Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Integration of IoT Devices via a Blockchain-based Decentralized Application

Afzaal Ahmad

Course of Study:	Infotech
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	M.Sc. Michael Falkenthal, Dr. rer. nat. Uwe Breitenbücher
Commenced:	23 May 2017
Completed:	23 November 2017
CR-Classification:	C.2.4, E.1, H.3.4, H.3.5, H.5.2

Abstract

Blockchains are shared, immutable ledgers for recording the history of transactions. They foster a new generation of transactional applications that establish trust, accountability, and transparency. It enables contract partners to secure a deal without involving a trusted third party. Initially, the focus was on financial industry for digital assets trading like Bitcoin, but with the emergence of Smart Contracts, blockchain becomes a complete programmable platform. Many research and commercial organization start diving into blockchain world bringing new ideas of its application in different sectors like supply chain, Health, and autonomous shopping.

This thesis presents an idea to integrate Internet of Things (IoT) devices via a blockchain based decentralize application based on Ethereum.The application consists of front-end application which can be deployed to any web server, and a smart contract which will be deployed on private blockchain network comprises of Peer-to-Peer (P2P) connected IoT devices acting as full ethereum node. The application emulates the digital transport ticketing system where the asset is a ticket which be can be bought and paid by the user using ether in their ethereum account on blockchain. Once the purchase transaction is mined, then it is propagated to all the peer. Ticket now can be accessed locally without requesting any centralize device which makes the system easily accessible, and safe because of the data Integrity and decentralization on the blockchain.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ABI Application Binary Interface. 46

AJAX Asynchronous JavaScript and XML. 48

API Application program interface. 36

CPU Central Processing Unit. 5

CSS Cascading Style Sheet. 16

DAG Directed Acyclic Graph. 5

Dapp Decentralize Application. 5

DNS Domain Name Service. 54

DOM Document Object Model. 48

EOA Externally Owned Accounts. 7

EVM Ethereum Virtual Machine. 5

GB Giga Bytes. 40

GDP Good Distribution Practice. 51

GPU Graphical Processing Unit. 5

GUI Graphical User Interface. 31

HTML Hyper Text Markup Language. 16

HTTP Hyper Text Transfer Protocol. 51

ICO Initial Coin Offering. 16

IoT Internet of Things. 3

IP Internet Protocol. 54

IPC Inter Process Communication. 87

IT Information Technology. 99

JSON Javascript Object Notation. 38

P2P Peer-to-Peer. 3

PoS Proof-of-Stack. 19

PoW Proof-of-Work. 19

RAM Random Access Memory. 40

REST Representational State Transfer Protocol. 52

RPC Remote Pocedure Call. 38

SPV Simplified Payment Verification. 23

SVG Scalable Vector Graphics. 49

TX Transaction. 20

URI Universal Resource Identifier. 54

URL Universal Resource locater. 51

UTXO Unspent Transaction Output. 20

VM Virtual Machine. 53

XML Extensible Markup Language. 49

1 Introduction

The concept of blockchain first introduced by Satoshi Nakamoto in 2008 in his paper which is later implemented by Bitcoin blockchain. The paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System" discuss P2P electronic cash system which will allow online payment to be sent directly from one party to another without third-party involvement like Banks. The trust issue was resolved using digital signatures, and a solution was provided to another critical issue with Electronic cash system called double-spending in the form of P2P network. The network timestamps transactions by hashing them to an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed. The longest chain serves as proof of the sequence of events witnessed and that it came from the most significant pool of CPU power. The longest chain cannot be generated by attackers until they own more than 50 percent of CPU power. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain [Nak08].Blockchains are shared, immutable ledgers for recording the history of transactions. They foster a new generation of transactional applications that establish trust, accountability, and transparency.

The primary motive of blockchain development was initially for financial application but after the introduction of smart contract, blockchain applications bear no boundary anymore. Blockchain can now be used as for financial and non-financial applications [CNSK15]. Among one of these applications is IoT. Blockchain allow us to have a distributed P2P network where trustless members can interact with each other without a trusted mediator, in a provable way. Smart contracts-scripts that reside on the blockchain that allows automation of multi-step processes. Blockchain facilitates the sharing of services and resources leading to the creation of a marketplace of services between devices and allows us to automate in a cryptographically verifiable manner several existing, time-consuming workflows. The blockchain-IoT combination is powerful and can cause significant transformations across several industries, paving the way for new business models and novel, distributed applications [CD16]. In the IoT era, new connected devices will spread highly sensitive personal data. Sending this type of data to centralized system represents a severe risk to privacy. A possible solution to protect privacy is to leverage the use of Peer-to-Peer storage networks in combination with the blockchain. However, such architecture, despite promising, embeds still limitations, especially regarding scalability [CVM17]. There are three significant value propositions provide by blockchain based IoT platform given in [OCo17].

1. Build trust between the parties that transact together. Blockchain-based IoT enables devices to participate in transactions as a trusted party. Individuals in a deal may not trust each other but unmodifiable data from devices stored on blockchain provide the necessary trust for businesses and people to cooperate.

2. Reduce costs which enable participants to reduce monetary and time commitment costs by eventually removing the ‘middle man’ from the process. Transactions and device data are now displayed on a peer to peer basis, removing most legal or contractual costs.
3. Accelerate transactions which enable more transactions overall because the ‘middle man’ is removed from the process. Smart contracts allow organizations to reduce the time needed for completing legal or contractual responsibilities.

Blockchain not only gives the solution to trust, reduce cost, accelerate the transaction, protect privacy but also give decentralize storage, accessibility to data on the blockchain. Which provides a solution to decentralize management of data, digital property resolution which having the vital impact on how big data may evolve [KM17].

1.1 Problem Statement

With the introduction of the Turing-complete smart contract into blockchain world, much well-renowned organizations and startups start diving into blockchain world, but until now most of the blockchain projects are Initial Coin Offering (ICO) which mostly raise money for different projects in return gives digital coins to the contributors. Those coins carry a value which depends on the investment in the project and overall popularity. There are few practical projects available which use blockchain technology for some real-world application such as Energy Sharing and Charging system, Smart Locks [slo17], tool booth payment and water meter system [Inn17]. Some research papers are published recently giving some simulated use of blockchain for IoT such as the solutions discussed in [HCK17] and [SD16b]. In this thesis, the primary objectives are to present the fundamentals of blockchain technology in reference with ethereum blockchain, build blockchain based IoT network, and give a detailed development guide for Dapp development based on blockchain technology.

1.2 Scope of Work

The basic motive behind this thesis is to understand and introduce blockchain technology to non-financial applications by developing a Dapp. So the work in this thesis has been divided into two phases, in the first, we will be focusing on understanding the concepts of blockchain in reference with ethereum, and all the available related literature to get entirely familiar with blockchain. In the second phase, we will work on digital transport ticketing system scenario based on blockchain and IoT, to realize this scenario we will first create our private ethereum blockchain network. Then discuss and use in detail some development technologies used by ethereum such as writing Smart Contract in Solidity, Using Truffle framework for compilation, deployment, and testing. Then we will develop front-end application to enable users to interact with the smart contract deployed on the blockchain. The key technology to realize front-end application are Web3, Truffle contract, jquery, Hyper Text Markup Language (HTML), Bootstrap and Cascading Style Sheet (CSS). For ethereum account management we will be using MetaMask or Mist browser.

The ticketing Dapp will enable us to buy tickets as a digital asset with digital currency ether; the ticket will be stored on blockchain can be accessed and verified on any node of our network locally without to access any centralized server. All the nodes in the network contain a full copy of the data stored on the network which enables us to access the smart contract locally on any available node.

1.3 Outline

The remaining content of the document are arranged in the follwing chapters.

Chapter 2 – Fundamentals: This chapter discuss the fundamental concept of blockchain technology in reference with ethereum blockchain.

Chapter 3 – Related Works This chapter Gives an overview of Up to date related research work and there approaches and motivation towards the ongoing application development based on blockchain platform.

Chapter 4 – Concept and Specification System requirements such as functional and non-functional and an overview of the system will be presented in this chapter.

Chapter 5 – Design This chapter gives a detailed overview of the Dapp architecture and data model for storing data on the blockchain which represent our application scenario.

Chapter 6 – Implementation and Validation Dapp Development steps will be described in detail from private blockchain network configuration, smart contract writing, compilation, deployment, front-end application development to validating the whole application.

Chapter 7 – Conclusion and Future Work Will summarize the whole work going through each chapter and will give some direction towards the enhancement in the application for future.

2 Fundamentals

In this chapter, the fundamental concept of blockchain technology will be discussed, explicitly focussing on Ethereum blockchain. We will be using it in our project, which is currently most widely used Dapp development platform based on the blockchain, due to the Turing-complete programming language support called solidity that can be used to write any smart contract.

2.1 Introduction to Blockchain

In this section, an overview of the core concept of blockchain will be presented and a breif comparative analysis of Bitcoin and Ethereum core differences.

The decentralized currency was first time implemented by Satoshi Nakamoto in 2009 [Nak08]. Managing ownership via public key cryptography with a consensus algorithm for keeping track of who owns coins, known as Proof-of-Work (PoW). The mechanism behind PoW was a breakthrough because it simultaneously solved two problems.

1. PoW provided a simple and reasonably efficient consensus algorithm, allowing nodes in the network to agree on a set of updates to the state of the Bitcoin ledger.
2. It provided a mechanism for providing free entry into the consensus process, solving the political dilemma of choosing who gets to determine the consensus, while concurrently preventing Sybil attacks.

PoW perform this by substituting a formal obstacle to participation, such as the condition to be registered as a unique entity on a specific list, with an economic barrier, the weight of a single node in the consensus voting process is directly proportional to the computing power that the node possesses. The alternative approach has been proposed called Proof-of-Stack (PoS) calculating the weight of a node as being equivalent to its currency holdings and not its computational resources.

2.1.1 Blockchain as State Transition System

The ledger of a cryptocurrency like Bitcoin may be considered as a state transition system. In which the "state" consisting of the ownership status of all existing bitcoins and a "state transition function," that utilizes the state and a transaction and generate a new state as a result. The state is a balance sheet in the standard Banking system. A transaction is a

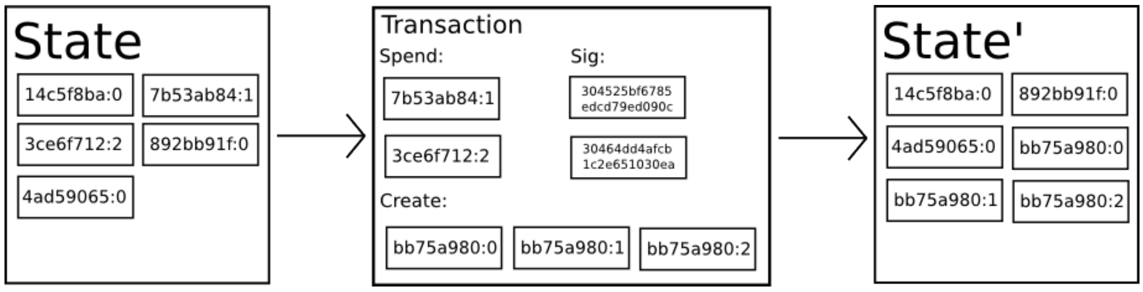


Figure 2.1: Bitcoin State Transition [Eth17a]

request to transfer amount X from account A to B. The state transition function reduces the value of former's account by X and raises the value of later one by X. If the sender account has less than X value, then the state transition function returns an error. The state transition function can be defined as

$$\textbf{APPLY(S,TX)} \rightarrow \acute{S} \textbf{ or Error} \text{ [Eth17a]}$$

The Bitcoin state is the collection of all coins Unspent Transaction Output (UTXO) that have been mined but not spent yet, each UTXO having a denomination and an owner defined by a 20-byte address which is a cryptographic public key. A transaction holds one or numerous inputs, with each input contains a reference to an existing UTXO and a cryptographic signature generated by the private key linked to the owner's address, and one or numerous outputs, each output containing a new UTXO to be added to the state.

Bitcoin state transition function

1. For all inputs in Transaction:
   - If the UTXO is not present in S, return an error.
   - If the signature provided does not match the owner of the UTXO, return an error.
2. If the total of all input UTXO denominations is less than the sum of the denominations of all output UTXO, return an error.
3. Return the new state S' with all input UTXO eliminated and all output UTXO added.

The state transition functions steps show that it not only prevents the user from using non-existing coins but also restrict it to not use other people coins and at the same time ensures the conservation of value.

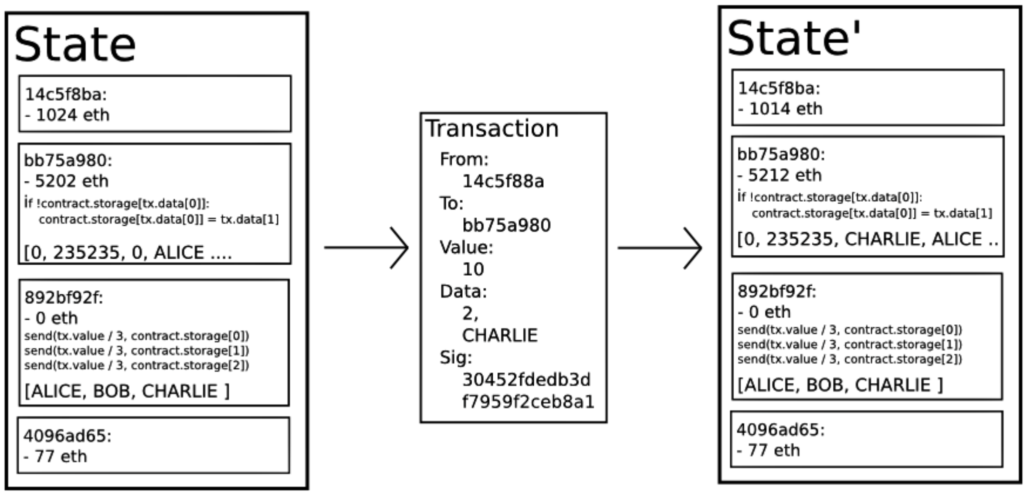

State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ....
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ..
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

Figure 2.2: Ethereum State Transition [Eth17a]

Ethereum state transition function

1. The Transaction (TX) has to be checked whether it is well-formed or not by validating the signature and matching the nonce value in the sender account and the TX.
2. The TX fee has to be calculated as STARTGAS * GASPRICE, sending address is determined from the signature. Then TX fee has to be deducted from the sender's account and sender nonce to be incremented. If spending balance is less than the available balance, an error has to be returned.
3. The initial value of GAS is set to STARTGAS, then gas per byte is paid for each byte of a TX.
4. The TX value has to be transferred from the sender's account to the receiving account. Check whether the receiver account exists or not If it does not exist yet, create it. If in case the receiving account is a contract, then run the contract's code to completion or till the execution runs out of gas.
5. If the transfer of value gets failed due to insufficient balance or execution process runs out of gas, then revert all changes to the state, and except fees payment, then add the fee to the miner account.
6. Remaining gas fee will be refunded to the sender, and the consumed gas fee will be sent to the miner.

2.1.2 Merkling in Blockchain

Merkle tree is a method in which a large number of chunks of data is hashed together which depend on dividing the data chunks into buckets. Each bucket holds only a few chunks, taking the hash of each bucket and perform the same process repeatedly until the total number of remaining hashes becomes one called the root hash [But15].

