, the leaves are almost always transactions from a single block [48]. Ethereum has a similar type of encryption key, however it uses what's called a Patricia Tree, which includes not just the transaction, but also state information and receipt information. Since Bitcoin doesn’t have general state information it does not have the essential key information like timestamp. Ethereum handles state management and not just
transactions. Ethereum has a deeper programming environment and a more simplified
programming language and it also manages stakes.

Ethereum nodes do not store the entire blockchain network and they only store the state.
Blockchain history can be validated through parsing of the encrypted tree. Any wrong
item is going to change the hash for every branch above it. Because of fewer heavy nodes
of transaction, there is a security concern. It is easier to fake nodes with light nodes. That
is why Ethereum requires every minor to run a full node [27]. As of 2018 Ethereum
moved on to proof of stake to solve this issue. No more mining is required and instead
anyone who owns a certain amount of Ether is able to stake Ether and their processing
power will be used to solve the problems. Proof of stake uses less energy.

3.3 Gas and Payment
One of the key differences between Ethereum and other programming environments is
that calling write operations cost gas, which is Ethereum. In order to avoid issues of
network abuse and to side step the inevitable questions stemming from Turing
completeness, all programmable computation in Ethereum is subject to fees. The fee
schedule is specified in units of gas. Thus any given fragment of programmable
computation (this includes creating contracts, making message calls, utilizing and
accessing account storage and executing operations on the virtual machine) has a
universally agreed cost in terms of gas. [30] Gas is in the form of ether, and it's usually
broken down to the smallest decimal. In table 2 the denominations of unit in ether has
been shown.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Wei Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>wei</td>
<td>1 wei</td>
<td>1</td>
</tr>
<tr>
<td>kwei (babbage)</td>
<td>1e3 wei</td>
<td>1,000</td>
</tr>
<tr>
<td>Mwei (lovelace)</td>
<td>1e6 wei</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Gwei (shannon)</td>
<td>1e9 wei</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>microether (szabo)</td>
<td>1e12 wei</td>
<td>1,000,000,000,000</td>
</tr>
<tr>
<td>milliether 9FINNEY)</td>
<td>1e15 WEI</td>
<td>1,000,000,000,000,000</td>
</tr>
<tr>
<td>ether</td>
<td>1e18</td>
<td>1,000,000,000,000,000,000</td>
</tr>
</tbody>
</table>

Table 2 - Denominations of unit in ether [27]

In general, Ether used to purchase gas that is not re-funded is delivered to the beneficiary
address, the address of an account typically under the control of the miner. Transactors are free to specify any gas price that they wish; however miners are free to ignore transactions as they choose. A higher gas price on a transaction will therefore cost the sender more in terms of Ether and deliver a greater value to the miner and thus will more likely be selected for inclusion by more miners. Miners, in general, will choose to advertise the minimum gas price for which they will execute transactions and transactors will be free to canvas these prices in determining what gas price to offer. Since there will be a (weighted) distribution of minimum acceptable gas prices, transactors will necessarily have a trade-off to make between lowering the gas price and maximizing the chance that their transaction will be mined in a timely manner [30]. Computing fees were introduced to discourage components requiring high computing power, such as smart contracts with infinite loops, which are a heavy burden for the network [22].

3.4 Ethereum Virtual Machine (EVM)

Unlike Bitcoin that has one specific use, Ethereum is a programmable blockchain. Ethereum offers a peer-to-peer network so transactions are safe and proven across the network. EVM or Ethereum Virtual Machine is at the center of this system and it is where all the smart contracts are executed [20]. EVM is completely isolated and has no access to file systems or processors. Since each node in the system runs the Ethereum virtual machine, it contains the consensus across the network. Ethereum is best suited for applications that automate direct interaction between peers or facilitate coordinated group action across a network. Ethereum virtual machine executes the transactions. Each node in the chain runs EVM. EVM is the app server that's going to handle processing smart contracts and the programming functionality. There are other services that can be available through these nodes including Swarm for hosting files and Whisper for messaging and DApps.

Every Ethereum node runs the EVM, therefore data is synced and consensus of the blockchain is maintained and as the result:

- Immutable
- No censorship
- Zero downtime due to the number of nodes involved
- High fault tolerance

3.5 **Ethereum Smart Contracts**

Smart Contracts are the strongest feature of Ethereum. Smart contracts are classes in which functions can be called on externally. Data will be stored in the blockchain. Smart contracts are programs, which control the behavior of accounts within the Ethereum state. When an Ethereum contract is compiled, it is compiled into bytecode. The bytecode is stored in the Ethereum network on the blockchain. Since the blockchain is immutable, after it is added to the blockchain no more changes can be done to it. Whenever an address is created and the smart contract is compiled, by recompiling, it's going to then have to have a new address. Every contract has it's own address. The original idea for Ethereum was essentially to do some coding that enables the user to put money in holding bin, and this holding bin would require certain parameters to be met. Once those parameters will be met, it could then release the money to someone else or to a number of different people or return the money to the original individual who sent it. However, smart contracts have evolved considerably. They've evolved to encompass all kinds of code that's executed on a blockchain node. This means they're behaving much like a normal programming class or an executable service. So each smart contract is essentially a grouping of functions and state variables that reside on the blockchain and also can store data and interact with that blockchain opening unlimited possibilities [27].

![Ethereum Smart Contract](image)

**Figure 10 - Ethereum Smart Contract [40]**

Smart contracts are one of the key reasons that blockchain and distributed ledger technology is advancing beyond the use case of storing value or handling financial transactions.
3.6 Decentralized Apps or DApps

Decentralized Apps or DApps are applications that run on the blockchain. The blockchain enables apps to run without having to have a central location. Everything is hosted in Ethereum network. Users can access that app from any node that can access the blockchain, which opens up a vast global network.

- Reduces fee since there is no centralized authority
- DDoS Free
- Decreases reliance on a central network and resources
- No need for personal trust

The way smart contract works is that the user puts a request in. The request goes on the blockchain and anyone who is listening for that application. The fund will be kept and when the transaction is complete the fund will be released. Files are hosted in Ethereum through an app called Swarm that exists on many of Ethereum nodes. Smart contracts are in charge of any interaction with applications. The data will be stored in the blockchain. Swarm can host any type of file such as HTML, JavaScript, etc. Different parts of a DApp are shown in figure 11.

![Figure 11 - Instruction of a DApp [20]](image)

3.7 Transaction Execution

The most complicated part of Ethereum protocol is executing the transaction. Each received transaction is validated before propagation it to other nodes. Each transaction signature is validated. The transaction nonce is verified. The sender's account balance has to have at least the cost of the transaction. There are a number of intrinsic parameters used when creating an account: sender, original transactor, available gas, gas price,
endowment together with an arbitrary length byte array, the initialization EVM code and finally the present depth of the message-call/contract-creation stack. [30]

3.8 Interacting with Ethereum Network

Developers can develop programs, which execute on EVM. These programs are written in a new JavaScript programming language called Solidity. Programs written in Solidity are compiled to EVM bytecode. The EVM acts as the runtime environment for these programs. These programs are completely isolated and don’t have access to the file system, network or any other processes running on the same machine. [38] In figure 12 and figure 13 the way a website talks to Ethereum blockchain with web3 has shown. A website talks to web3 with JASON-RPC. JSON-RPC is a remote procedure call protocol encoded in JSON. JSON-RPC is used to make a special request to a node on the network.

![Figure 12 - Interacting with Ethereum Network](image)

![Figure 13 - Ethereum and App Integration](image)
3.9 The Competition

For this thesis WhatsApp has been chosen as the competition since it is one of the biggest name in the world for messaging application. WhatsApp uses the Erlang programming language which was most appreciated for its performance reasons, speed and scalability. Contus Fly is built on Erlang where the garbage collector allows updating the code instantly for different functional optimization. The programming language implemented in Contus Fly helps to have a high volume of short messages processing in very low latency. [45]

WhatsApp is best known for: Voice Over Internet Protocol, voice chats, group calls, end-to-end chat encryption, cloud service sync, multimedia support, geo location integration, calendar synchronization and multi platform chatting. It is entirely free and does have mobile advertising. WhatsApp does not have an open source code, one cannot trust that WhatsApp is totally private and not expect to get data mined for valuable information – information that could easily be sold to advertisers.
Chapter 4 – Tools of Implementation on Blockchain

4.1 Structure of Smart Contracts

In most instances, smart contracts are written in Solidity, but there are other options available such as Lisk, Bamboo and Viper. A basic coding structure of smart contract in Solidity:

- **Version pragma** will tell the compiler what version of the Solidity to use to compile the contract. By using the right version the code compiles at it was originally intended.
  
  ```solidity
  pragma solidity ^version; //version of the required compiler
  ```

- The **Contract** can inherit properties of other contracts. Contract is the highest element of the code.

- **State variables** will be permanently stored in smart contract. State variables are similar to class variables. State variables are defined by their type and level of visibility. State variables cannot be external.

- **Functions** (similar to functions in object oriented programming) are the codes that execute the actions of the contract. Functions can be called as many times as necessary. Default Functions are public but they can be defined public or private.

- **Function Modifiers** modify semantics of functions. They can be seen as helpers to avoid repetition in methods' conditions (in terms of access or execution) [16]. Modifiers are conditioned to a function or conditions before we run the function. They are used to amend the semantics of a function in a declarative way. These can be used to automatically check a condition prior and/or after the execution of a function [21]. A function can have few function modifiers.

- **Events** are the logs created by the contract. Events are visible for everyone to see on the blockchain. Events expect **address from** and **address to**

- **Mapping** is used to structure value types, such as booleans, integers, addresses, and structs. It consists of two main parts: a _KeyType and a _ValueType; they appear in the following syntax [25]:

```solidity
mapping (_KeyType => _ValueType) mapName;
mapping (address => uint) public balances;
mapping (address => mapping (address => ConnectionType)) connections;
mapping (address => Member) public members;
```
• **Block** is the block of information within that Ethereum. *Number, difficulty, coinbase* (current block miner’s address, which is a function and return address payable), *gaslimit* (current block’s gas limit), and *timestamp* are members of each block.

• **Msg** which is the data in the receiving message. *Sender, value* (number of wei sent with the message), *data* (complete calldata) and *sig* (first four bytes of the calldata also known as function identifier) are the members of *msg*. The values of all members of *msg*, including *msg.sender* and *msg.value* can change for every external function call. This includes calls to library functions [26].

Ethereum is a platform to build DAPPs and Solidity is the language to program smart contracts. Solidity is a contract-oriented, high-level language for implementing smart contracts. Solidity supports inheritance, complicated user-defined types, libraries, and is similar to object-oriented programming languages, such as C++, Python or JavaScript [17]. Solidity is written to be compatible with the Ethereum virtual machine, and it is statically typed. Solidity can be developed on a Mac, Windows, or Linux. It has many plugins and extensions for IDEs and editors, like Visual Studio, VSCode, Atom, and more. Solidity enables almost limitless possibilities for configuring decentralized applications. In fact, this blockchain processing platform runs a protocol enabling the representation of any computable function. [18] Solidity's code uses syntax similar to JavaScript statements and serves as the base for smart contract software programming. [19] Solidity’s specificities remain mainly in the structure of smart contracts and in interactions made with the network, such as transactions. For instance, one common specificity in smart contract coding is the term *throw* which means to block the execution of the corresponding method. Any entity on the network, such as members or smart contracts, interacts through Ethereum Addresses that are defined by 20 byte values. A proper DApp will consist of core and side contracts interacting between each other [16]. *Address* in Solidity refers to wallet or wallet address. Address is a sequence of numbers and characters.
Block’s property is accessible from within any smart contract. `block.coin` is the current block miner’s address. There are a few globally available variables and units in Solidity. The recommended method of sending funds after an effect is using the withdrawal pattern. Although the most intuitive method of sending Ether, as a result of an effect, is a direct transfer call, this is not recommended as it introduces a potential security risk [2]. Units that we can use inside a contract are Ether, Szabo, Finney, and Wei. These units or variables can be used inside the code, but new variables with these names cannot be created. [20] With Solidity you can create contracts for uses such as voting, crowd funding, escrows, blind auctions, safe remote purchases, micropayment channel, payment channel and multi-signature wallets.

4.2 Dependencies and Developer Setup

4.2.1 Node.js and Node Package Manager

Node.js provides the tools to build JavaScript applications that run outside the scope of the browser. Node.js is single threaded and all the users are sharing the same thread. Events are raised and recorded in the event queue, and then handled in the order that were raised [36]. Node.js is asynchronous, non-blocking and can handle more than one event at a time, therefore it is fast. Node package manager or NPM is the first installed dependency. NPM is a way to reuse codes from other developers and share codes with them as well. The bits of reusable codes are called packages and sometimes modules. It is a module-based approach. A package is a directory with one or more files in it that also has a file called `package.json` with some metadata about that package. NPM comes with Node.js. The command “npm install” will install all the dependencies for the `package.json` and creates the `node_modules` folder.

4.2.2 React

React is one of the most popular JavaScript libraries for creating user interfaces, and is a vital skill for many front-end and full stack developers [34]. React creates reusable components, and these components display data as it changes over time. React makes it
painless to create interactive UIs. It will efficiently update and render just the right components when the data changes [24]. React manages all of the view and template logic in the same file. React page is a collection of components. In ChatApp project create-react-app is also used.

### 4.2.3 Web3

The Web3 class is a wrapper to house all Ethereum related modules [29]. Web3 is a JavaScript library that allows the frontend application to talk to the blockchain and interact with DApps. Web3 is an open source standard project to interact from the client side applications to the Ethereum network. One of the challenges with developing the client side of the decentralized application is finding a way to effectively wiring up the frontend application with the blockchain, web3 helps us with this challenge. Web3.eth is an important package that allows the user to interact directly with the blockchain. Web3.eth gives the required tools to the user to access Coinbase of a particular block, look at transactions, look at different accounts and check their balances and much more.