Binary Merkle tree is the most common form of Merkle tree, in which a bucket always comprises of two adjacent chunks or hashes. The hashing of chunks provide a mechanism

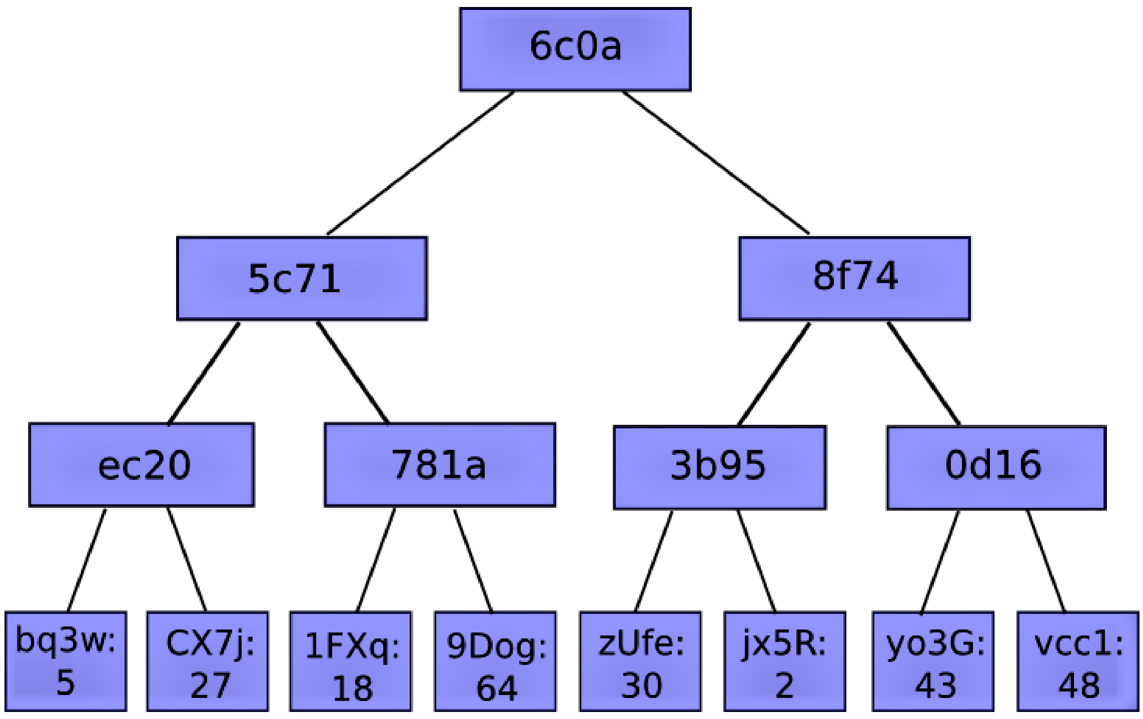


Figure 2.3: Merkle Binary Tree [But15]

called Merkle proof, which comprises of a chunk, the tree root hash, and the branch contains all the hashes from the chunk to the root. The consistency of the hashing can be verified by anyone reading the proof for at least that specific branch, going all the way up to the tree root. So if it is consistent, it means the given chunk exists in the tree. Thus in case if anyone tries to change part of the tree, it will lead to inconsistency somewhere up the chain. As given in Figure 2.4.

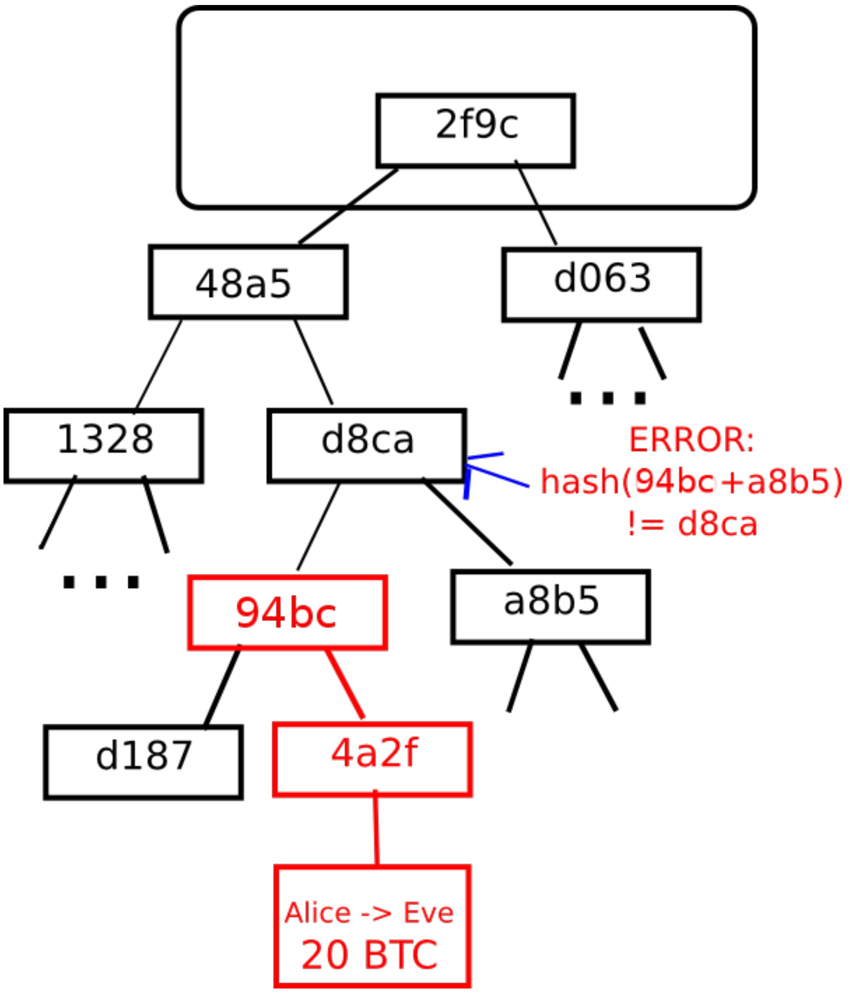


Figure 2.4: Merkle Tree with Inconsistency [Eth17a]

Merkle Proofs in Bitcoin

Satoshi Nakamoto [Nak08] First creates Merkle proofs in 2009 in Bitcoin blockchain [Bit17]. The Bitcoin blockchain uses Merkle proofs for the transactions within every block, gives the benefit of Simplified Payment Verification (SPV), which enables us to use "light client," It only download chain of block headers, 80-byte chunks of for every block that contains five components

1. The previous header hash.
2. A timestamp. Difficulty value of mining
3. A nonce for proof of work.
4. A root hash for the Merkle tree containing the transactions for that block.

To check the status of a transaction by the light client, it will only ask for Merkle proof, which shows the transaction is in one of the trees whose root is in the block header for the main chain. Bitcoin-style light clients have limitations like they cannot prove anything about the current state such as the holding of digital assets, the status of the financial contract. If you want to check, how many Bitcoin do you have right now? Bitcoin light client uses a protocol which will query multiple nodes with the trust that one of them will notify you if any particular transaction spending from your addresses. For complex applications, the exact nature of the outcome of a transaction depends on several previous transactions, which themselves depend in prior transactions. So you will have to authenticate every transaction in the chain ultimately [But15].

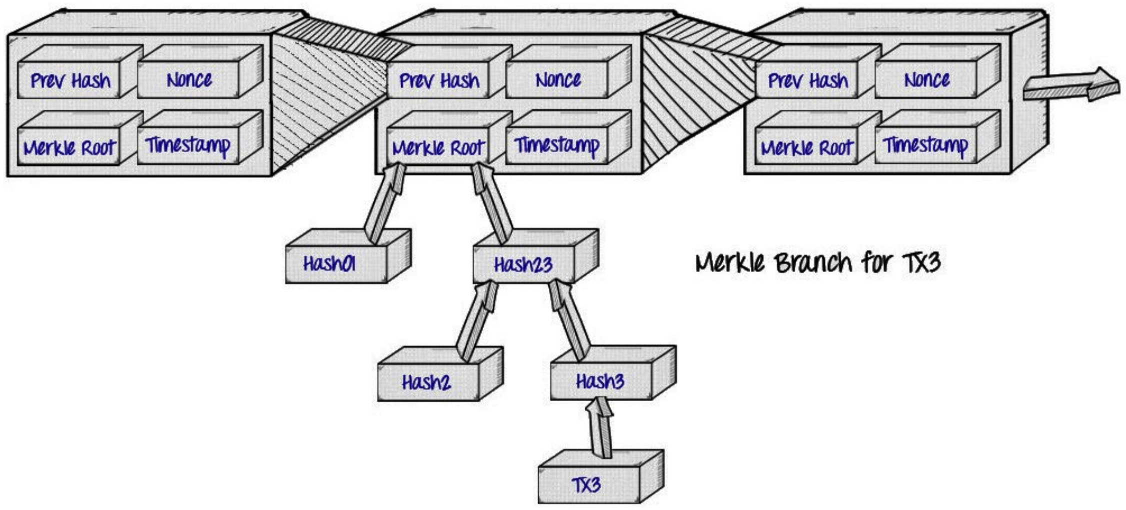


Figure 2.5: Merkling in Bitcoin [But15]

Merkle Proofs in Ethereum

Ethereum block header contains three Merkle trees for three different objects

1. Transactions
2. Receipts (basically, Data pieces determining the effect of each transaction)
3. State

Which provides an advanced light client protocol, which enables light clients to make and get valid answers efficiently to several types of questions.

1. Check whether this transaction has been in included in a specific block?
2. Give me all the instances of an event type X emitted by this specific address in the last 30 days (e.g., crowd-funding contract reaching its goal).
3. What is the current balance of my account?

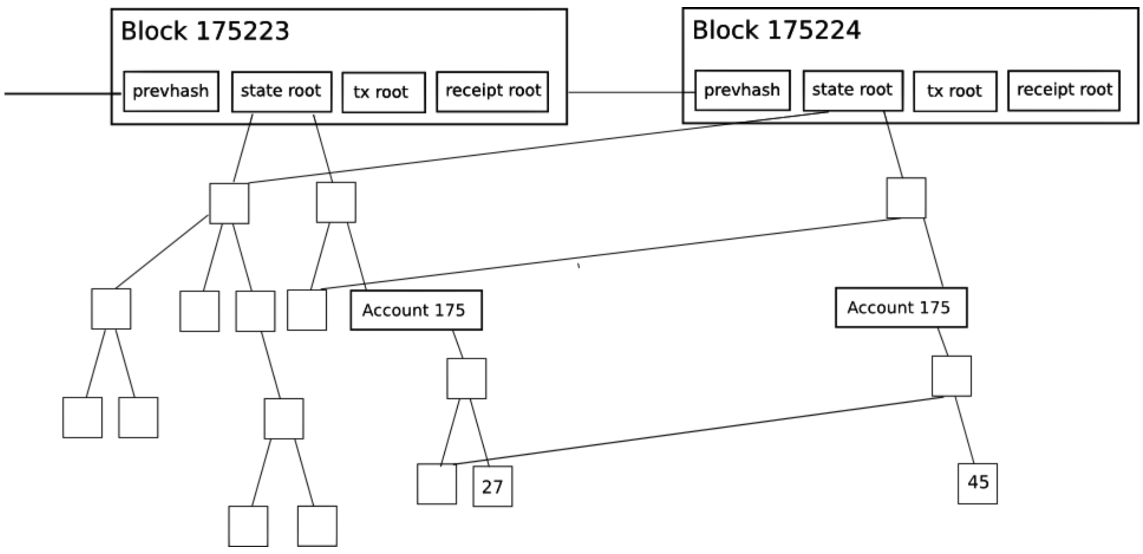


Figure 2.6: Merkling in Ethereum [But15]

4. Is this specific account exist?
5. Let assume by running this transaction on this contract, what will be the output?

The transaction tree handles the first question, the second by receipt tree, the third and fourth by state tree. The first four are reasonably straight-forward to compute; the server just finds the object, fetches the Merkle branch and in reply send the branch to the light client.The state tree also handles the fifth one but its computation method is more complex. Merkle state transition proof has to be constructed. Merkle state transition proof make a claim "if you run transaction T on the state with root S, the result will be a state with root S', with log L and output O" (output exists as a concept in Ethereum due to every transaction is a call).The proof is computed by the server locally by creating a fake block and setting the state to S and pretending to be a light client while applying the transaction. If the process requires applying transaction in order the client to check the account balance of an account, the light client performs balance query. The light client makes query if we need to check a particular item in the storage of a specific contract and so on for further thing needed. The server responds to all of its queries accurately and keeps track of all the data its send back. The server then sends the combined data from all the requests as proof. The client then follows the same exact procedure, but use the provided proof as a database. If the result is equally as what the server claims, then the client accepts the proof [But15].

Ethereum use Merkle Patricia tree which is a more complex form of Binary Merkle tree; it provides a cryptographically authenticated data structure which can be used to store all bindings. They are fully deterministic in a sense Patricia trie with same key and value bindings are guaranteed to be the same down the last byte and therefore have the same root hash [Tea17a].

2.1.3 Scripting in Blockchain

Bitcoin protocol does support a weak version of "smart contract" in its core without any extension. UTXO in Bitcoin blockchain can be owned by a public key and by a complicated script written in a simple stack-based programming language. In this model, data must be provided by the transaction to satisfy the script. Certainly, even the basic public key ownership system has been implemented with a script. The script takes the signature as input, verifies it against the transaction and the address that owns the UTXO and returns the result "one" if the verification is successful or " zero" otherwise. Some complicated script exists for numerous other cases. One of the cases is to write a script that requires two out of three private keys to sign ("multisign") a transaction; it is beneficial for corporate accounts and some merchant escrow situations. The script can also be used to pay bounties for solutions to computation problems. However, Bitcoin provides such basic scripting functionality, but it has some limitations such as [Eth17a].

Lack of Turing-completeness

Bitcoin scripting language support large subset of computation but does not support every computation. The key missing category is looping. The reason behind this is, to avoid infinite loops during transaction verification; theoretically, its is an attainable barrier for script programmers, because any loop can be simulated by merely many times repeating the underlying code with an if statement, But it leads to a script that is very space-inefficient. Unlike scripting in Bitcoin, Ethereum is Turing complete, has built-in language which support all kind of loops.

Value-blindness

UTXO script does not provide fine-grained control over the amount that can be withdrawn. One of the best cases is that of an Oracle contract, where A and B put in €2000 worth of BTC, and after 30 days the script sends €2000 to B and the rest to A. To perform this operation it would need an oracle to determine the value of 1 BTC in €. Since it is a tremendous improvement concerning trust and infrastructure requirements over the centralized solutions that are currently in operation. However, due to all-or-nothing nature of UTXO, the only way to achieve this is the very inefficient hack of having several UTXO of varying denominations. Unlike Bitcoin Ethereum is fully value-aware platform

Lack of state

UTXO remains in two states either spent or unspent; There is no way for a multi-stage contract to keep internal state beyond these two states. Due to this reason its complicated to make multi-stage options contracts, decentralized exchange offers or two-stage cryptographic engagement protocols necessary for secure computational bounties. So UTXO can on be used to build simple contracts not complex state-full contracts like decentralized organizations and makes meta-protocol challenging to implement. Binary state combined with value-blindness makes withdrawal limits impossible in Bitcoin. Unlike Bitcoin, Ethereum fully supports state.

Blockchain-blindness

Bitcoin puts severe limitations in the application of gambling and several other categories, due to UTXO blindness to some blockchain data such as the nonce and previous block hash, which deprive the scripting language of a valuable source of randomness. Ethereum, on the other hand, entirely blockchain-aware platform.

2.2 EVM

Ethereum is a general blockchain platform [Eth17b], which allows developers to define their complex operations in a way they wish to develop. It is used for Dapp development for cryptocurrencies but not limited to it only it this; it can be used for any non-financial Blockchain Dapps, unlike Bitcoin.

Ethereum in the narrow sense refers to protocols suit that defines platform for Dapps. At The heart of it is the EVM [WOO17], which is capable of executing code of arbitrary algorithmic complexity. Ethereum is "Turing complete ." Developers can create Dapps in a built-in language called Solidty, which is modeled on existing languages such as Python and JavaScript.

Ethereum includes a P2P network protocol like any other blockchain. The database of blockchain is maintained and updated by many nodes connected to the network. EVM runs on every node of the network. Likewise, the instructions also run on all the node. Due to this reason, Ethereum is occasionally called a "world computer."

This immense parallelization of computing across the entire Ethereum network does not make computation more efficient. Indeed, this parallelization makes the Ethereum too expensive and too slower than a traditional computer. But every Ethereum node runs EVM to maintain consensus across the network. Decentralized consensus provides Ethereum absolute levels of fault tolerance, which ensures zero downtime by removing a single point of access and make data stored on blockchain censorship-resistant and unmodifiable.

The Ethereum platform is designed to be featureless or value-agnostic. It's up to entrepreneurs and developers to decide for what purpose it should be used, similar to programming languages. Ethereum is particularly suited for an application that wants to automate direct interaction between peers or facilitate coordinated group action across a network. For example applications for P2P market-place coordination or the complex financial contracts automation. Bitcoin allows individuals to exchange cash without the involvement of any middlemen like financial institutions, banks or governments. Unlike

Bitcoin, Ethereum impact is far more than just financial exchanges automation. It can carry out complex financial exchanges, besides that it can be applied to any of kind of non-financial application where security, trust, and permanence are significant for example voting, asset-registries, governance, and IoT.