Web3 is used for:
- Performing transactions
- Managing user accounts
- Smart contract Interaction

To get web3.js into the project

```
$ npm install web3
$ node
> var Web3 = require('web3')
```

Create a web3 instance and setting a provider [28]:

```javascript
if (typeof web3 !== 'undefined') {
    web3 = new Web3(web3.currentProvider);
} else {
    // set the provider you want from Web3.providers
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
}
```

In this project web3 was used to interact with Ethereum blockchain by calling methods, and usage of objects, utils and some other packages.
4.2.4 Application Binary Interface (ABI)

ABI is a JSON encoding of a smart contract. Etherscan website has a copy of ABI which is up for grab. The `web3-eth-abi` package allows decoding and encoding parameters from an application binary interface. This will be used for calling functions of a deployed smart-contract [30].

4.2.5 EthereumJS

Main focus of *EthereumJS* is to provide high-quality and robust implementations of core Ethereum infrastructure technologies (virtual machine), protocols (devp2p) and data structures [52]. To install:

```bash
$ sudo npm install --g ethereumjs-testrpc
$ testrpc
```

The “testrpc” command will allow us to spawn a virtual blockchain in the memory and work in it without the need to install a real blockchain. This will provide 10 different accounts and their associated private keys and local host 8545 [46]. Http is our communication channel, then indicate the path which is on the local host 8545

```javascript
web3.eth.defaultAccount = web3.eth.account[0];
```

This will choose the first account of the 10 free accounts that testrpc provides. Method `eth.contract` is used to initiate contract on an address. It takes one parameter that refers to ABI or application binary interface. ABI will allow us to call functions and receive data from the smart contract. It is an interface.

```javascript
var chatAppContract = web3.eth.contract(abi);
```

Now we have the interface for interacting with the smart contract through `chatAppContract` variable. The last thing is to define the contract address.

```javascript
var chatApp = chatAppContract.at(contractAddress);
```

4.2.6 Truffle Framework

Smart contracts don’t get compiled on Ethereum network. They are already compiled bytecode when they exist in the blockchain. So we need to compile and migrate them and one way is using Truffle [26]. Truffle also has a test server. It provides the tools to test
the smart contracts and check for the bugs. By running Truffle a contract is initiated and it takes a deposit. Here are some uses of Truffle:

- Build decentralized applications on the Ethereum blockchain
- Tools to write smart contacts with the Solidity programming language
- Test the smart contracts
- Deploy smart contracts to the blockchain
- A place to develop client-side application