2.3 Ethereum Clients

The multiple client implementations across a range of different operating systems, give client diversity and a huge win for the ecosystem from the starting days of the project. And let us verify that the protocol in [WOO17] is undoubtful. It makes way for innovations. But still, its is confusing for a user because there is no single universal installer. The leading implementations are go-ethereum and Parity as of September 2016.

Client	Language	Developers	Latest Release
go-ethereum	Go	Ethereum Foundation	go-ethereum-v1.4.18
Parity	Rust	Ethcore	Parity-v1.4.0
cpp-ethereum	C++	Ethereum Foundation	cpp-ethereum-v1.3.0
pyethapp	Python Go	Ethereum Foundation	pyethapp-v1.5.0
ethereumjs-lib	Javascript	Ethereum Foundation	ethereumjs-lib-v3.0.0
Ethereum(J)	Java	<ether.camp>	ethereumJ-v1.3.1
ruby-ethereum	Ruby	Jan Xie	ruby-ethereum-v0.9.6
ethereumH	Haskell	BlockApps	no Homestead release yet

Table 2.1: Ethereum Clients [Eth17b]

Selection of Ethereum client entirely depends on personal preferences and the functionality it provides. Becuase the clients are not developed by a single entity, and it's an open platform, can be developed by anyone by following the standards mentioned in [WOO17]. When the client connects to Ethereum network, then it can communicate to any other node irrespective of whether its using Geth, Parity, Ethereum(j) etc as given in Figure 2.7.

For a user who doesn't want to use the command line to interact with Ethereum or don't want to use miner then Mist / Ethereum Wallet installation is enough and will fulfill their needs. Ethereum Wallet is an only DApp deployment of the Mist browser.

The Mist Browser comes with bundled go-ethereum and cpp-ethereum binaries, which automatically start syncing the blockchain using one of the clients (default is geth) when Mist starts. In case if a private network is intended to be used, then node should be started first then Mist, it will connect to the private network instead starting one itself [Eth17b].

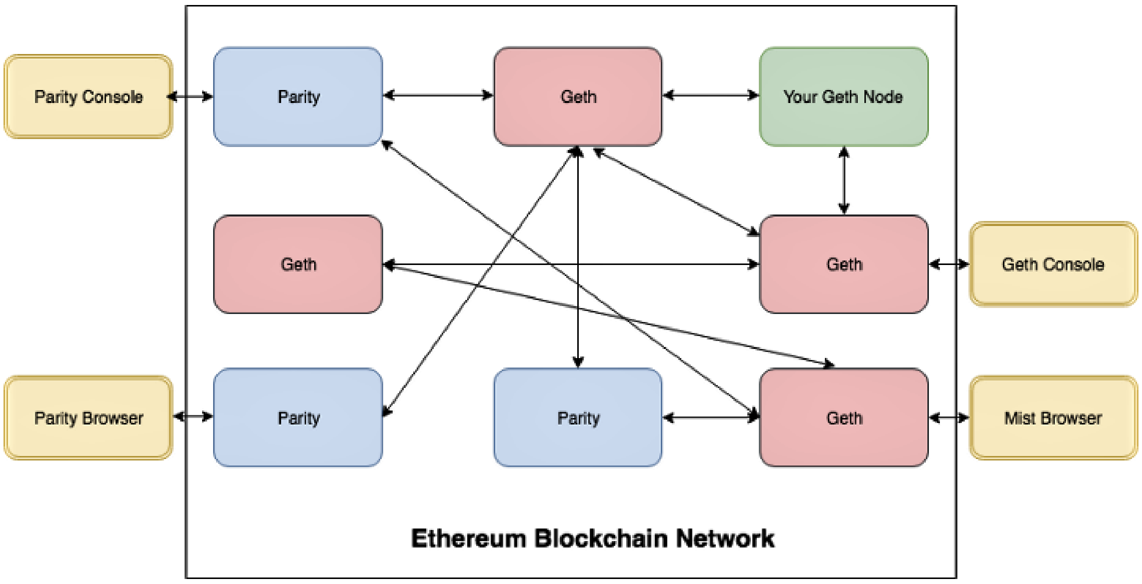
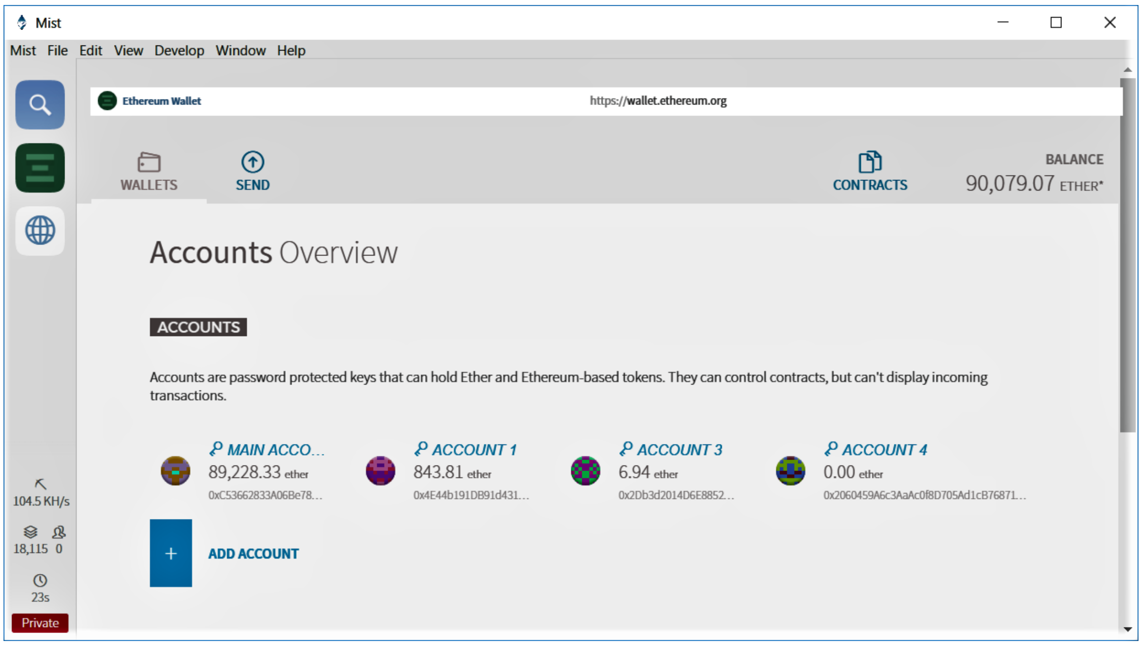


Figure 2.7: Ethereum Blockchain Network with different clients [Mur17]



Accounts Overview

ACCOUNTS

Accounts are password protected keys that can hold Ether and Ethereum-based tokens. They can control contracts, but can't display incoming transactions.

MAIN ACCO...
89,228.33 ether
0xC53662833A06Be78...

ACCOUNT 1
843.81 ether
0x4E44b191DB91d431...

ACCOUNT 3
6.94 ether
0x2Db3d2014D6E8852...

ACCOUNT 4
0.00 ether
0x2060459A6c3AaAc0f8D705Ad1cB76871...

Figure 2.8: Mist browser connected to private network

2.4 Ethereum Account Managment

Accounts are the essential component of Ethereum blockchain. It is vital for a user to interact with Ethereum blockchain via transactions. There are two types of accounts [Eth17a; Eth17b; WOO17]

1. EOA.

2. Contract accounts.

EOA accounts is simply referred as accounts. Both EOA and contract accounts are state objects. They have a state, EOA has balance and contract has balance and storage. With every block creation, the state of every account is updated to achieve consensus in the network. If only EOA accounts used and allowing transactions between them, Ethereum will become "altcoin," system which can only be used for ether transfer and which is less potent than Bitcoin. Accounts are the identities of the external agents such as human personas, mining nodes or automated agents. Public key cryptography is used by Accounts to sign transaction which allows EVM to validate the transaction sender identity securely.

2.4.1 Ethereum Keyfiles

A pair of keys is associated with every account, a private key and public key. The address is used to index accounts and addresses are acquired from the public key by getting the last 20 bytes. Each private key/address pair is encoded in a key file. Key files are JSON text file can be opened in a text editor but the significant part of the key file which is encrypted with the password we used while creating the account, is the private key. Key files are found in the keystore subdirectory of Ehtereum data directory. It is strongly adviced to backup key files regularly .

Creating a key is comparable to creating an account.

1. No need to tell anybody else while creating it.
2. No need to synchronize with the blockchain
3. No need to run a client
4. No need to be connected to the internet.

Indeed new account will not hold any Ether. It is the creator property, and certainly, without the private key and password, nobody will be able to access it. If the key file does not exist on a node, it will not recognize the account, and the user will not be able to perform a transaction on that specific node. So if you ever want to perform a transaction from a system first copy the key file to the keystore directory of that node or copy the entire directory as it is safe to transfer keys between Ethereum nodes [Eth17b].

2.4.2 Ehtereum Account Creation

There are multiple ways to create an account (EOA). Which are

Using geth account new

After geth client installation, Account can be created just by executing the "geth account new command" in the terminal, and for executing this command running geth client or sync up with blockchain is not necessary. Similarly to list all accounts whose keyfiles are in the keystore directory of the current node, "geth account list" command is used.

Listing 2.1 Account creation using geth account new [Eth17b]

```
$ geth account new

Your new account is locked with a password. Please give a password. Do not forget this
    password.
Passphrase:
Repeat Passphrase:
Address: {168bc315a2ee09042d83d7c5811b533620531f67}
```

Using geth console

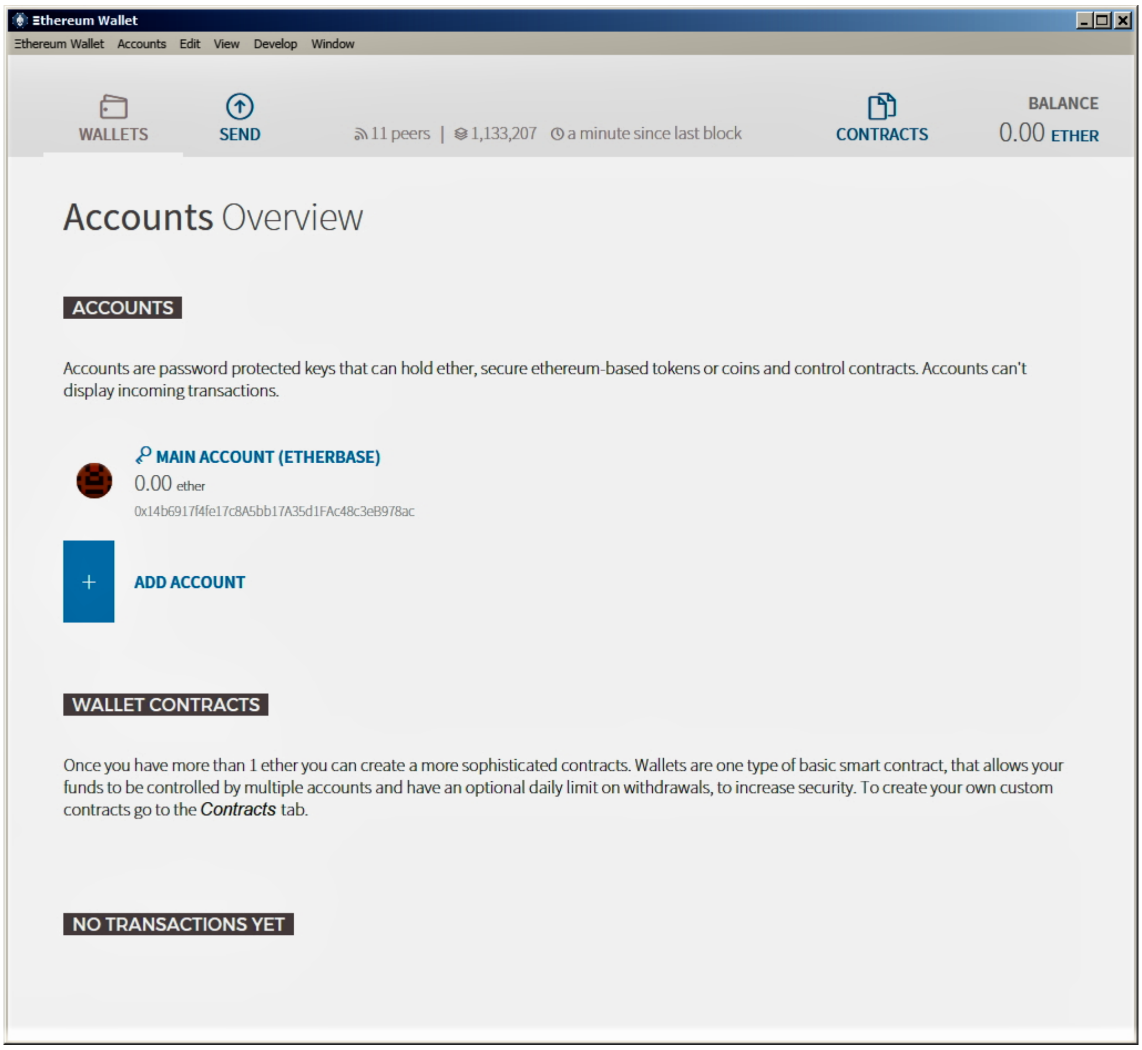
Listing 2.2 Account creation using geth console [Eth17b]

```
> geth console 2>> file_to_log_output
instance: Geth/v1.4.0-unstable/linux/go1.5.1
coinbase: coinbase: [object Object]
at block: 865174 (Mon, 18 Jan 2016 02:58:53 GMT)
datadir: /home/USERNAME/.ethereum

OR

$ geth attach
Welcome to the Geth JavaScript console!

instance: Geth/miner1/v1.7.0-stable-6c6c7b2a/windows-amd64/go1.9
coinbase: 0xc53662833a06be78090b4c67ef06fe533155176e
at block: 17148 (Sat, 28 Oct 2017 01:31:08 CEST)
datadir: C:\Users\fzlah\OneDrive\IotChain\miner1
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0
    web3:1.0

> personal.newAccount()
 Passphrase:
 Repeat passphrase:
 "0xb2f69ddf70297958e582a0cc98bce43294f1007d"
```

To create an account using geth we should have first to start geth in console mode or open a terminal and execute "geth attach" command to attach to an already running instance. The console allows interaction with the local node by issuing commands as shown en Listing 2.2.

Using Mist Ethereum Wallet

The official Mist Ethereum wallet is the Graphical User Interface (GUI) based solution for creating accounts. Ethereum foundation is developing the Mist project. The wallet Dapp is available for all platforms Linux, Mac, and Windows. When wallet Dapp loads for the first time after synchronization with blockchain, it creates the primary account called "coinbase" automatically. To add new account follow the following steps.


Ethereum Wallet
Ethereum Wallet Accounts Edit View Develop Window
WALLETS
SEND
11 peers | 1,133,207 a minute since last block
CONTRACTS
BALANCE
0.00 ETHER

Accounts Overview

ACCOUNTS

Accounts are password protected keys that can hold ether, secure ethereum-based tokens or coins and control contracts. Accounts can't display incoming transactions.

MAIN ACCOUNT (ETHERBASE)
0.00 ether
0x14b6917f4fe17c8A5bb17A35d1FAc48c3eB978ac

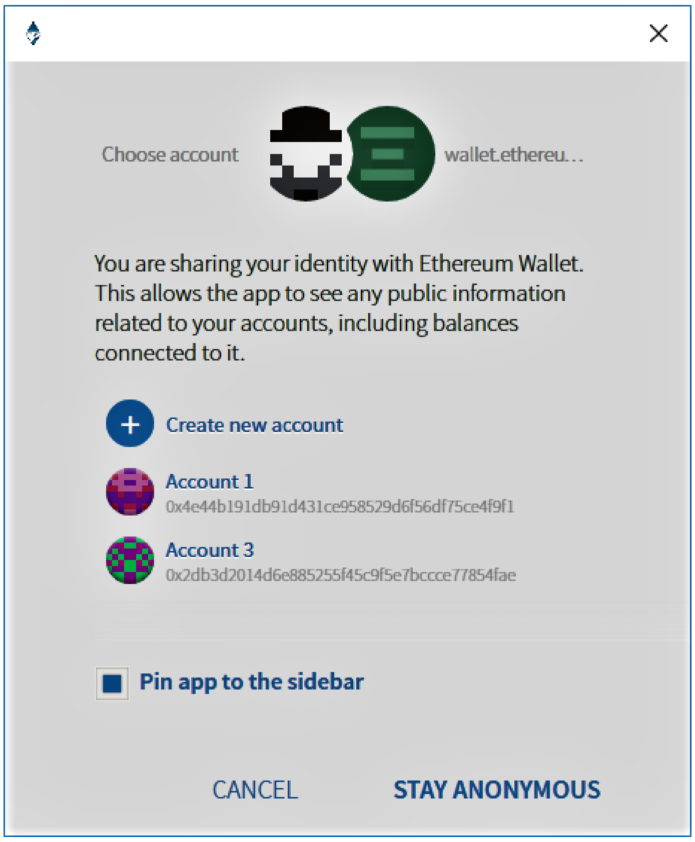
WALLET CONTRACTS

Once you have more than 1 ether you can create a more sophisticated contracts. Wallets are one type of basic smart contract, that allows your funds to be controlled by multiple accounts and have an optional daily limit on withdrawals, to increase security. To create your own custom contracts go to the *Contracts* tab.

NO TRANSACTIONS YET

Figure 2.9: Ethereum Wallet Dapp [Eth17b]

1. Click add account button shown in figure 2.9. The figure 2.8 show wallet Dapp can also be load in Mist browser using Dapp URI and perform the same actions.
2. Then click create a new account button shown in figure 2.10 and feed the password that's it, a new account is added.



You are sharing your identity with Ethereum Wallet. This allows the app to see any public information related to your accounts, including balances connected to it.

Create new account
Account 1
0x4e44b191db91d431ce958529d6f56df75ce4f9f1
Account 3
0x2db3d2014d6e885255f45c9f5e7bccce77854fae
Pin app to the sidebar
CANCEL
STAY ANONYMOUS

Figure 2.10: Ethereum wallet create new account

Using Eth

Eth provide the same options for key management like geth and can be used the same way as shown in listing 2.3.

Listing 2.3 Account creation using eth [Eth17b]

```
    > eth account list // List all keys available in wallet.
    > eth account new // Create a new key and add it to the wallet.
    > eth account update [<uuid>|<address> , ... ] // Decrypt and re-encrypt given keys.
    > eth account import [<uuid>|<file>|<secret-hex>] // Import keys from given source and
        place in wallet.
```

2.4.3 Account Update

An existing account is updated to upgrade the keyfile to latest keyfile format or upgrade keyfile password. The update is performed through "update" subcommand of geth, using the command line with the account address or index number as the parameter. An important thing to note here is that never rely on the index number because it gets changed.

Listing 2.4 Account Update [Eth17b]

```
$ geth account update a94f5374fce5edbc8e2a8697c15331677e6ebf0b

Unlocking account a94f5374fce5edbc8e2a8697c15331677e6ebf0b | Attempt 1/3
Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
account 'a94f5374fce5edbc8e2a8697c15331677e6ebf0b' unlocked.
Please give a new password. Do not forget this password.
Passphrase:
Repeat Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
```

2.4.4 Account backup and restore

Account Keyfile is the essential requirement to sign transaction from that account otherwise account is considered as unknown. All the keyfiles are in keystore subdirectory of Ethereum node's data directory. The default location on different platforms is

1. Windows: C:%appdata
2. Linux: /.ethereum/keystore
3. Mac: /Library/Ethereum/keystore

If backup or restoration is to be done manually, then just copy keyfiles from or to the keystore. Geth supports importing an unencrypted private key . Geth subcommand "geth account import /path/<key.json>" is used to import the key. In case if a custom data directory is used then the command is executed in a way mentioned in listing 2.5. After import operation, check the accounts ordering because the ordering is based on the timestamp due to which it may get changed. So it is advised to double check indexes before using it.

Listing 2.5 Importing an Unencrypted private key [Eth17b]

```
$ geth --datadir /someOtherEthDataDir account import ./key.prv
The new account will be encrypted with a passphrase.
Please enter a passphrase now.
Passphrase:
Repeat Passphrase:
Address: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

2.5 Ether

Name of the currency used in Ethereum is called ether. All the computation charges are paid in ether within the EVM. This is done implicitly by purchasing gas with ether. There is a metric system of denominations in Ehtereum used as units of ether. Every denomination has its unique name. The most basic denomination of ether is called Wei. There are some other denominations with their values in Wei are list in the table 2.2

Unit	Wei Value	Wei
wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

Table 2.2: Denominations of ether and its value in Wie [Eth17b]

Getting ether

Ether can be obtained in the following ways

- To become an Ethereum miner.
- Trade other currencies for ether using centralized or trustless services.
- Utilize Mist Ethereum GUI Wallet that as of Beta 6 introduced the ability to purchase ether using the http://shapeshift.io/ API.

Sending Ether

Ether can be sent through Ethereum wallet and Metamask (its a browser plugin which enable a normal browser to run Dapps). Geth console can also be used for the ether transfer from one account to another as shown in the listing 2.6.

Listing 2.6 Using Geth console to transfer ether [Eth17b]

```
> var sender = eth.accounts[0];
> var receiver = eth.accounts[1];
> var amount = web3.toWei(0.01, "ether")
> eth.sendTransaction({from:sender, to:receiver, value: amount})
```

2.6 Ethereum Network

The P2P network of participating nodes which provides a base for decentralized consensus maintain and secure the blockchain. The dashboard which contains all the information related to network stats like current block, hash difficulty, gas price, and gas spending is called EthStats.net. Anyone can add a node to this dashboard [Eth17b].

2.6.1 Ethereum blockchain Types

Based on the permissions such as read and write, the participation of in the network, being a part of the consensus process, The Ethereum blockchain is divided into three types which are.

Public blockchain

The major Blockchain, which anyone from anywhere in the world can read, and can send the transaction to and expect to be added in the block after validation in the consensus process. Consensus process decides which blocks to be added to the chain and determine the current state of the chain. Blockchains are secured by crypto-economics, which means the combination of cryptographic verification and economic incentives using PoW or PoS, which follows a general principle that the weight of influence someone carries in the consensus process is equivalent to the number of economic resources that they can offer.

Consortium blockchain

The blockchain in which consensus process is limited to pre-selected nodes. Let suppose we have a consortium of fifteen financial institutions, every one of which operates a node. To make a valid block and to add it to the chain, it must be signed by ten institutions. Reading from blockchain may be public or restricted to the participants. There are hybrid routes like root hashes of the blocks being public together with an Application program interface (API) that permits public members to perform a limited number of queries and take back partial cryptographic proofs of the blockchain. This sort of blockchain is known as partially decentralized.

Private blockchain

In this type of blockchain write permission are centralized, belongs to one organization. Read permissions may be public or limited to an arbitrary amount. It belongs to a single company; possible applications contain database management system, auditing, etc. In some cases, public readability is desired while in some not at all.

2.6.2 Connecting node to network

Geth in a network continuously try to connect to other nodes in the network until it has peers. If UPnP is enabled on the router or by running Ethereum on an Internet-facing server, it will also allow connection from other nodes. Geth discovers peers through something called the discovery protocol. To find other nodes in the network geth use discovery protocol. In discovery protocol, nodes communicate with each other to find out about other nodes on the network. Initially, geth utilizes a set of bootstrap nodes whose endpoints are stored in the source code. We can use admin API to perform node related operations like getting node info, listing all peers, an addition of a new peer to the network. To check the node Information we use geth console to execute "admin.nodeInfo" command as given below.

Listing 2.7 Node Information [Eth17b]

```
> admin.nodeInfo
{
      Name: 'Geth/v0.9.14/darwin/go1.4.2',
      NodeUrl: 'enode://3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60
      f14998a3a98 c0cf14915eabfdacf914a92b27a01769de18fa2d049dbf4c17694@[::]:30303',
      NodeID: '3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60f14998a3a
      98c0cf1491 5eabfdacf914a92b27a01769de18fa2d049dbf4c17694',
      IP: '::',
      DiscPort: 30303,
      TCPPort: 30303,
      Td: '2044952618444',
      ListenAddr: '[::]:30303'
}
```

To check the list of peers use "admin.peers" command as given below. Static node can be

Listing 2.8 Peers list [Eth17b]

```
> admin.peers
[{
      ID: 'a4de274d3a159e10c2c9a68c326511236381b84c9ec52e72ad732eb0b2b1a227793
      8f78593cdbe734e6002bf23114d434a085d260514ab336d4acdc312db671b',
      Name: 'Geth/v0.9.14/linux/go1.4.2',
      Caps: 'eth/60',
      RemoteAddress: '5.9.150.40:30301',
      LocalAddress: '192.168.0.28:39219'
}, {
      ID: 'a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69
      ad0dce72a4d8db5ebb4968de0e3bec910127f134779fbcb0cb6d3331163c',
      Name: 'Geth/v0.9.15/linux/go1.4.2',
      Caps: 'eth/60',
      RemoteAddress: '52.16.188.185:30303',
      LocalAddress: '192.168.0.28:50995'
}]
```

added at runtime using the Javascript console by executing "admin.addPeer()" command as given below.

Listing 2.9 Adding Peer [Eth17b]

```
> admin.addPeer("enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768
e296f5e4a54e8b995de102e0ceae2e826f293c481b5325f89be6d207b003382e18a8e
cba66fbaf6416c0@33.4.2.1:30303")
```

2.7 Mining in Blockchain

The term mining is used in analogy to gold mining for crypto-currencies. Gold or any other precious metal are acquired through mining, similarly is the case for crypto-currencies. Ethereum has only one way to issue post-launch that is through mining. The mining process is also responsible for securing the network by verifying, creating, publishing, and propagating blocks in the blockchain [Eth17a; Eth17b; WOO17].

Mining ether = Securing the Network = Verifying Computation

Ethereum uses incentive-driven model of security. Ethereum consensus is based on choosing the block with the highest total difficulty. Miners are responsible for block creation, and other nodes check the validity. The well-formedness of a block also depends on whether the block contains PoW of a given difficulty otherwise, it is not considered valid. In the next version of Ethereum PoW will be replaced by PoS.

The algorithm of PoW used is called Ethash. It concerns in finding a nonce input to the algorithm so that the result is under a certain difficulty threshold. The critical point in PoW algorithms is that there is no better approach to find such a nonce than counting the possibitlites, Since outputs have a uniform distribution (hash application results), it can be guaranteed that on average the required time to find such a nonce depends on the difficulty threshold. This makes it feasible to control the time-span of finding a new block by managing time.

The difficulty can be adjusted dynamically as directed by the protocol. Due to which the entire network create one block every fifteen seconds. In blockchain terminology called blockchain with fifteen seconds block time. This "heartbeat" essentially punctuates the synchronization of system state and guarantees that maintaining a fork(which allow double spend attack) or editing history by an attacker is impossible, unless if the attacker owns at least 51 percent of mining power.

2.7.1 Ethash Mining Algorithm

Mining algorithm, Ethash(previously known as Dagger-Hashimoto) is based around the provisioning of a large, transient, randomly produced dataset which constitutes a DAG (the Dagger-part), and trying to solve a particular limitation on it, partly determined with a block's header-hash.

The algorithm is designed to hash with a fast verifiability time within a slow CPU only environment, But if large memory and high bandwidth is given, it can provide high speed-ups for mining. The large memory needs mean that large-scale miners receive relatively little super-linear benefit. The high bandwidth requirement suggests that a speed-up from gathering on many super-fast processing units sharing the same memory provides a slight advantage over a single unit. That is important in that pool mining have no benefit for nodes doing verification, thus hindering centralization. Communication within the external mining application and the Ethereum daemon for work provision and submission occurs over the Javascript Object Notation (JSON)-Remote Pocedure Call (RPC) API. At least one ethereum account and a fully synced Ethereum client that is enabled is needed to mine. The account is used to send the mining compensations to and usually referred as coinbase or etherbase.

2.7.2 Ethash DAG

The PoW algorithm Ethash uses DAG, which is generated for each epoch such as every 30000 blocks (125 hours). The DAG takes long time span to generate . On-demand generation by a client will face long waiting time at each epoch transition before the first block of the recent epoch is found. DAG is dependent only on the block number, so it can be calculated in advance to avoid long delays at each epoch transitions. Geth and ethminer realize automatic DAG generation and maintains two DAGs at a time for soft epoch transitions. Automatic DAG generation can be turned on or off when mining is managed from the console. It can be turned on at launch time using –mine flag. If multiple instances of any client are running, make sure automatic dag generation is turned off in all but one instance. To generate the DAG for an arbitrary epoch:

Listing 2.10 Arbitrary DAG generation [Eth17b]

```
$ geth makedag <block number> <outputdir>
```

2.7.3 CPU Mining

CPU mining can be used to mine ether. But as GPU mining is almost two orders of magnitude more productive, that's why CPU mining is no more profitable. However, it can be used on the testnet or private chain to generate ether for testing contracts and transactions without going to live network and spend real ether. Starting ethereum node with geth does not automatically start mining, you have to explicitly give the command at launch time in terminal or geth console at runtime.The important thing to remember is that before starting miner etherbase(coinbase) address has to be set otherwise miner will not start. The listing 2.11 shows both launch time and runtime command to manipulate the CPU miner. Sometimes it happens frequently that one gets a block which never makes to the accepted chain. It happens because a miner mines a block locally and the state shows a reward for it in its etherbase, but after a sometimes, a better chain is discovered on the network and switch to that chain on which the mined block from a particular miner is not

Listing 2.11 CPU Miner manipulation commands [Eth17b]

```
      Launch time commands
            // number of threads your want to use , default is the number of processor cores.
            $ geth --mine --minerthreads= number of threads
            // setting etherbase command
         $ geth --etherbase '0xa4d8e9cae4d04b093aac82e6cd355b6b963fb7ff'

    Runtime commands
         // starting and stoping miner in console
         > miner.start(8)
         true
         > miner.stop()
         true
         // to set etherbase account
         miner.setEtherbase(eth.accounts[indexnumber of account])
         // to check miner hashrate
         > miner.hashrate
            712000
         // To check etherbase account balance
         > eth.getBalance(eth.coinbase).toNumber();
         '34698870000000'
         // coinbase unloking
         > personal.unlockAccount(eth.coinbase)
            Password
         true
```

included, so no reward is credited. It's normal if coinbase balance fluctuates. To check which blocks are mined by a particular miner (address), the code in listing 2.12 can be used in geth console.

Listing 2.12 Checking miner mined Blocks [Eth17b]

```
function minedBlocks(lastn, addr) {
      addrs = [];
      if (!addr) {
            addr = eth.coinbase
      }
      limit = eth.blockNumber - lastn
      for (i = eth.blockNumber; i >= limit; i--) {
            if (eth.getBlock(i).miner == addr) {
                  addrs.push(i)
            }
      }
      return addrs
}
// scans the last 1000 blocks and returns the blocknumbers of blocks mined by your
    coinbase
// (more precisely blocks the mining reward for which is sent to your coinbase).
minedBlocks(1000, eth.coinbase);
//[352708, 352655, 352559]
```

2.7.4 GPU Mining

For GPU mining, DAG need minimum 1-2Giga Bytes (GB) of Random Access Memory (RAM) . If there are error in GPU mining. GPU memory framgmenation, simply means there is not enough memory. To start mining on single GPU, exeute the command mention in listing 2.13

Listing 2.13 GPU Mining with single GPU [Eth17b]