Migrations are JavaScript files that help deploy contracts to the Ethereum network. They will stage the deployment tasks.

```
$MacBook-Pro-mr$ testrpc
EthereumJS TestRPC v6.0.3 (ganache-core: 2.0.2)

Available Accounts
===========================================
(0) 0xa8e5f7f9b57c1001bd4beb89f134e2f52b08de08
(1) 0xf92dab9e9aa3ca9db5cc727c0aded7d0fa8b4787
(2) 0x1dd74c1ca7e18d79a0e7e223baa75ec4f7ba
(3) 0x341d5e86b2eb7c79cfe0f2ec9a79563ecb378
(4) 0xe9f9c873b7d2421653e7b879ccf64ff7d31353
(5) 0xdab00f9f42090e9dcf72b4020f617bfc9aef731
(6) 0xf5fada9de93079997f09c69d7f59611a22f4f92f
(7) 0x3e6b67dd578f11d0b9b0899eaa8860aaf2d
(8) 0x56e2ccbf3ed898181deb2b2ca72a283ea48381e
(9) 0x5788568479c5910b413a8522d9e5d5b6b88a296

Private Keys
===========================================
(0) a1d6d313e087ac6fc95e6d1b73e4e2534f1e8a29c539ec168606ca58a87af8
(1) 6b87e6c5d64e99a9d2d120511ba8f8c9dcd47b66dce0ae73f211fae9e90e36
(2) 1d8aef9fbf8173f4fedef5fed50e2287f95f72100e34581a5e43b67e3386b8
(3) 1bc6ca6f8a9536a4e7d82a253dd6be4c6c3f373ecb8a6f7229b2abf4a0b07
(4) 77a6e47b9fd36dd36cb5faec56ba3b3f24f9b8f5e2e98c1c799695c0d3ac724
(5) 9ed5db03a3a0c3a15ec4f444fd7808cc93b0999e26e9d3ac6ed6cbac32997
(6) 5be8db9e5c6e368f4b9674b8a58ddadebe081d2ea38ce7e6ec0f6e5a8f4db
(7) a9f7058c948a2472363781e2a572b405e4f5e80f0b13b31f9000080b70b0b7a
(8) f165f77fe7f0c4dd0e2bd33e2096f19fd2690dd0f1a3996dd08a9795f496f21c
(9) a4cc89087798131bb05d584768f3a3af010c9df7800c180cb71721160a0

HD Wallet
===========================================
Mnemonic: sound page aisle huge mesh effort surface fiction weather call staff win
Base HD Path: m/44'/60'/0'/0/0 (account_index)

Listening on localhost:8545

Figure 14 – Running Migrations Using Truffle

4.2.7 Ganache

Ganache is a local in-memory blockchain. Ganache gives 10 external accounts with addresses on the local Ethereum blockchain. Each account is preloaded with 100 fake Ether (Figure 15). Ganache works side by side with Truffle and provides a nice visual interface.
4.2.8 Metamask for Authentication and User Authorization

The next dependency is the Metamask extension for Google Chrome. Metamask is used to connect to blockchain. MetaMask is used for verification of the owner of a blockchain address, which can send and receive messages. Metamask is used to connect to the local Ethereum blockchain with the personal account, and interact with the smart contract. It enables the user to have a wallet that is built into the browser so Ethereum can be sent and it can be signed any time Ethereum has to be sent to the network.
Copy the mnemonic phrases and in MetaMask extension, create a custom RPC as shown in figure 17 and figure 18 and point it to http://localhost9545.

Figure 17 - Create custom RPC

Figure 18 – Reset the account

In Figure 19 MetaMask is setup and ready to test the DApp.

Figure 19 – The localhost:9545

It is possible to test the apps without having to sign them in local environment, but once the developer starts running in the test environment, the contracts have to be signed [26]. Brave Browser is an alternative to Chrome which has Metamask built in.
4.2.9 Go-Ethereum for Ethereum Client

One of the most popular EVMs is Go-Ethereum also known as Geth. Geth is used to run EVM and you can use it to run on a Testnet or the Mainnet. Go-Ethereum is one of the three original implementations (along with C++ and Python) of the Ethereum protocol. It is written in Go, fully open source [30]. Once the app is developed and tested the next step is to install the Ethereum client. Go-Ethereum will run a node of Ethereum as well as Ethereum environment on the user’s machine. The contract can be synced up to a testnet or mainnet. By connecting to these nets, the migration of the smart contracts to different networks will be possible. In order to install Go, the Homebrew package has to be installed. The package is updated and upgraded.

$ brew install geth

Unlike Truffle, brew does not have it’s own installed accounts. In order to get an account for testing:

$ geth account new

After setting up and account an address will be provided.

$ geth

This is how it will be connected to the Ethereum main network and syncing with it. Once geth is installed we are ready for the next step, which is the test network.

4.2.10 Rinkeby Test Network

Once the smart contract is tested locally it needs to be tested in a testnet and eventually mainnet. Geth is used for both. Depending on where geth is pointed to, it will test it on testnet or mainnet. Go-Ethereum supports connecting to a proof-of-authority based test network called Rinkeby. This network is lighter, more secure, but is only supported by Go-Ethereum [32]. Network ID value for Rinkeby is 4.

4.2.11 Swarm for Decentralized Hosting

Once the smart contract is deployed to the backend of the app has been distributed. At this time to access the DApp either we have to have the assets locally or through a web UI. Both of those methods are centralized and have their disadvantages. To solve this
problem Ethereum comes with an option to deploy the files on the blockchain using _Swarm_. Swarm comes with geth and already set up on the server. Like Ethereum itself, these networks use decentralization to ensure that no _one_ computer (and thus no one person or organization) has control of the data or application [42]. Files are hosted in Ethereum through Swarm, and it stores any type of file.
Chapter 5 – The Development and Implementation

The ChatApp has been developed on the base of last chapter's dependencies. ChatApp has a smart contract and a web interface. The smart contract will allow accessing the blockchain. The smart contract is where the data is stored. The user interface is where the encrypted messages are sent from one address to another. Only the sender and the receiver of the message can decrypt the message, however the encrypted messages will forever be on the Ethereum blockchain. ChatApp does not require a centralized server and only depends on the Ethereum network. In order to get in touch with other people, the user shares his public key with other users while keeping his private key secret. Messages are encrypted using an encryption key, which is the same key that will be used to decrypt the message. Only the sender and the receiver can generate the same encryption key. The sender generates the key from sender’s private key and receiver’s public key. The receiver will compute the key from his private key and the sender’s public key. The contract generates events that will be stored on the network. It does not store the messages or contacts. All the encryption and decryption will be done with web app and not through the contract. The messages and friends list will be aggregated based on history of events generated by the contract. ChatApp members and the relationship between members are the only information saved in the contract’s storage.

Basic structures of ChatApp are

- Joining by signing the private key (There is a transaction fee involved)
- Accept friend request (transaction fee)
- Add friends using their public key (transaction fee)
- Send messages to friends and receive messages from friends (transaction fee)
- Receive messages from friends (free)

5.1 JSON File

All the dependencies in the project will be specify in package.json file. A package.json file is a manifest that contains information about the app. It will easily distribute the
application code without having to worry about distributing all the dependencies as well [36].

$ npm init

package name: (chatapp) chatapp  
version: (1.0.0)  
description: A Decentralized Approach to Messaging, Using BlockChain Technology  
entry point: (index.js)  
test command: (mocha)  
git repository:  
keywords: decentralized app, ethereum app, messaging app  
author: Mahzad Stearns  
license: (ISC)

Npm also provides us a way to automate running, testing, debugging our applications or,  
really, running any Unix or DOS commands [36]. Scripts property is part of  
package.json. "Npm start" is used to start the web application. Scripts create mock data  
on the blockchain for manually testing the app.

"scripts": {  
  "start": "next start -p 8000",  
  "build": "next build",  
  "dev": "next dev -p 8000",  
  "test": "mocha"  
},

By adding these scripts now the file-system is the main API. Every .js file becomes a  
route that gets automatically processed and rendered and make server rendering and  
indexing of .pages. React will create components and use the React library. React-dom  
takes those components and places them in the dom, to render them to the page.

Package.json will keep track of all the dependencies. The save flag will add the  
dependency to package.json.

$ npm install react --save

"dependencies": {  
  "create-react-app": "^2.1.8",  
  "react": "^16.8.6"  
},
The dependencies that are used for ChatApp are: `solc`, `web3`, `react`, `react-dom`, `create-react-app`, `ethereumjs-tx`, `ethereumjs-wallet`, `truffle-hdwallet-provider`, `ganache-cli`, `next`, `mocha` and `flux`.

5.2 The Smart Contract

Communication between contracts takes more gas than method calls within a single contract. Also, every separately deployed contract in the system increases the overall attack surface. When the contracts are compiled, only that final contract needs to be deployed to the blockchain. Its bytecode contains the whole inheritance chain, structs, state variables, methods and mappings [42]. Solidity contracts support multiple inheritances.

*Address:* is the address on the Ethereum network. There is one address for sender and one address for the receiver. *Message* is a special object that comes in called message and the message is going to contain some data including the sender, which will be the address of who’s sending the money, and also the value, which will be the value of what's being sent. The `msg.sender` variable is always present upon execution of a contract’s public methods (including the constructor) and could represent a wallet address or the address of another smart contract. [42] [12]

```solidity
enum ConnectionType {NotFriend, Waiting, Friend}

mapping (address => mapping (address => ConnectionType)) connections;
mapping (address => Member) public members;
```

- The Join Function

```solidity
function join(bytes32 pkr, bytes32 pkl) public {
    require(members[msg.sender].isJoined == false);
    Member memory addMember = Member(pkl, pkr, "", ",", 0, true);
    members[msg.sender] = addMember;
}
```
• Add Friend Function
The function `addFriend` will add friends by their addresses. One has to be joined to be able to add friends.

```solidity
function addFriend(address adr) public isJoined {
    require(connections[msg.sender][adr] == ConnectionType.NotFriend);
    require(connections[adr][msg.sender] == ConnectionType.NotFriend);
    connections[msg.sender][adr] = ConnectionType.Waiting;
    emit addFriendEvent(msg.sender, adr);
}
```

• Accept Friends Function

```solidity
function acceptFriends(address adr) public isJoined {
    require(connections[adr][msg.sender] == ConnectionType.Waiting);
    connections[msg.sender][adr] = ConnectionType.Connected;
    connections[adr][msg.sender] = ConnectionType.Connected;
    emit acceptFriendEvent(msg.sender, adr);
}
```

• Send Message Function

```solidity
function sendMsg(bytes message, address to, bytes32 encryption) public isJoined {
    require(connections[to][msg.sender] == ConnectionType.Friend);
    emit sendMsgEvent(message, msg.sender, to, encryption);
}
```

• Encryption
In a blockchain system, any key holder can use their private key to sign a piece of data. This results in a signature. Whoever obtains the signature can use this to:
  • Recover the public key (account address) of the Author
  • Verify whether the message is the same as the one signed by Author