```
eth -v 1 -a 0xcadb3223d4eebcaa7b40ec5722967ced01cfc8f2 --client-name "OPTIONALNAMEHERE"
    -x 50 -m on -G
```

1. -v 1 Set verbosity to 1.
2. -a Set the coinbase, where the mining rewards will be collected.
3. –client-name "OPTIONAL" Set an optional client name to identify you on the network.
4. -x 50 Request a high amount of peers. Helps with finding peers in the beginning.
5. -m on Turn mining on at launch time.
6. -G set GPU mining on.

2.7.5 Pool Mining

Mining pools are cooperatives that aim to ease out expected income by combining the mining power of participating miners. In return, they mostly charge individual miners 0-5 percent of mining their rewards. The mining pool uses a central account to submit PoW and distribute the compensation to participants according to their contributed mining power. Most mining pools are the third party, centralize, they can run away with the participant's earnings, but there are decentralize pools available with open source codebase. The mining pool does not validate blocks or check state transitions; they only outsource PoW calculations. Regarding security pools behave like single node, their growth poses a centralization risk of 51 percent attack. So it is strongly advised to keep network capacity distribution in check and does not allow pools to grow too large.

2.8 Transaction, Message, and Gas

Transaction, Message, and Gas are the Most frequent words used in blockchain technology. This section gives a detail overview of all terms [Eth17a; Eth17b].

2.8.1 Transaction

Transaction in ethereum refers to the signed data package that contains a message to be transmitted from an EOA to another account on the blockchain. Transactions contains the following parameters

1. The receiver of the message.
2. Signature of the sender, which determine the purpose of the sender to send the message through blockchain to the receiver.
3. The value field, which carries the amount of wei to transfer from the sender to the receiver.
4. Data field, which is optional, contains the message sent to a contract.
5. STARTGAS value, which represents the maximum number of computational steps permitted to be executed by the transaction.
6. GASPRICE value, which represents the fee, the sender is ready to pay. One unit of gas is equivalent to the execution of one atomic instruction such as computational step.

2.8.2 Message

Messages are virtual objects that exist only in ethereum execution environment and never serialized. Messages are transferred from one contract to another. They can be considered as function calls. The message is similar to transaction except it is generated by contract and transaction is generated by an external actor. Message get produced when currently running contract execute the CALL or DELEGATECALL opcodes, which generate a message [Eth17a; Eth17b]. A message contains

1. The sender of the message (implicit).
2. The receiver of the message.
3. The VALUE field, which contains the amount of wei to transfer besides the message from one contract to another contract address.
4. Data field, which is optional, contains the actual input data sent to a contract.
5. STARTGAS value, which restricts the maximum amount of gas the code execution triggered by the message can acquire.

2.8.3 Gas

Ethereum execution environment on the blockchain is called EVM. Every node on the network run EVM as a component of block verification protocol. They check all the transaction in the block and run the code trigger by a transaction. Ethereum executes the same code on all full nodes and stores the same values, that's why ethereum is computationally efficient, but it is a useful way to reach consensus on system state without relying on third parties. The redundant computation makes them expensive and makes it unsuitable for computation which can be done offchain [Eth17a; Eth17b; WOO17].

The name of the charges the sender pay for the execution of transactions is called Gas. This act as crypto-fuel, driving the motion of smart contracts. Gas is purchased implicitly from the miners that execute the code. Ether and gas are decoupled because units of gas align with computation units having natural cost, while the price of ether fluctuates as a result of market forces. The price of gas is decided by the miners, who has the right to refuse the processing of a transaction with lower gas price than their minimum limit. The purchase process is implicit; the user only specifies the maximum limit. Ethereum protocol uses pay per computational step model. Every step execution on blockchain cost something, which prevents attacks and resource exploitation on the ethereum network. Every transaction must have to include gas limit and a fee that it is willing to pay. If the number of computational steps and all other charges does not exceed the gas limit, then the transaction is processed if it exceeds all changes are reverted except the transaction is still valid and the fee will be collected by miners. It is good practice to keep gas limit high because all extra gas is reimbursed to the user account, so there is no risk of overspending, charges deduction are consumption based. Table 2.3 show gas cost for different operation at EVM.

Total transaction cost = gasUsed * gasPrice [Eth17b]

2.9 Smart Contract

A smart contract in ethereum is a collection of code (its functions) and data (its state) that reside on ethereum blockchain at a unique address. A contract is also called internal account or contract account. A contract lives on blockchain in the ethereum-specific binary format called EVM bytecode. Contract account can send a message to another contract and perform Turing complete computation. A contract can be written in high-level languages like solidity, and serpent [Eth17b].

Solidity is a statically typed contract oriented high-level language whose syntax is similar to that of JavaScript. It supports inheritance, libraries and very complex user-defined types and many other functionalities. Solidity is designed to target EVM To dive into solidity, check the complete documentation with practical examples at [Tea17b].

Operation Name	Gas Cost	Remark
step	1	default amount per execution cycle
stop	0	free
suicide	0	free
sha3	20	-
sload	20	get from permanent storage
sstore	100	put into permanent storage
balance	20	-
create	100	contract creation
call	20	initiating a read-only call
memory	1	every additional word when expanding memory
txdata	5	every byte of data or code for a transaction
transaction	500	base fee transaction
contract creation	53000	changed in homestead from 21000

Table 2.3: EVM operation Gas Cost [Eth17b]

2.9.1 Contract Compilation

There are different ways of contract compilation, but here only one is explained which is easy to use and very helpful for testing contract. To compile the contract open geth console and execute the commands as shown in the listing 2.14. The compiler output contract object which contain list of field which are given below

code The compiled EVM bytecode.

info Additional metadata output from the compiler.

source The contract source code.

language Source code language such as Solidity or Serpent.

languageVersion The Contract source code language version.

compilerVersion The solidity compiler version that was used to compile this contract.

abiDefinition The Application Binary Interface Definition.

userDoc The NatSpec Doc for users.

developerDoc The NatSpec Doc for developers.

Listing 2.14 Solidity Contract compilation in geth JavaScript Console [Eth17b]

```
// first assign contract source code to variable
> source = "contract test { function multiply(uint a) returns(uint d) { return a * 7; }
    }"
// now run the compilation script
> contract = eth.compile.solidity(source).test
{
      code: ’605280600c6000396000f3006000357c010000000000000000000000000000
      000000000000000000000000000090048063c6888fa114602e57005b6037600435604
      1565b8060005260206000f35b60006007820 29050604d565b91905056’,
      info: {
            language: ’Solidity’,
            languageVersion: ’0’,
            compilerVersion: ’0.9.13’,
            abiDefinition: [{
                  constant: false,
                  inputs: [{
                        name: ’a’,
                        type: ’uint256’
                  } ],
                  name: ’multiply’,
                  outputs: [{
                        name: ’d’,
                        type: ’uint256’
                  } ],
                  type: ’function’
            } ],
            userDoc: {
                  methods: {
                  }
            },
            developerDoc: {
                  methods: {
                  }
            },
            source: ’contract test { function multiply(uint a) returns(uint d) { return
                a * 7; } }’
      }
}
```

2.9.2 Contract Deployment

To deploy a contract to the blockchain, the first step is to unlock the account from which contract is going to be deployed and keep some ether in that account. Then run the script given listing 2.15.

Listing 2.15 Contract deployement using geth JavaScript console [Eth17b]

```
var primaryAddress = eth.accounts[0]
var abi = [{ constant: false, inputs: { name: 'a', type: 'uint256' } }]
var MyContract = eth.contract(abi)
// arg1, arg2 may be more, are used for constructor of the contract, can be omitted if
    not required.
var contract = MyContract.new(arg1, arg2, ..., {from: primaryAddress, data:
    evmByteCodeFromPreviousSection})
```

2.10 Dapp development Tools and Technologies

In ethereum Dapps are the applications which enable a user to interact with blockchain with a user-friendly GUI. The business logic of Dapps are the smart contracts, and the front-end comprises of technologies like HTML, Javasrcipt, etc. As contract code is redundant and expensive to execute, so Dapps are designed to perform only necessary computation on the blockchain, and all other which can be performed by the clientside are executed on the client machine to avoid unnecessary cost and utilization of blockchain resources [Eth17b]. An overview of some of the Dapps technologies is given below

2.10.1 Web3.js

Ethereum node provide RPC interface. The interface provides access to Dapps to interact with blockchain and utilize the functionalities provide by blockchain. But communication using JSON-RPC interface is very error prone especially when working with Application Binary Interface (ABI). Web3.js is a javascript library which provides an abstraction layer and works on top of ethereum RPC. It is user -friendly and less error-prone. Web3 includes the eth object - web3.eth (for specifically Ethereum blockchain interactions) and the shh object - web3.shh (for Whisper interaction)[tea17].

New developments are underway in addition to blockchain which is an effort to decentralize another aspect of a web application like storage, application-to-application communication. This is an effort towards the distributed web, in which every component of web applications will be decentralized like blockchain, and most of these services will be accessible through web3 API as shown in figure 2.11. Most of the new services support in web3 will be available in the new version 1.0, which is not currently released as per official statement, currently some features are missing like Swarm support.

Swarm is the p2p storage sharing network, which wil be used to store the file contents. In swarm files are addressed by the hash of their content. It is possible to fetch from multiple nodes at the same moment of time. As long as one system has host the data, it will be accessible all over the network.

Whisper is a communication protocol for Dapps to communicate with each other. It's a low-level API only exposed to Dapps, not users, designed for small data transfer, does not support real-time communication because of uncertain latency, and its dark with no reliable method for packet tracking [Tea16].

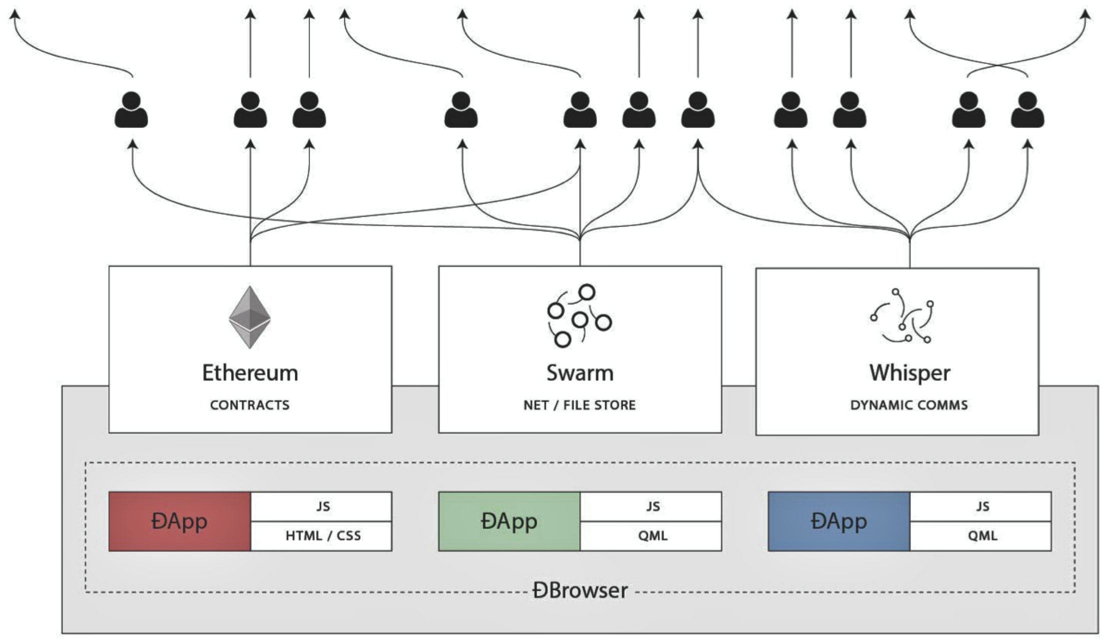


Figure 2.11: Web3 base Layer Services [Eth17b]

2.10.2 Truffle Framework

Truffle is a development environment, testing framework and asset pipeline for Ethereum, trying to make life as an Ethereum developer easier. In this thesis, the truffle is used as a development framework for contract compilation, deployment, etc. Truffle provides the following services [Cou17a].

1. Built-in smart contract compilation, linking, deployment and binary management.
2. Automated contract testing for rapid development.
3. Scriptable, extensible deployment and migrations framework.
4. Network management for deploying to any number of public and private networks.
5. Package management with EthPM and NPM, using the ERC190 standard.
6. Interactive console for direct contract communication.
7. Configurable build pipeline with support for tight integration.
8. External script runner that executes scripts within a Truffle environment.

2.10.3 Truffle-contract

Truffle-contract provides services beyond web3.js and contains better contract abstraction. Some of the prominent features are given below [Cou17b].

1. Synchronized transactions for better control flow (i.e., transactions won't finish until you're guaranteed they've been mined).
2. Promises. No more callback hell. Works well with ES6 and async/await.
3. Default values for transactions, like from address or gas.
4. Returning logs, transaction receipt and transaction hash of every synchronized transaction.

To set up a new web3 provider instance and initialize contract using truffle-contract and execute a function in a contract.The following listing show sample code to perform such activities.

Listing 2.16 Contract Initialization with truffle-contract [Cou17b]