```javascript
const message = web3.sha3('The Message');
const signature = await web3.eth.sign(account, message);
const { v, r, s } = ethUtil.fromRpcSig(signature);
```

The first line creates a SHA3 hash of the message we want to sign. This results is 32 bytes (256 bits) hash.
Second line then uses Ethereum’s JSON RPC to tell the Ethereum wallet (which controls the private key) to sign the message on a given account, resulting in a signature. The final line is decoding the JSON RPC output so that we obtain the signature values $v$, $r$ and $s$. In Solidity, a signed message can be verified with the following code:

```solidity
function getOriginAddress(bytes32 signedMessage, uint8 v, bytes32 r, bytes32 s)
constant returns(address) {
bytes memory prefix = "\x19Ethereum Signed Message:\n32";
bytes32 prefixedHash = keccak256(prefix, signedMessage);
return ecrecover(prefixedHash, v, r, s);
}
```

This code will return the Ethereum address (public key) that was used to sign the message. Any change of the message hash or signature will result in a different address than the origin address. [59]

Messages are encrypted using an encryption key. The exact same key will be used to decrypt the message. The only users that can generate the same encryption key are the sender and the recipient. The computation will be done using the sender’s private key and the recipient’s public key. On the recipient side, the computation will be done using the sender’s public key and the recipient’s private key. Instances of the Cipher class are used to encrypt data. Cipher class is part of Crypto module. The ECDH class is a utility for creating Elliptic Curve Diffie-Hellman (ECDH) key exchanges [43]. “aes256” is a Node.js module to simplify using the built-in crypto module for AES-256 encryption with random initialization vectors. This module generates a random initialization vector each time one of the encrypt methods is called [35]. The ECDH key exchange method generates a shared secret from an ephemeral local elliptic curve private key and ephemeral remote elliptic curve public key. This key exchange method provides explicit server authentication using a signature on the exchange hash. Every compliant SSH ECC implementation must implement ECDH key exchange [53].
5.3 Functionalities

- Sending Transactions

Method `eth_sendTransaction`, creates new message call transaction or a contract creation, if the data field contains code [39]. The parameters are:

- **Object** - The transaction object
  - **to**: data, 20 bytes - (optional when creating new contract) The address the transaction is directed to. The `to` field in ChatApp is the `ContractAddress`. There is no `from`, because when the transaction is signed with the private key, it is obvious who is sending it
  - **gas**: quantity - (optional, default: 90000) Integer of the gas provided for the transaction execution. It will return unused gas.
  - **gasPrice**: quantity - (optional, default: To-Be-Determined) Integer of the gasPrice used for each paid gas. It is the reward that is sent to the miner and it is the cost of each unit of gas when we send the transaction. Every time Ethereum is built on the blockchain gas has to be paid.
  - **gasLimit**: is safeguard for the maximum money allowed to be sent
  - **value**: quantity - (optional) Integer of the value sent with this transaction. For ChatApp this value is zero, since no ether will be transferred
  - **data**: data - The compiled code of a contract or the hash of the invoked method signature and encoded parameters. We send data which is the bytecode of the smart contract
  - **nonce**: quantity - (optional) Integer of a nonce. Nonce is the account’s transaction count. It has to be hexadecimal. It is a safeguard that prevents the double spending problem. For a brand new account the nonce value is zero.

Returns: data, 32 bytes - the transaction hash, or the zero hash if the transaction is not yet available.
Method `eth_estimateGas`, generates and returns an estimate of how much gas is necessary to allow the transaction to complete. The transaction will not be added to the blockchain. Note that the estimate may be significantly more than the amount of gas actually used by the transaction, for a variety of reasons including EVM mechanics and node performance [37]. This method returns the amount of gas used.

Updating the balance of the logged in account is done by web3’s `getbalance` property:

```javascript
web3.eth.getBalance(this.walletAccount.getAddress().toString('hex'));
```

Ethereum web3.eth package allows the app to interact with an Ethereum blockchain itself and the deployed smart contracts.

The “`web3.eth.Contract`” will create a web3 contract object that represents the Ethereum smart contract.

The last updated block number is achieved by “`web3.eth.getBlockNumber()`”.

Transaction count is done by:

```javascript
var transactionCount = await web3.eth.getTransactionCount(this.account.getAddress());
```

The “`web3.eth.sendSignedTransaction` will sends an already signed transaction. The “`web3.eth.getGasPrice`” will return the current gas price oracle. The gas price is determined by the last few blocks median gas price. GasPrice is the wei per unit of gas.

- Local Storage
  Maps the Ethereum addresses with its information.

Also to get and store private keys, get addresses, get and set balance, get and set the joined status, update local messages, `window.localStorage` has been used. [12]

### 5.4 Web3

In the Ethereum world, `web3` is used as an absolute solution for communicating between a JavaScript app and the Ethereum network. Web3 is the library for programmatic access to a deployed contract on the blockchain. All functions in our contract are asynchronous that will be executed using `web3`. Different instances of Web3 library were built in the
project, so each instance could connect to a different Ethereum network. One instance of Web3 at a time is used. Web3 expects a defined network, with which it would communicate. We call it a provider. The provider is coming from the local test network but can be swapped out for another provider from a different network.

```javascript
// connecting to local provider
const ganache = require('ganache-cli');
const Web3 = require('web3');
// creates an instance of web3 and tells it to connect to local test network
const web3 = new Web3(ganache.provider());
```

Ganache will take care of Gas and payments, since the accounts are in the unlock state, therefore no need to get involved with public and private keys to gain access. The only way to deploy to a network is to have access to an account.

![Unlocked Accounts in Ganache Local Test Network](image)

Figure 20 - Unlocked Accounts in Ganache Local Test Network [57]

5.5 The Testing Architecture

It is very important to test the smart contract before we deploy it to network because the blockchain is immutable and also transactions on the blockchain cost gas. The only way to fix a bug is to deploy a new version of the contract; the old version will always stay on the blockchain. Testing the smart contract before deploying it helps to make sure the functions have the expected behaviors.

By compiling the contract we get the ABI and bytecode as output. We deploy the bytecode to the local test network. The test will call different functions inside the contract. Figure 21 shows the test architecture.
We generate a local network solely for deploying and testing the smart contract. This involves a custom script. A library named Ganache, which is discussed in previous chapter, will create the local network. The JavaScript interface (ABI) will be fed to web3. A series of tests were written to make sure the app is running as it was expected without errors before it is deployed to the main network.

5.5.1 Testing with Mocha
Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple. Mocha tests run serially allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases [41]. Mocha is a general testing framework.
Figure 23 shows the structure of Mocha.
Mocha functions:

- “it”: to run one individual assertion on the test target. It’s the unit of measurement for testing. It takes two values: one is the value that the code has produced and the other value is the desired correct value and compares them to make sure the value produced by the code is equal to that.

- “describe” The it statement is the core of any test and describe groups it functions together. Describe keeps the test organized. Inside the describe block is more than one it statement.

- “beforeEach” is a utility function that is used to extract some amount of logic that is common to a lot of our test. We have multiple it functions that are doing similar logic. We use beforeEach to extract that logic and can only write out code one time. It will execute some general setup code.

The assert module provides a simple set of assertion tests and was used to test invariants. “assert.ok()” will test, an AssertionError is thrown with a message property set equal to the value of the message parameter in case of error. “assert.equal()” tests shallow, coercive equality between the actual and expected parameters using the Abstract Equality Comparison (==). beforeEach function goes above all the it functions. Any logic inside the beforeEach function will be executed before any it statement. beforeEach runs some common initializing or configuration logic code before each it function. It is used when some logic is used repeatedly.

```javascript
beforeEach(async () => {
    accounts = await web3.eth.getAccounts();
});
```

In `package.json` add: "test": "mocha",

```bash
$ sudo npm install --g mocha
$ npm run test
```

Since we specified the test in `package.json` with Mocha, it will look in the test directory to run its entire test. After running tests it will give the number of passing tests.
5.6 Build, Compile and Deploy

5.6.1 Solidity Compiler

Solidity compiler compiles the smart contract into bytecode and ABI. The compile and deploy structure is shown in figure 20. Solidity Compiler was installed as part of npm package.

![Diagram of Compile & Deploy to Rinkeby Structure]

The reason we can’t simply require our `ChatApp.sol` file is because Node will try to execute files as JavaScript and would throw an error with a .sol file.

The `compile.js` file will look into the contract folder and compile the solidity files. The contents of the file will be read of the hard drive. Therefore we make use of two separate built-in modules.

```javascript
const solc = require('solc');
const path = require('path');
const fs = require('fs-extra');
```

The `path` module will build a directory path from the current compiled js file into the contract file. By using the `path` module we are guaranteed to get cross-platform compatibility. This will generate a path that points directly to ChatApp.sol file.

```javascript
//contract is the directory for the smart contract
const appPath = path.resolve(__dirname, 'contracts', 'ChatApp.sol');
```

To read the raw source from the contract we create the constant source. Now we can write the compiled statement, this is where the `solc` comes to work.
5.6.2 Truffle Compiler

The `buildPath` variable finds the current working directory (`__dirname`) and navigates to the chatApp.sol file within it. The source variable is populated by the raw source code of ChatApp.sol, which is located in the `buildPath`. The UTF-8 string represents the encoding of the file. We call `compile()` on the source code and specify how many contracts we’ll be compiling. In our case, it’s one. The deployed result is always an object. This contract is the top-level object, containing the compiled contract. This is the code that will be deployed onto the blockchain and executed in the Ethereum Virtual Machine (EVM). It is not human readable. ABI is the JavaScript methods that we’ll be able to use when testing and deploying the contract. Migration scripts will compile and deploy the contract to the blockchain. The ABI will change anytime the migration is run. It will give different addresses of the contract based on the networks that is configured. If it is deployed to a test network, the main network or development network these addresses are going to change and JSON file that is supplied by JSON for free will know about all those changes. In contractManager.js file the JSON file is being read from `import compiledContract from '../build/ChatApp.json';` and generating a truffle contract that can be used to interact with. A truffle contract is generated from that JSON file, which creates an intelligent JSON file and library of truffle.

Inside the directory:

```
$ truffle compile
```

Once the contract is compiled, the build is migrated and stored in the `build` directory. The Truffle configuration file is set up to deploy to the desired network, which is the development network. In the `truffle.js` the network is configured. The port is Truffle’s default port.
Whenever a smart contract is deployed to the blockchain we are creating a transaction. The transaction is signed with a private key.

```javascript
module.exports = {
    networks: {
        development: {
            host: "127.0.0.1",
            port: 8545,
            network_id: '\x'
        }
    }
};
```

The network is all configured now and the next step is to run the migration script. Migrations are run in order.

```
$ truffle develop
```

In another terminal, the network is specified. The smart contract will be migrated to the network and ready to be accessed by DApp.

```
$ truffle migrate development
```

Truffle’s default configuration looks for a contracts folder, a migrations folder and a test folder.

Install solc

```
$ sudo npm install -g solc
```

At this point the solidity code needs to be compiled into an interface (abi) and bytecode(bin) [47]. In the .sol directory:

```
$ solcjs --abi chatApp.sol
$ solcjs --bin chatApp.sol
```

The other way to get abi and bin is through the remix.ethereum website. This will start a node pointing at the Rinkeby test network. This command will use the default Ethereum data directory, which on Mac is Library/Ethereum/Rinkeby [47].

```
$ geth --rinkeby console 2>> ./rinkebyEth.log
```
Check to see if there is enough ether in the account with etherscan:

```
> personal.unlockAccount(eth.coinbase)
Unlock account 0x491e60edbf0a77a613b1f06e0c00261bd56e37ae
Passphrase: 
```

<table>
<thead>
<tr>
<th>TxFhash</th>
<th>Block</th>
<th>Age</th>
<th>From</th>
<th>To</th>
<th>Value</th>
<th>[TxFee]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x146ad5a1044e4b...</td>
<td>4296284</td>
<td>12 hrs 47 mins ago</td>
<td>0x31bf8d1407bcd...</td>
<td>IN</td>
<td>0x491e60edbf0a77a...</td>
<td>3 Ether 0.000021</td>
</tr>
</tbody>
</table>

```
> personal.unlockAccount(eth.coinbase)
```

Set up the bytecode and abi variables

```
> var cAabi = eth.contract(<CONTENTS_OF_ABI_FILE>)
> var bytecode = '0x'<(CONTENTS_OF_ABI_FILE)'
```

The method `deploy` is used for deploying.

```
var chatApp = artifacts.require("chatApp");
module.exports = function(deployer){
    deployer.deploy(chatApp);
}
```

### 5.6.3 The Build Process

To build the application for production the command `npm run build` is used. A build folder containing the deployable build is created and only that folder needs to be deployed. The url to the address will be built and goes in the `package.json`.

```
$ npm run build
```

It correctly bundles React in production mode and optimizes the build for the best performance. The build is minified and the filenames include the hashes [37]. By running this command ChatApp is ready to be deployed.

The best approach is to compile one time and write the output to a file in `build` folder that can be read in the future. The compiled script is run one time and then run the application
as many as time as desired without waiting for solidity to compile it again. If we run the compiler again it means the smart contract has changed. In that case the entire build folder if it exists needs to be deleted. After that it will read the contents of the smart contract (.sol). Then it will compile the contract with solidity compiler. The output is written to the build directory as JSON file. The api can be read from the build directory without having to recompile the contract

```javascript
//logic for deleting entire build project if it exists:
//build a path to the folder & get a ref to the build folder
//with the ref to build dir -> call function on fs module
//to remove entire directory & anything inside it
const buildPath = path.resolve(__dirname, 'build');
fs.removeSync(buildPath);
//read the contract form the contracts folder
const chatPath = path.resolve(__dirname, 'contracts', 'ChatApp.sol');
//read the source code to that file
//pass the chatPath and specify the encoding of the file: utf8
const source = fs.readFileSync(chatPath, 'utf8');
//read the source code to that file
//the only property we care about is the contract property
//will pull off the contract property and assign the output to it
const output = solc.compile(source, 1).contracts;
```

The output variable contains the object that is the output from compiling the ChatApp contract. The build folder needs to be recreated since it was deleted.