```
var provider = new Web3.providers.HttpProvider("http://localhost:8545");
var contract = require("truffle-contract");
var MyContract = contract({
      abi: ...,
      unlinked_binary: ...,
      address: ..., // optional
      // many more
})
MyContract.setProvider(provider);
var deployed;
MyContract.deployed().then(function(instance) {
      var deployed = instance;
      return instance.someFunction(5);
      }).then(function(result) {
            // Do something with the result or continue with more transactions.
});
```

2.10.4 Bootstrap Framework

Bootstrap is an open source front-end web application development framework. It contains HTML and CSS based design templates for typography, buttons, forms, navigation, and some other components. The bootstrap uses a Grid system for Responsive layout. Which comes with standard 1170 pixel wide grid layout. Alternatively, the developer can also use a variable-width layout. For both cases, the toolkit has four variations to make use of various resolutions and classes of devices such as mobile phones, portrait, and landscape, tablets and PCs with low and high resolution. Each variation fits with the width of the columns [Mar17; Wik17a].

JQuery is a cross-platform, lightweight JavaScript library designed to simplify the client-side scripting. It is free, open-source software. jQuery's syntax is designed to make it simpler to navigate a document, select Document Object Model (DOM) elements, create animations, handle events, and develop Asynchronous JavaScript and XML (AJAX) applications. JQuery enables developers to build plug-ins on top of the JavaScript library and allows them to build abstractions for low-level interaction and animation, advanced effects and high-level, themeable widgets. The modular approach to the jQuery library supports the creation of powerful dynamic web pages and Web applications [Wik17b].Bootstrap JavaScript component is alos denpendent on jQuery so it has to be linked before using bootstrap JavaScript component [Mar17].

CSS is a style sheet language used for defining the presentation of a document written in a markup language. Mostly used with HTML for the interface design, the language can be applied to any XML document, including plain Extensible Markup Language (XML), Scalable Vector Graphics (SVG), etc and applies to rendering in speech, or on other media. In combination with HTML and JavaScript CSS can be used to design user friendly interfaces for various platforms.The primary objective of CSS was to separate the presentation and content, including features such as the layout, colors, and fonts. The separation of formatting and content makes it possible to present the same markup page in various styles for varying rendering methods, such as on-screen, in print, by voice (through a speech-based browser or screen reader), and on Braille-based tactile devices. It can also present the web page differently depending on the screen size or viewing device. Readers can also specify a different style sheet, such as a CSS file stored on their computer, to override the one the author defined [Mar17; Wik17c].

HTML is the language for describing the structure of Web pages. HTML gives authors the means to [W3C16]:

1. Publish online documents with headings, text, tables, lists, photos, etc.
2. Retrieve online information via hypertext links, at the click of a button.
3. Design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
4. Include spread-sheets, video clips, sound clips, and other applications directly in their documents.
5. With HTML, authors describe the structure of pages using markup. The elements of the language label pieces of content such as “paragraph,” “list,” “table,” and so on.

3 Related Works

In this chapter, some related research will be presented which are based on blockchain use for IoT. Most of these research contents are very latest, publish around mid-2017, and majority of them present concept with possible scenarios such as "Blockchains everywhere - a use-case of blockchains in the pharma supply-chain" [BRSS17], "Towards Better Availability and Accountability for IoT Updates by Means of a Blockchain" [BBG+17], etc. An overview of all of these studies will be presented based on their motivation for the work and implementation approach.

3.1 Blockchains everywhere - a use-case of blockchains in the pharma supply-chain

Modum.io AG is the startup company working on blockchain and IoT based solution for pharmaceutical supply-chain to provide public access to immutable temperature records [BRSS17].

3.1.1 Motivation

The author in [BRSS17] mentioned that the primary benefit of using a blockchain with smart contracts is that these contracts can be assessed automatically. Using smart contracts, the temperatures can be assessed automatically and notify sender and recipient. Moreover, the stored data is tamper-proof and can be used for audits by External parties to ensure Good Distribution Practice of medical products. With Ethereum, such a tamper-proof entirely a decentralized system can be used at a low cost and on a per contract and per-byte basis. As multiple stakeholders are involved in the supply chain, the use of blockchain technology can be used to automate processes and eventually save costs by ensuring trust between the involved parties.

3.1.2 Implementation Approach

The approach presented in [BRSS17] is a hybrid one. It uses blockchain alongside traditional central server and database dependent approach. So the application is not entirely decentralized, part of it is decentralized as given in figure 3.1 . The key components of

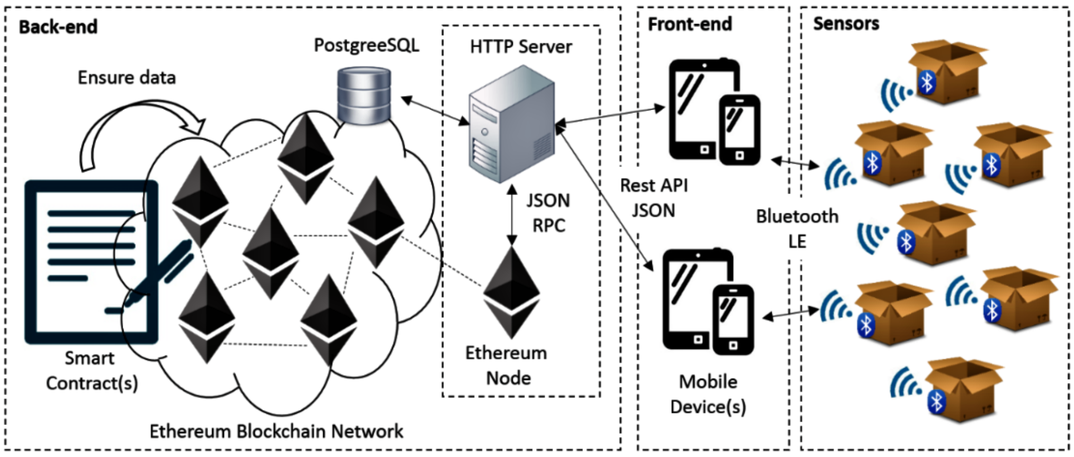


Figure 3.1: Modum.io AG supply-chain Architecture [BRSS17]

this system are Ethereum Blockchain Network, Smart contract, Database, Server, Mobile Devices, and Sensors.

Temperature is insured by smart contracts written in solidity in the back-end. For every new shipment or group of pharmaceutical product including distinct temperature requirements, a smart contract is configured and deployed in the server-side to ensure the Good Distribution Practice (GDP) compliance requirements. Therefore, the mapping from the shipment to its corresponding smart contract or address of the contract is made using a relational database with very low additional complexity or cost. The server at modum.io AG hosts an Ethereum node that participates in the Ethereum network and can watch changes on its smart contracts, create new smart contracts, or call smart contract functions. The Ethereum node communicates with the Hyper Text Transfer Protocol (HTTP) server over JSON . Data that is sensitive or too large to store in the blockchain is stored in a PostgreSQL2 database. This incorporates the raw temperature data, as these are too large To be stored in a smart contract. The smart contract verifies the temperature range and stores the verification result in the smart contract together with a Universal Resource locater (URL) that point to the raw temperature data and its hash [BRSS17].

Android clients in the front-end communicate with the server using Representational State Transfer Protocol (REST) API using JSON to encode and decode requests/responses. Through a mobile phone,users can register new shipments including administrative details within the system and that a smart contract is created for every shipment. The API should also allow the recipient of a shipment to upload the temperature measurements reported by the sensor to the server. Both the sender and the recipient should be informed of the result of the contract and be able to get access to the temperature measurements, preferably using a graphic visualization.

3.2 Towards Better Availability and Accountability for IoT Updates by means of a Blockchain

IoT requires deploying a large number of devices with full or limited connectivity to the Internet. If these devices are exposed to attackers and not secured-by-design. It is necessary to be capable of updating them, to cover their vulnerabilities and to deter hackers from enrolling them into botnets [BBG+17].

3.2.1 Motivation

In [BBG+17] a solution is presented to securely update IoT devices which are vulnerable to attacks. To avoid attacks and fix these vulnerabilities, the update platform design must ideally implement confidentiality, integrity, and availability. An investigation is performed to achieve such functionalities using blockchain platform. Every IoT device shows specific vulnerabilities to cyber attacks due to several factors including longevity, lack physical protection, hardware weaknesses or stripped down Human Machine Interface. Recent attacks or proof of concept attacks have highlighted how these factors combine with or complicate implementation designs. The mass deployment and the vulnerability property of IoT devices, needs for comprehensive software update infrastructures enabling IoT product manufacturers/integrators to maintain software-based devices remotely. Indeed the best-conceived software update platforms can fall short if they are intentionally targeted as part of a combined attack scenario.

3.2.2 Implementation Approach

The key components are a web portal, blockchain infrastructure based on Multichain, and IoT devices as given in figure 3.2. The web portal enables manufacturers to deploy software updates securely and efficiently. The web portal has access to the blockchain infrastructure, which is shared between manufacturers. Each manufacturer is required to provide at least one worker node to improve the availability and the computing power of the infrastructure. For the prototype, blockchain nodes are implemented as Virtual Machine (VM)s and are hosted on a XenServer server. The web portal and the IoT devices can exchange software updates and confirmations, using the blockchain infrastructure. The system relies on asymmetric cryptography to guarantee data confidentiality and integrity. The IoT devices are implemented either (i) physically by means of development boards, such as Raspberry Pi, that are rather close to real field devices, or (ii) virtually, by means of Qemu virtual machines, to evaluate the scalability of the prototype [BBG+17].

To push the software update, Manufacturer first signs into the portal, then select the device to update and then upload software update package along with metadata. Then choose one of these two options (i) only sign the update, (ii) sign and encrypt it. The benefit of the first method is that only one unencrypted file will be pushed to the blockchain, while the second method guarantees confidentiality. However, it requires pushing as many files as there are devices. However, An assumption is taken that the decryption/secret key is specific to each device. The blockchain infrastructure then validates the transaction(s) within seconds. While each IoT device regularly connects to the blockchain and checks whether a new update is available for download. If so, the IoT device downloads the update and installs it. Eventually, it sends a response to the blockchain to keep track of up-to-date devices [BBG+17].

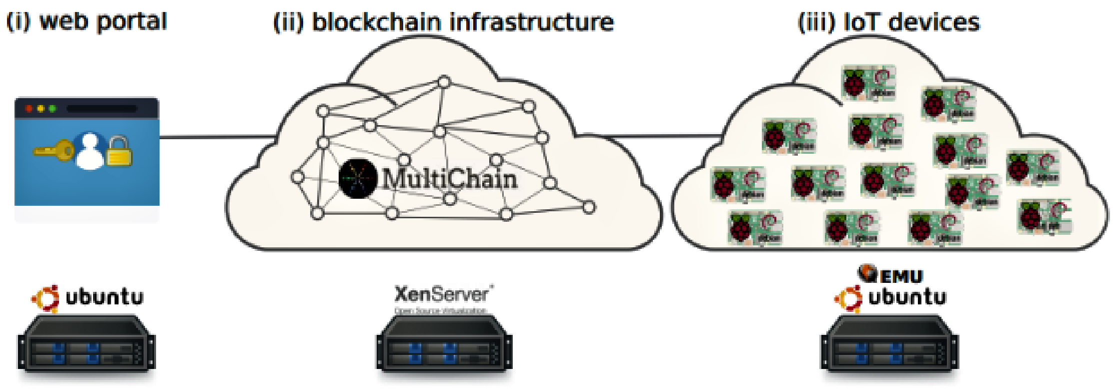


Figure 3.2: IoT Updates by means of a Blockchain, Prototype Architecture [BBG+17]

3.3 CONNECT: CONtextual NamE disCovery for blockchain-based services in the IoT

How to discover things or services in IoT is the primary research question raised by [DPKS17] and try to provide a unified solution to the discovery process based on the blockchain.

3.3.1 Motivation

The current solution for IoT discovery problem and to build a truly interconnected world focus on the definition of new naming standards and are intended at the process of “labeling” devices. This approach is currently used in the classic web scenario where Domain Name Service (DNS) is used to map human-readable Universal Resource Identifier (URI)s to Internet Protocol (IP) addresses. As a result, URIs can be used or coded in applications, scripts, and software While actual IP addresses can vary over time. However, porting this approach to the IoT will push manufacturers to alter their identifiers (such as serial numbers) and it will remain focused on the data source (i.e., the devices) rather than on the data itself (i.e., the services). Moreover, IoT manufacturers and administrators could be other entities rather than the service providers.To solve the discovery problem [DPKS17] present a solution which leverage blockchain technology, by building a new contextual scheme capable of identifying a service and if required later, it will also identify the device that runs the service.

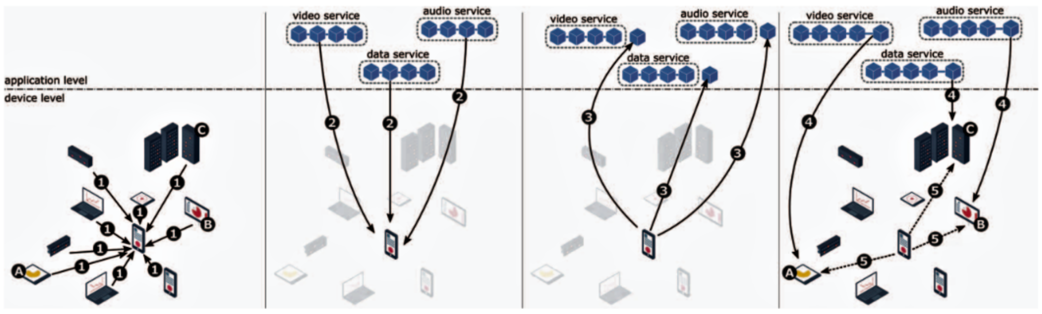


Figure 3.3: Blockchain based discovery process [DPKS17]

3.3.2 Implementation Approach

The solution presented by [DPKS17] is called CONNECT. It defined Two different kinds of nodes (i) Virtual nodes or VNodes, are logical nodes in the form of blockchain peers. These nodes are responsible for handling all the operations that involve the blockchain such as creating and validating transactions as browsing the blockchain to find information among other peers. (ii) Physical Nodes or PNodes represent the devices. Each VNode belongs to a layer which describes a particular blockchain application (might be data, audio and video services) as shown in figure 3.3. So, each audio, video or data request will be handled by a blockchain which is located at a certain point on the blockchain tree depending on the service provider. The following three possible interactions for PNodes and VNodes are :

- PNode to PNode: the connection between two physical devices in the network.
- PNode to VNode: the connection between a physical device and a blockchain peer who can be either executed by the same PNode or by others. Devices are locally running virtual nodes already have all their information. Otherwise, they need to reach the other the physical node running the peer and download all its information.
- VNode to VNode: the connection between two peers, for the sake of simplicity and clarity, called VNodes and VNoder for sender and receiver respectively. If the VNodes belong to the same blockchain, the PNode hosting them just browses the chain and finds the required information. If the VNoder belongs to a different blockchain than VNodes, then the PNode hosting it has to i) download the VNode blockchain from the client and ii) browse it for the required information.

The author describes discovery process steps in [DPKS17] which are:

1. Initially, a hello message is sent to all the devices in the network. If the services required are already known, then it can also be specified. Otherwise, send an empty hello message which will cause all the surrounding devices to respond with the

information on all their services. The replies from surrounding devices contain the blockchain addresses of the peers, which are providing the demanded services on top of the blockchain.

2. Once received the blockchain peer addresses, browsing can be started, the relevant blockchains in the cloud to find the last activities of those addresses.looking for a proof at this step, within the blockchains that supports what the devices claimed by responses to the hello message. In the example depicted in figure 3.3, let suppose that the A, B and C devices respond to the hello message with some information about the video, audio and data services.
3. Communities which have verified and validated blockchain transactions from A, B, and C, are trusted, now create three new transactions (one for each service on a different blockchain) in which specify the conditions to access the service, and pay for them.
4. Once the transaction is verified and validated by the other peers within the blockchains, A, B, and C will be notified thus allow the access to the paid service.
5. (Optional) if Communication through blockchain is not possible, physical layer can be used to communicate. However, it has to be stressed that, this last and optional step is accomplished by using some temporary device identity which can be set up for this exact communication. Hence, this identity is not able to permanently identify the device and other devices cannot use it for future requests.

3.4 Towards an Optimized BlockChain for IoT

The interest to adopt blockchain in the IoT is increasing due to security and privacy. But there are certain issues associated with the usage of blockchain in IoT, which are discussed in [DKJ17].

3.4.1 Motivation

Blockchains are computationally costly and include high bandwidth overhead and delays which are not suitable for most IoT devices. A solution to such problem is presented in [DKJ17]. The solution comprises a lightweight blockchain-based architecture for IoT that virtually reduces the overheads of the traditional blockchain while sustaining most of its security and privacy functionalities. IoT devices take advantage from a private immutable ledger, which works similar to blockchain but is controlled centrally, to optimize energy consumption. High resource devices create an overlay network to realize a publicly accessible distributed blockchain that guarantees end-to-end security and privacy. The proposed architecture uses distributed trust to overcome the block validation processing time.

3.4.2 Implementation Approach

The solution presented in [DKJ17] is a hybrid approach as shown in figure 3.4 because the resulted application is still centralized but integrate some of the features of blockchain like security and privacy. The suggested architecture, contains three tiers, the smart home, the overlay, and the cloud storage.

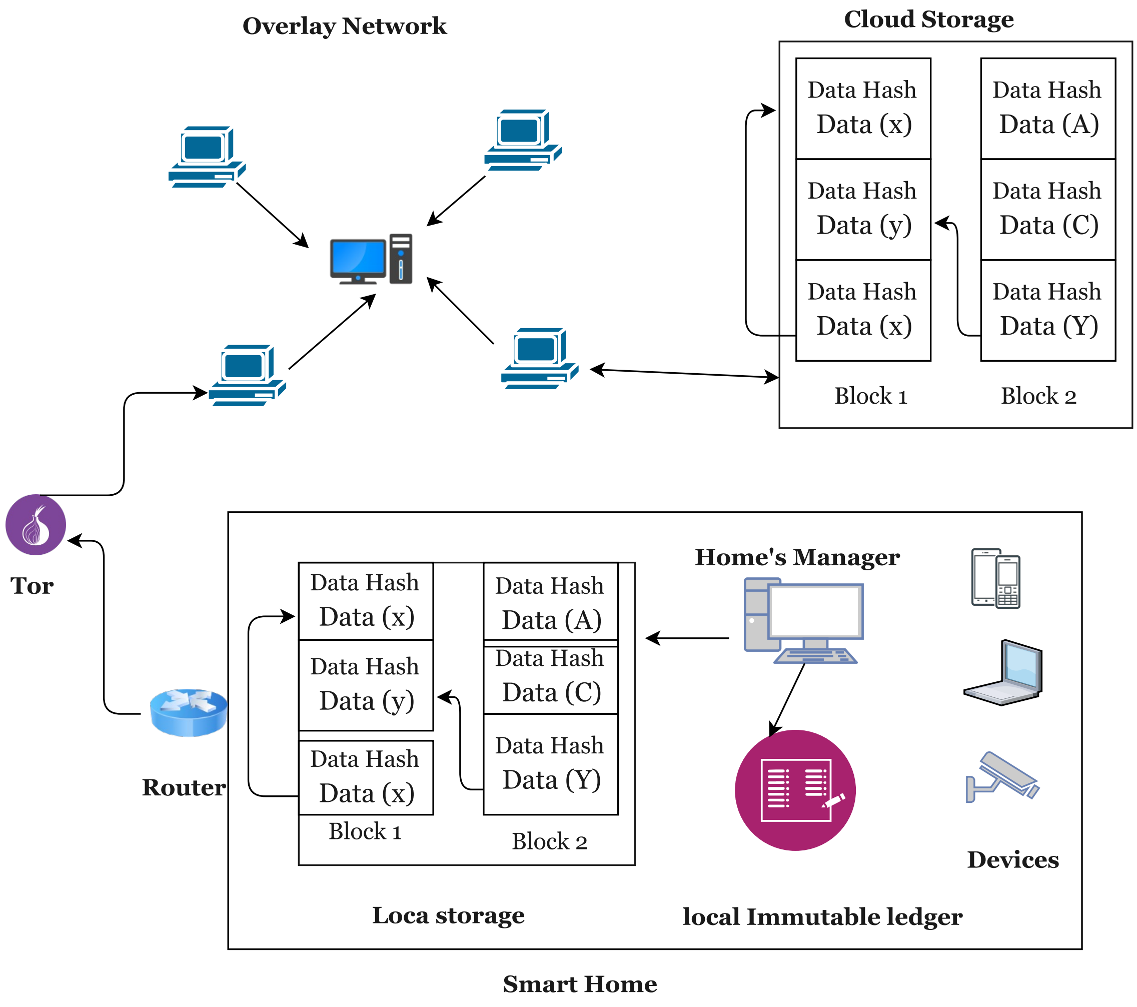


Figure 3.4: Blockchain-based Smart Home [DKJ17]

Smart Home includes IoT devices, local immutable ledger, and local storage. Every home has a local private immutable ledger that is comparable to a blockchain but is controlled centrally by the smart home manager. It processes all incoming and outgoing transactions and utilize a shared key for local communications with IoT devices and local storage. The local immutable ledger maintains a policy header specified by the home owner to approve the received transactions. Local devices inside the home or overlay nodes might generate transactions to share, request, or store data. In [DKJG17] in an extension to this work, the author added a miner to each home which will control and audit all the communication inside the home and from external to the home.

Overlay is a P2P network . The participant nodes are called overlay nodes, could be smart home managers, other large resource devices in the home, or the user's smart-phone or personal computer. To reduce network overhead and delay, nodes in the overlay are classified in clusters. Every cluster head has a unique primary key, known by other cluster heads in the overlay, utilized for generating new blocks so that other cluster heads could approve the block's generator. Every node is free to switch its cluster if it experiences excessive delays. Moreover, nodes in the cluster can choose a new cluster head any time. Every cluster head maintains primary key of the requesters and the primary key of the requestees. The overlay cluster heads maintain a public blockchain, which has a ledger for each overlay node that describes the history of transactions sent by the overlay user and is used to gain reputation.

Cloud Storage groups user's data in same blocks linked with a unique block-number. The block-number is used by the cluster head manager for authentication along with the hash of saved data. If the storage can successfully find data with the received block-number and hash from the cluster head manager, then the user is authenticated.

3.5 Blockchain as a Service for IoT

Due to the popularity of Software as a service (SaaS) model, nowadays most of the software is available as service, so it is a good idea to have such kind of platform, but there are some questions regarding this platform for IoT. How much network latency will be there, hosting blockchain on the cloud does obey the basic concept and standards of blockchain or not, etc.

3.5.1 Motivation

A blockchain is a distributed and decentralized ledger that contains connected blocks of transactions.Blockchain guarantees tamper-proof storage of approved transactions. Becuase of decentralization; security, and privacy the blockchain is started to be used to manage device configuration, store sensor data and enable micro-payments. As we know most of the IoT devices have limited resources, hosting blockchain on IoT devices is a key challenge. Therefore many blockchain platforms are developing a light client, which will only keep the crucial data on the device. In [SD16a] a cloud and fog based solution is presented to tackle this problem. Hosting the blockchain directly on resource-constrained IoT devices are not preferred because:

1. IoT devices has Low computational resources

2. IoT devices has low bandwidth.
3. IoT devices has limited power for Consumption.

3.5.2 Implementation Approach

The experimental setup described in [SD16a] to utilize fog and cloud as hosting for blockchain consist of fog (local cluster) and cloud (IBM's Bluemix). As IoT device, Intel Edison Arduino board, is used. In the experiments, the IoT device executes ten concurrent clients that write (720 bytes) to the multichain (via python server). In the execution, various delays are used to write requests from 0, 50, 100, 150, 200, 250 to 300 milliseconds. The results given in figures 3.5 and 3.6 for 300 ms delay shows that the fog outperformed the Bluemix cloud due to network latency.

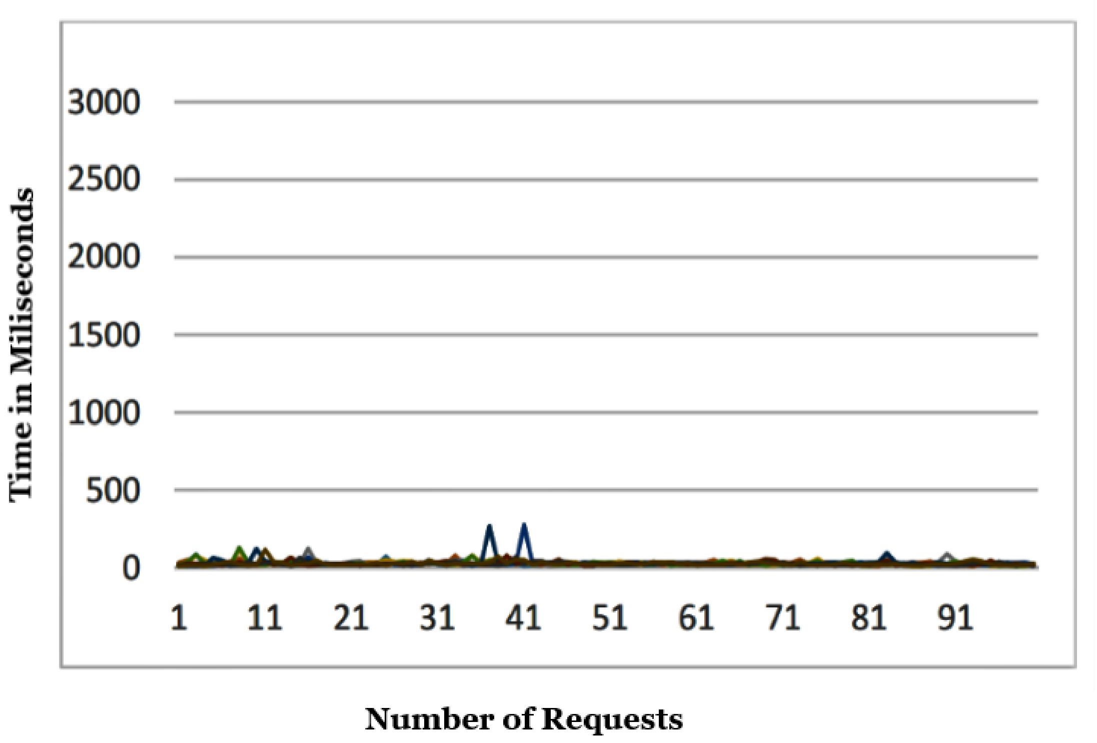


Figure 3.5: Request with 300 ms delay to fog hosted blockchain [SD16a]

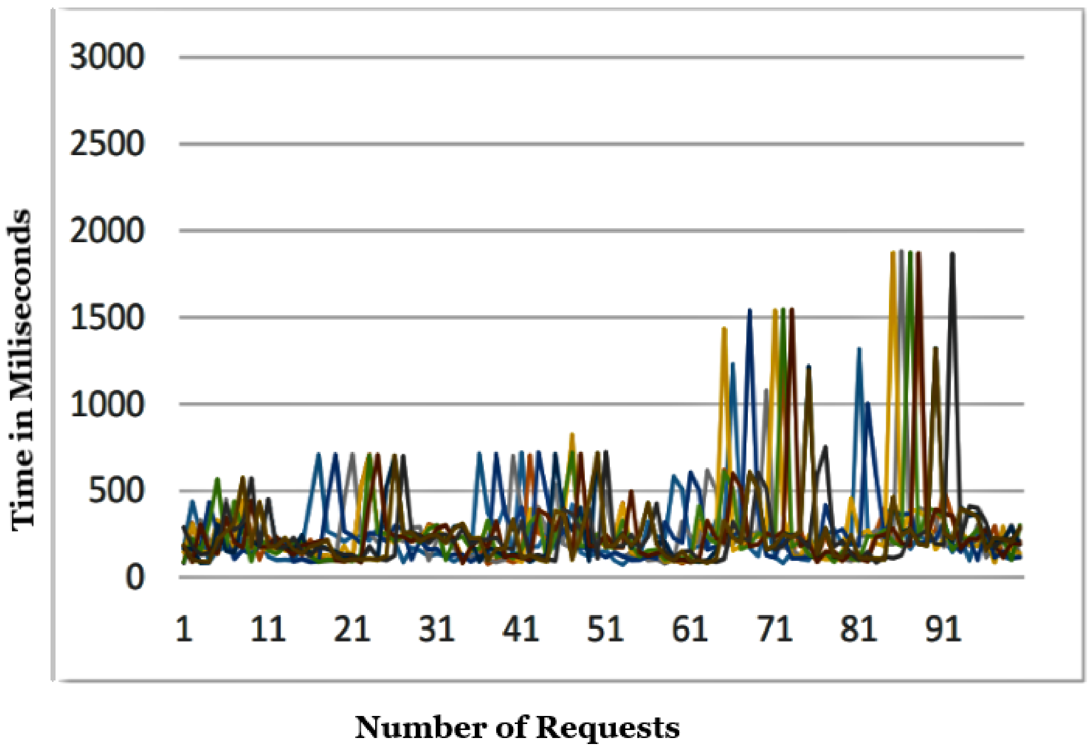


Figure 3.6: Request with 300 ms delay to Bluemix hosted blockchain [SD16a]

4 Concept and Specification

This chapter presents the system overview of the components used by the blockchain based IoT integration Dapp. The system functional and non-functional requirements including the detailed use cases will also be presented here.

4.1 System Overview

The figure 4.1 shows Dapp overview for the blockchain based ticketing system. The system consists of ethereum private blockchain, front-end application on the cloud or local web server, smart contract deployed on the private blockchain.

Private blockchain comprises of multiple P2P connected nodes of different IoT devices such as raspberry pi, laptop or personal computer, etc. Every node must install ethereum client and connect to same chain id or network id to become part of one network; when they become part of the same chain then they can execute transactions, calls, perform mining and take part in the consensus process.

The smart contract is the business logic of Dapp, and it is not centralized like traditional applications. The contract is deployed on the blockchain by the contract owner using his ethereum account. Once contract deployment transaction mined by miners, then it will be available on every node on the network on a specific address.

For customers to interact with the blockchain such as buy ticket, check the validity of ticket or for contract owner to change the ticket fee, transfer either from contract to another account there is a user-friendly GUI. To use the GUI, the user must have to use Mist browser or Metmask plugin for chrome and an ethereum account. Mist or Metamask allow the user to import his ethereum account to the system he is using and will enable the system to connect to blockchain network by making the user system a virtual blockchain node. They also inject web3.js component to the loading HTML pages. Web3.js is wrapper library allows for the transformation of data such as coding and decoding and allow a lot of other functionalities to interact RPC port provided by blockchain. If a user is using a direct blockchain network node and is capable of handling command line, then he does not need Mist or Metamask. The front-end app can be stored on the cloud or local web server or personal computer web server as we have given in the figure 4.1.

Ticket service is decentralized because of the business logic "smart contract." is available on every node of the network and the front-app access the contract using the unique address of the contract which uniquely identifies it on the network and similarly all the data attached with that particular contract.

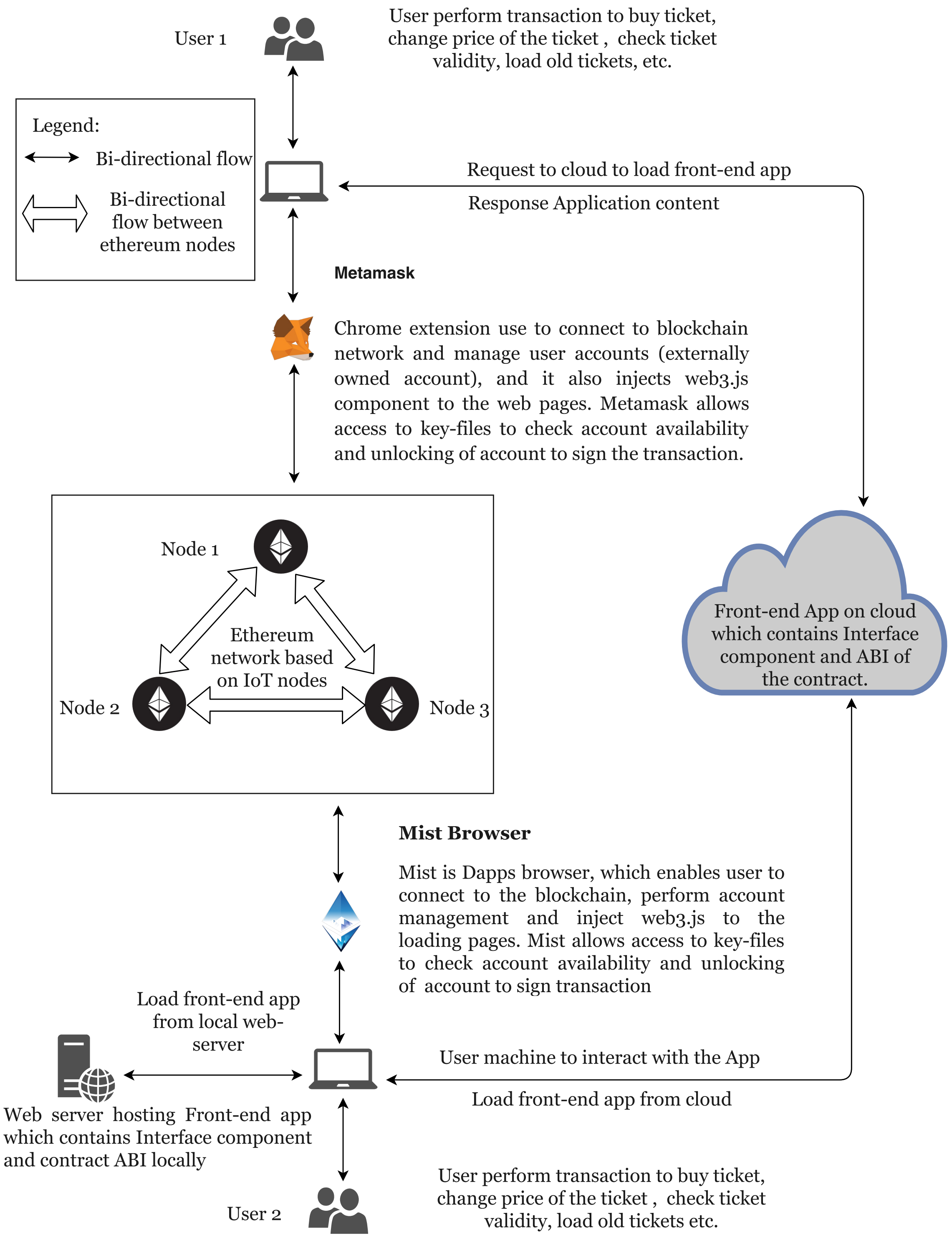

User 1
User perform transaction to buy ticket, change price of the ticket , check ticket validity, load old tickets, etc.
Legend:
Bi-directional flow
Bi-directional flow between ethereum nodes
Request to cloud to load front-end app
Response Application content
Metamask
Chrome extension use to connect to blockchain network and manage user accounts (externally owned account), and it also injects web3.js component to the web pages. Metamask allows access to key-files to check account availability and unlocking of account to sign the transaction.
Node 1
Ethereum network based on IoT nodes
Node 2
Node 3
Front-end App on cloud which contains Interface component and ABI of the contract.
Mist Browser
Mist is Dapps browser, which enables user to connect to the blockchain, perform account management and inject web3.js to the loading pages. Mist allows access to key-files to check account availability and unlocking of account to sign transaction
Load front-end app from local web-server
User machine to interact with the App
Load front-end app from cloud
Web server hosting Front-end app which contains Interface component and contract ABI locally
User 2
User perform transaction to buy ticket, change price of the ticket , check ticket validity, load old tickets etc.

Figure 4.1: System Overview of Ticketing Dapp

4.2 User Roles

There are two kinds of roles defined for this application; contract owner and customer.

4.2.1 Contract Owner

Contract owner has most of the administrative rights of the application. He can perform the following actions:

1. Create ethereum etherbase account for deployment of contract.
2. Deploy contract on the blockchain network.
3. Change ticket price/ day in wei, using his etherbase account address.
4. Transfer ether from contract account to another ethereum account.
5. Destory contract, to make it inaccessible including its ether and data.

4.2.2 Customer

The customer is the majority entity of the application. The customer can use application to buy, validate and check the history of all transport tickets. The actions of customer are:

1. Create ethereum account to buy a ticket.
2. Purchase a ticket for specific days and class.
3. Check validity of the currently available ticket.
4. Check the list of all ticket purchased.
5. Check the balance of his account.

4.3 Storage Requirement

With the great benefit of decentralization comes the cost of storage, because every ethereum node maintains a copy of blockchain data. But now lighter versions are in the development, which will retrieve data in real time from full nodes and will store only headers of the blocks. For private blockchain, it's not a problem, but for a public blockchain, it will be very costly to store a substantial amount of data. For this application, we use Raspberry Pi and a personal computer, which works fine. It is strongly advised to store as minimum data on the public blockchain because it is not designed for files, so avoid try uploading files to the blockchain. It is best suited for a key-value pair of data which is very efficiently presented in the ticketing Dapp.

4.4 Non-functional Requirements