```javascript
//rebuilding the build folder, checks to see if the dir exists if not builds it
fs.ensureDirSync(buildPath);
for (let contract in output) {
    fs.outputFileSync(
        path.resolve(buildPath, contract.replace(':', ''), '') + '.json',
        output[contract]
    );
}
```

The build folder and JSON file inside it will be generated after compiling.

### 5.6.4 Smart Contract Deployment to Main/Test Networks

In order to write procedures on the Ethereum network, gas is required. Rinkeby test network has been used in this project. Ethereum Rinkeby faucet provides free gas to the provided address.
The web3 library is the sole mean for interacting with Ethereum network. It’s the portal to everything with Ethereum.

```javascript
// connecting to local provider
const ganache = require('ganache-cli');
const Web3 = require('web3');
// creates an instance of web3 and tells it to connect to local test network
const web3 = new Web3(ganache.provider());
const {interface, bytecode} = require('./compile');

// get a list of accounts
accounts = await web3.eth.getAccounts();
// use one of these accounts to deploy contract
chatApp = await new web3.eth.Contract(JSON.parse(interface))
  .deploy({ data: bytecode })
  .send({from: accounts[0], gas: '300000'});
```

Constructor function allows us to interact with existing contracts on the blockchain, or to create and deploy new contracts. The first argument to the contract’s constructor is the ABI. This is where we get a conversion between the JavaScript world and Solidity world. The `interface` property is passed to `JSON.parse`. Solidity compiler gives out the `interface` in JSON. A JavaScript object needs to be given to the contract and this is where it is taken care of. It will tell web3 that there is a contract out there and gives it the interface it expects. There is no information about the deployment or creation. The deployment of the contract is asynchronous therefore `await` is used.

Deploy tells web3 that there is a contract that needs to be deployed to the network. It will create a transaction object and it has the data property. The bytecode is specified in data.
The `send` function takes the contract and deploys it to the network. Calling `deploy` will create an object that then can be deployed to the network. The `send` method triggers the communication from the web3 to the network. It takes `from` property, which its value is the account to be used to create the contract. The process of migrating to a local development network or out to the Rinkeby Test Network or even to the main network is similar. Ethereum Virtual Machine doesn't do any compilation; it simply parses the bytecode [27].

There are some differences between deploying to a local network using Ganache and to the main/test network. The accounts are unlocked using Ganache and it has free Ether in it. Deploying to main/test network an account with Ether required. Figure 25 illustrates that Web3 does not operate on its own and setting up a provider on the instance is required. This is how web3 communicates with some given network. Provider tells web3, which network it, needs to communicate with.

![Figure 25 - Deployment Process to Main/Test Network [58]](image)

*Provider* needs an account that is used as a source of Ether for deployment. The accounts mnemonic (12 words provided by MetaMask) is used to unlock the account and use the Ether within it to deploy the contract [58]. The accounts mnemonic is passed to the account that is going to be created. In local deployment, the provider that was given by Ganache had been used. To deploy to main network, the setup has to be done by the programmer. Some node that we can connect to is required to deploy to Rinkeby network. The common solution is to run a local node on our machine. It would be a real Ethereum node that can connect to the network, and then we can use this node that is running on our machine to connect to the Ethereum network.
5.6.4.1 Deploying to Main/Test Network with Infura

Running a local code on the machine could be time consuming and the setup could be difficult. The Infura service by ConsenSys, provides Ethereum clients running in the cloud, so eliminates the need to install a node on the local machine to work with Ethereum [33]. Infura is a public API. It provides the ability to access a node that is hosted on Rinkeby network by Infura. Infura is the portal beyond web3 into the Rinkeby network. The company that runs Infura API also has nodes that run other networks [58]. Infura is an easy way to deploy and access the network. It supports Mainnet, Ropsten, INFURAnet, Kovan, Rinkeby and IPFS networks. By setting up a project, a unique link to access the Infura API is provided. After signing up to the service the user is provided with a token to use to connect to the relevant Ethereum network:

Main Ethereum Network:
https://mainnet.infura.io/<your-token>

Test Ethereum Network (Rinkeby):
https://rinkeby.infura.io/<your-token>

By having the Infura link, we can install the modules and get started with the deployment. One module that helps creating a provider that will connect to the Infura network is truffle-hdwallet-provider.

```
$ npm install --save truffle-hdwallet-provider
```

As shown in 25, provider is how the web3 instance talks to the particular network and truffle-hdwallet-provider is a special provider that not only allows us to connect to Rinkeby network via Infura but also at the same time unlocking the account.

Inside deploy.js which is in the same directory as compile.js:

```javascript
const HDWalletProvider = require('truffle-hdwallet-provider');
const Web3 = require('web3'); //web3 constructor
//import the interface and bytecode from the compiled script
const {interface, bytecode} = require('./compile');
```

Unlock the account and specify the outside API or node to connect to. Two arguments will be passed on to this new object of HDWalletProvider(). One is the account’s
mnemonic (to derive public and private key, therefore total access to the account through this provider) and the argument is the url of what we will connect to which is the Infura link:

```javascript
const provider = new HDWalletProvider(
  'mango apple bread orange pasta kiwi rice strawberry cereal almond dates flour ',
  'https://rinkeby.infura.io/v3/2c34e8394a2334b72361a62789'
);
const web3 = new Web3(provider);
```

The last line of code will take the `provider` and pass it onto web3 constructor and get an instance of web3. The web3 constructor is completely enabled for the Rinkeby network.

The code for deploying in `deploy.js` is very similar to `chatApp.test.js`. We get a list of accounts from web3 instance. Then creating a constructor creates a new instance of the contract. Then `deploy` and `send` will be called. There are two asynchronous calls, one to attempt to get a list of the accounts to access and a second one to create and deploy that contract. The `async await syntax` has been used but it has to be inside the function.

Account 1 is used as the base to deploy the contract. A console log will show which account we are using. Inside the instance of web3, we will access the Ethereum module to make a new contract that takes the ABI. Since the ABI is in JSON we will pass it to JavaScript object. After that is `deploy()`. We pass an abject to deploy which is the contract’s bytecode. `send` will deploy it to the network. The object that we pass on to `send` is the amount of gas and from which account. We also record the address where it was sent.