Besides all the functionalities discussed, The ticketing Dapp must also take care of the following non-functional requirements as well.

Security - As the blockchain provide asymmetric cryptography and enforce its usage in the Dapps. But still, the user must be properly guided how to use his private keyfiles and the private key to avoid any financial loss.

Performance - Since on blockchain every instruction execution happen on all the connected nodes in the network, so try to limit the execution of code on blockchain because it is computationally expensive and inefficient.

Data integrity - Blockchain provides full data integrity, using cryptography and consensus to make any change to the system state. The data must be protected between the front-end application and blockchain because it is the time when most of the data goes unencrypted.

Usability - How to use Dapp is a bit tricky for a common non-technical person, so proper guidelines must be provided to educate the user to use the system.

Cost - Writing to blockchain or executing code on blockchain is expensive, so Dapp must be designed in a way that it perform most of the computation on the client side and minimal on the blockchain.

Portability - The Dapp must be designed in a way that it may run on all available platforms such windows, Linux, Mac, etc.

4.5 Use Cases

As mentioned above, there are two kinds of users, contract owner, and customer. Use cases show the possible actions they can perform on the system. The action constraint and perquisites are all given in detail in the description of each case.

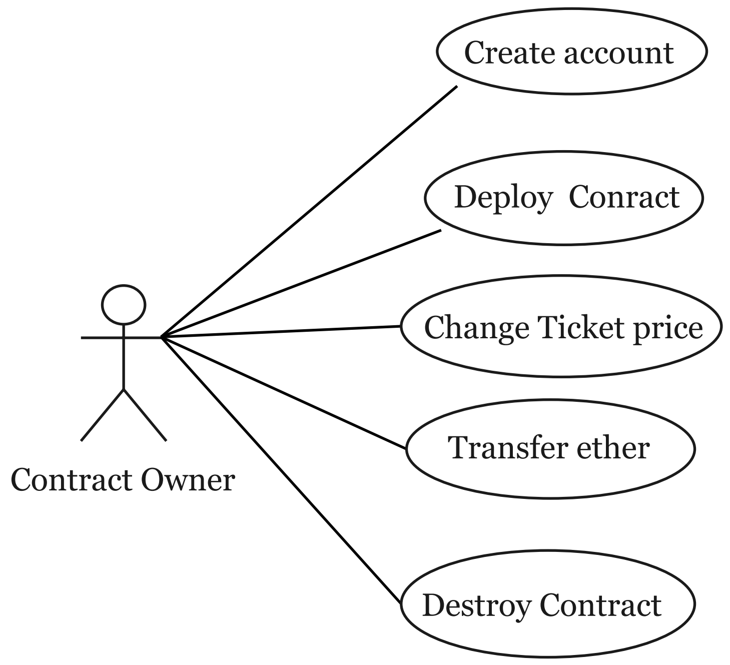


Figure 4.2: Use Case diagram of Contract Owner

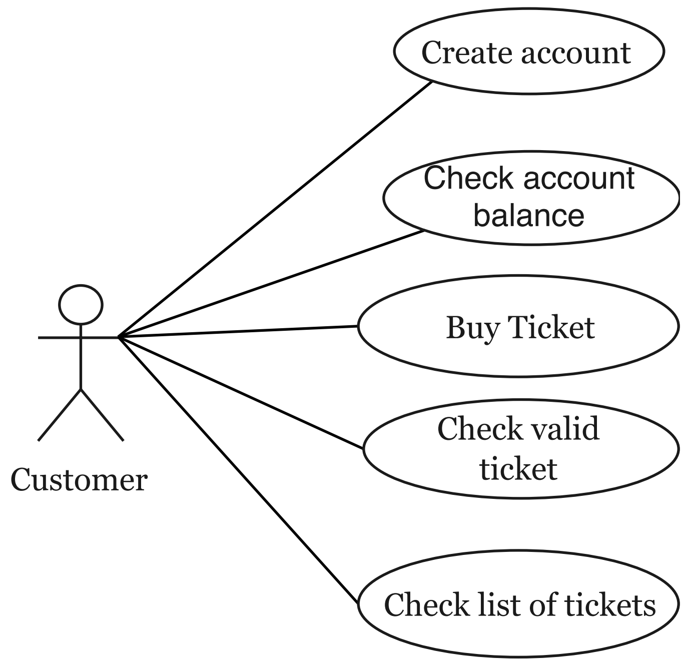


Figure 4.3: Use Case diagram of Customer

Name	**Create Account**
Goal	Create an etherbase account, which will be used to deploy contract on ethereum network.
Actor	Contract Owner
Pre-Condition	The Contract Owner system must be connected to ethereum network.
Post-Condition	A new account is created successfully.
Post-Condition in Special Case	Account creation failed.
Normal Case	• Contract Owner request new account. • An account is created, in response give 20 bytes account address, a unique identifier. • A keyfile is created which has encrypted private key, which is used to sign the transaction and generate public keys.
Special Cases	Account creation request failed because of error network, but it is a sporadic case.

Table 4.1: Description of use-case *Create an Account for the Contract Owner*

Name	**Deploy Contract**
Goal	Deploy a contract on blockchain.
Actor	Contract Owner
Pre-Condition	An etherbase (coinbase) account must be set to deploy contract on blockchain.
Post-Condition	Contract is successfully deployed.
Post-Condition in Special Case	Contract deployment failed.
Normal Case	• Contract owner deploys a contract on the blockchain. • Contract account is created on the blockchain. • Give the 20 bytes contract account address.
Special Cases	• Contract deployment failed because the etherbase account is locked to sign and pay the deployment transaction. • Insufficient funds error, if the etherbase has not enough ether to pay the deployment fee.

Table 4.2: Decription of use-case *Deploy Contract*

Name	**Change Ticket Price**
Goal	Change ticket price/day.
Actor	Contract Owner
Pre-Condition	Contract Owner account must be unlocked.
Post-Condition	Ticktet price changed successfully.
Post-Condition in Special Case	Ticket price change failed.
Normal Case	• Contract owner sends transaction to change ticket price with his account address and desire ticket value. • Transaction mined successfully, and the price gets changed. • Receive transaction hash.
Special Cases	• Invalid owner address if there is a mistake in address. • Insufficient funds error, if the etherbase has not enough ether to pay the transaction fee. • if somebody else tried to change using different account address, the transaction would execute and consume gas but no change in price or system state.

Table 4.3: Decription of use-case *Change Ticket Price*

Name	**Transfer Ether**
Goal	Transfer ether from contract account to an externally owned account.
Actor	Contract Owner
Pre-Condition	Contract Owner account must be unlocked.
Post-Condition	Ether transferred successfully.
Post-Condition in Special Case	Ether transfer failed.
Normal Case	• Contract owner sends the transaction to transfer ether from contract account with his account address, receiving account, an amount in wei. • Transaction mined successfully; ether transferred successfully. • Receive transaction hash.
Special Cases	• Invalid owner address or receiver address if there is a mistake in addresses. • Insufficient funds error, if the etherbase has not enough ether to pay the transaction fee. • if the sending amount is greater than the current balance of contract account. • if somebody else tried to transfer using different account address, the transaction would execute and consume gas but would not transfer ether or change system state.

Table 4.4: Decription of use-case *Transfer Ether*

Name	**Destroy Contract**
Goal	Destroying contract, make it inaccessible.
Actor	Contract Owner
Pre-Condition	Contract Owner account must be unlocked.
Post-Condition	Contract destroyed successfully.
Post-Condition in Special Case	Contract destruction failed
Normal Case	• Contract owner sends the transaction to destroy contracts, to make it inaccessible, using his account address. • Transaction mined successfully; contract destroyed successfully. • Receive transaction hash.
Special Cases	• Invalid owner address if there is a mistake in owner address. • Insufficient funds error, if the etherbase has not enough ether to pay the transaction fee. • if somebody else tried to destroy transaction using different account address, the transaction would execute and consume gas but would not destroy contract or change system state.

Table 4.5: Decription of use-case *Destroy Contract*

Name	**Create Account**
Goal	Create an account, which will be used to buy ticket using Dapp on ethereum network.
Actor	Customer
Pre-Condition	Customer must be connected to ethereum network.
Post-Condition	A new account is created successfully.
Post-Condition in Special Case	Account creation failed.
Normal Case	• Customer request new account. • An account is created, in response give 20 bytes account address, a unique identifier. • A keyfile is created which has encrypted private key, which is used to sign the transaction and generate public keys.
Special Cases	Account creation request failed because of error network, but it is a sporadic case.

Table 4.6: Description of use-case *Create an Account for the Customer*

Name	**Buy Ticket**
Goal	Buy a blockhain based transport e-ticket.
Actor	Customer
Pre-Condition	Customer account must be unlocked.
Post-Condition	Ticket purchased Successful.
Post-Condition in Special Case	Ticket purchase failed.
Normal Case	• Customer sends the transaction to buy ticket. • Transaction mined; Ticket purchased successfully. • Receive transaction hash.
Special Cases	• Invalid customer address error if there is a mistake in customer address. • Insufficient funds error, if the customer account has not enough ether to pay the transaction fee and ticket price. • If the customer has a valid ticket available currently, then a notification will be displayed showing the valid ticket information, and will not buy the new ticket.

Table 4.7: Description of use-case *Buy Ticket*

Name	**Check current valid ticket**
Goal	Check a valid ticket using customer account address.
Actor	Customer
Pre-Condition	Customer system must be connected to blockchain.
Post-Condition	Ticket load successfully.
Post-Condition in Special Case	Ticket loading failed.
Normal Case	• Customer calls the contract using contract function with customer address to load current valid ticket. • The response received is in JSON containing string values in hexadecimal format and integer values in Big numbers.
Special Cases	• Invalid customer address error if there is a mistake in customer address. • The ticket will not load if the address does not match the record in contract storage.

Table 4.8: Description of use-case *Check Current Valid Ticket*

Name	**Check list of tickets**
Goal	Check the list of all tickets purchased by the customer.
Actor	Customer
Pre-Condition	Customer system must be connected to blockchain.
Post-Condition	List of tickets load successfully.
Post-Condition in Special Case	Ticket list loading failed.
Normal Case	• Customer request contract to load all ticket events log that match customer address. • The response received is in JSON containing string values in hexadecimal format and integer values in Big numbers.
Special Cases	• Invalid customer address error if there is a mistake in customer address. • The ticket list will not load if the address does not match the address in the event logs in contract storage.

Table 4.9: Description of use-case *Check the List of Tickets*

Name	**Check account balance**
Goal	Allows customer to check account balance before or after ticket purchase.
Actor	Customer
Pre-Condition	Customer system must be connected to blockchain.
Post-Condition	Current balance displayed.
Post-Condition in Special Case	Balance loading failed.
Normal Case	• Customer request balance check using customer address. • The response displays current balance in wie and ether.
Special Cases	• Invalid customer address error if there is a mistake in customer address. • Unknown account error if the key file does not exist on metamask or Metamask.

Table 4.10: Description of use-case *Check the Account Balance*

5 Design

This chapter presents in detail the system design with a focus on the components involved in the Dapp development and the interaction between them. The data model for contract storage and reading/writing contract storage using different RPC requests to the functions and events of the contract will be discussed in detail here.

5.1 System Architecture

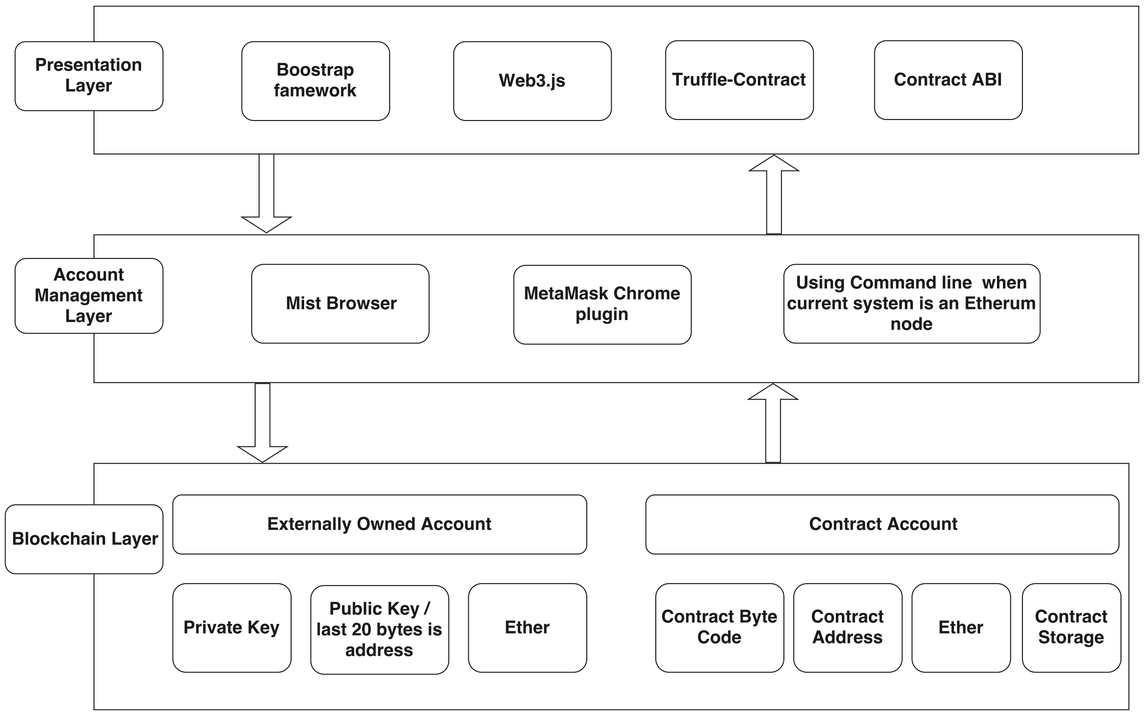


Figure 5.1: System Architecture, Blockchain based ticketing Dapp

As shown in figure 5.1, the architecture is based on three layers such as blockchain layer, account management layer, and presentation layer. The Dapp utilize all these three layers, which make it decentralized, secure, and interactive.

Blockchain Layer represents all the blockchain nodes because every node in blockchain network contains all the components that belong to this layer. It connects all the nodes within a blockchain network, in our case these nodes includes IoT devices and this layer enable us to trigger events on the blockchain which can be used to actuate an actuator connected to the node or read a sensor. There are multiple components of this layer but mainly divided into two types based on account type. EOA and Contract account. Each EOA has a private key, public key, account address, and hold ether. The contract account has contract-byte-code which execute on EVM, contract address, hold ether and has contract storage which is used to store contract data permanently on the blockchain. Events can also be specified in the contract which is associated with that particular contract, can be easily accessed though contract instance or triggered and watched to perform a certain operation and to filter the history of an operation using indexed field. In our Dapp we are using events to search old tickets for a particular customer.

Account Management Layer consist of account management tools which enables us to perform transactions or calls on the blockchain. As shown in the figure 5.1 it contains three components which can be used to manage ethereum account, which is Mist, Metamask, and command line when ethereum client is running on the user system, and it is part of ethereum network. Mist is the Dapp browser from ethereum which allows the user to create, import existing account to the wallet and use it for different operation on the blockchain. It allows users to connect to the network of their choice, unlock account for transactions, makes the user system virtual ethereum node, and also injects web3.js to the loading pages, which act as a wrapper to perform different operation on blockchain using JSON-RPC. Mist is a plug-in for chrome which makes chrome a Dapp browser and performs similar activities as Mist except it does not deal with contract accounts, however, for EOA it makes it possible to conduct transactions using contract code on the blockchain. Command line component is used when the user system is part of the ethereum network and running ethereum client, and the application is using the local web3.js library. Mist and Metamask are for the non-developers and developers to run Dapps and command line is for Dapps developers only.

Presentation Layer is concerned with the web-based GUI for interaction with blockchain. The presentation layer components are the Bootstrap framework, web3.js, truffle-contract, and contract ABI. Bootstrap is used to build HTML and CSS based web GUIs. Web3.js is a wrapper library for interacting with JSON-RPC of blockchain to avoid low-level conversion which is error-prone. Truffle-contract is built on the top of web3.js, and it provides more functionalities than web3.js such as it uses promises which eliminates the need for callback functions; it uses synchronized transaction which allows full control by not finishing transaction until it is not mined. Contract-ABI is the list of contract functions and argument in JSON format which is used in the front-end application to specify which function to call with what number of parameters and guarantee the function return data format. Contract ABI does not reside on blockchain; we can get this after contract compilation if using truffle framework then in the build directory contractName.json file.

5.2 Data model