```javascript
const deploy = async() => {
  const accounts = await web3.eth.getAccounts();
  console.log('Deploying from account', accounts[0]);
  // result is the instance of the contract
  const result = await new web3.eth.Contract(JSON.parse(interface))
    .deploy({data: bytecode})
    .send({from: accounts[0], gas: '3000000'});
  console.log('Deployed to ', result.options.address);
};
deploy();
```

We are ready to deploy the contract to the Rinkeby network, which could take some amount of time.

```
$ node deploy.js
```
5.6.4.2 Deploying to Main/Test Network with Geth/Local Hosting

After testing the ChatApp locally by using Truffle, the next step is going to be to install an Ethereum client. This will run a node of Ethereum, as well as the EVM on the users machine. The Ethereum client will sync it up to testnet or the mainnet, in order to migrate the smart contract to different networks. Homebrew was used to install the package manager on macOS. Unlike Truffle that comes with built in accounts, we need to create our own accounts that can be used for testing. Once a new account is created we get our own address. By calling Geth a new connection to the Ethereum main network is created, and is synching up with the main network.

Install Homebrew:

```bash
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

==> Installation successful!

==> Homebrew has enabled anonymous aggregate formulae and cask analytics.
Read the analytics documentation (and how to opt-out) here:
  https://docs.brew.sh/Analytics

==> Homebrew is run entirely by unpaid volunteers. Please consider donating:
  https://github.com/Homebrew/brew#donations

==> Next steps:
  - Run `brew help` to get started
  - Further documentation:
    https://docs.brew.sh
```

Install geth:

```bash
$ brew tap ethereum/ethereum
$ brew install Ethereum
$ geth account new
```

The `geth` command will connect and start syncing with the main Ethereum network.

```
$ geth
```

---

54
Rinkeby is used as testnet for test ChatApp application. A new account is created by the command below and a new account address will be generated.

```
$ geth --rinkeby account new
```

To start the test on Rinkeby network, use the command below, rpc for remote procedure calls for log in

```
$ geth --rinkeby --rpc --rpcapi eth,net,web3,personal
```

This will start firing up Rinkeby test network. It will connect to an enode on one of the Ethereum nodes, which is an address of a specific node. It will use this node to sync and get everything it requires.
Also it begins syncing the blockchain.

The next step is to add Rinkeby to the truffle config file. 8545 is the Go-Ethereum port
Once the Rinkeby network is set up, syncing occurs then the contract can be deployed to Rinkeby network. Connect to geth server and run the command below to make all the api modules available:

```
$ geth attach http://127.0.0.1:8545
```

To check if it is still syncing:

```
> eth.syncing
```

The test account (personal) needs to be unlocked so it is possible to interact with it via Truffle. To unlock the account:

```
> personal.accountUnlock(eth.account[0])
```

With geth configured, it is time to run the migrate command and deploy the contract to Rinkeby using Truffle:

```
$ truffle migrate --network rinkeby
```

The address that is provided can be tracked on Rinkeby.etherscan.io/givenAddress

To deploy to mainnet we sync the local network with the mainnet and not the Rinkeby net. The network id matches the mainnet in the truffle.js file.

### 5.7 Connecting to the Blockchain and Interacting with it

The `web3.eth.Contract` object makes it easy to interact with the smart contract on the Ethereum blockchain. When a new contract object is created, the JSON interface of the respective smart contract has been given and web3 will auto convert all calls into low-level ABI calls over RPC for the developer. This will allow the developer to interact with smart contract, as they were JavaScript objects. [29]

```javascript
getContract = async () => {
  this.contract = await new web3.eth.Contract(JSON.parse(compiledContract.interface),
    Config.ENV.ContractAddress);  
  appDispatcher.dispatch({
    action: Constant.EVENT.CONTRACT_READY
  })
}
```

### 5.8 Application of Metamask

Since interaction with MetaMask is used in this project, MetaMask will inject it's own web3, therefore web3 is defined [46].
This way if we have MetaMask running it will use the MetaMask’s version of web3. Web3 will allow the contract to directly interact with the Ethereum blockchain. In Metamask login with Mnemonic code and use the public key from Truffle development to interact with the contract. This will ask for signature and will launch the MetaMask.

![Figure 26 - Signature Request from MetaMask](image)

### 5.9 Decentralized Hosting with Swarm

To interact with the deployed contract on the network we need a frontend. ChatApp is written in React. The transactions are made to the ChatApp’s contract. The messages and user’s contact list are made based on history of events generated by ChatApp’s contract. The web app also handles the encryption and decryption of the application. Unlike most web applications, no email or password combination is required to check against database. The Web3 library will take care of getting us the list of the available user’s addresses, either from MetaMask or the local blockchain that is run in development. The entry point represents the node.js file that starts to run the app and by default it is `index.js`. Main index component, manages the whole app at top level. It also manages the state and React. This is where the app configurations, setting up web3 and instantiating the contract happens. This component will render all of the sub components.
The frontend of ChatApp was deployed to the decentralized network Swarm. An account is required to start up Swarm.

```
$ geth account new
```

The above command provides an address that can be used in Swarm. Ens-api is Ethereum naming service, which is turned off in the command below. Bzzaccount is the Swarm account and the private key provided from the previous command goes there.

```
$ swarm --ens-api "" --bzzaccount providedAddress
```

In the web directory upload everything:

```
$ swarm --recursive up ./
```

The command above returns a hash, which is the link to the Swarm. Anyone running an Ethereum node can access this link. It has direct access to the contract (after it is deployed to the mainnet). There is no hosting environment or central location for ChatApp and everything necessary to run ChatApp is located on the Ethereum network.
Chapter 6 - Conclusion

The goal of this project was to analyze inefficiencies of the traditional central based messaging applications and address them by using Ethereum smart contracts in trustless and secure decentralized application. This has been achieved by developing a decentralized application that can be run by any user to send and receive encrypted messages. Adding capabilities like picture and video sharing, customized chat groups and exploring a new technology in blockchain called Whisper for messaging are some of the future directions.
References

[13] Suyash Gupta and Mohammad Sadoghi Department of Computer Science, University of California, Davis, CA, USA, Blockchain Transaction Processing
[16] Starkenmann, Oliver 2017, Implementation of a Crowdfunding Decentralized Application on Ethereum, University of Zurich
[18] Buterin, Vitalik et al. 2014. A next-generation smart contract and decentralized application platform
[34] Create robust user interfaces with React.js url: [https://www.lynda.com/React-js-tutorials/Create-robust-user-interfaces-React-js/761961/5011943-4.html?srchtrk=index%3a38%0alinktypeid%3a2%0aq%3areact%0apage%3a1%0as%3arelevance%0asa%3atrue%0aproducttypeid%3a2} Accessed [15-Sep-2018]
[36] Node.js Essential Training – url: [https://www.lynda.com/Node-js-tutorials/How-Node-js-works/417077/454426-4.html?srchtrk=index%3a1%0alinktypeid%3a2%0aq%3anodejs%0aes%3arelevance%0asa%3atrue%0aproducttypeid%3a2} Accessed [11-Dec-2018]
[53] D. Stebila, J. Green, Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer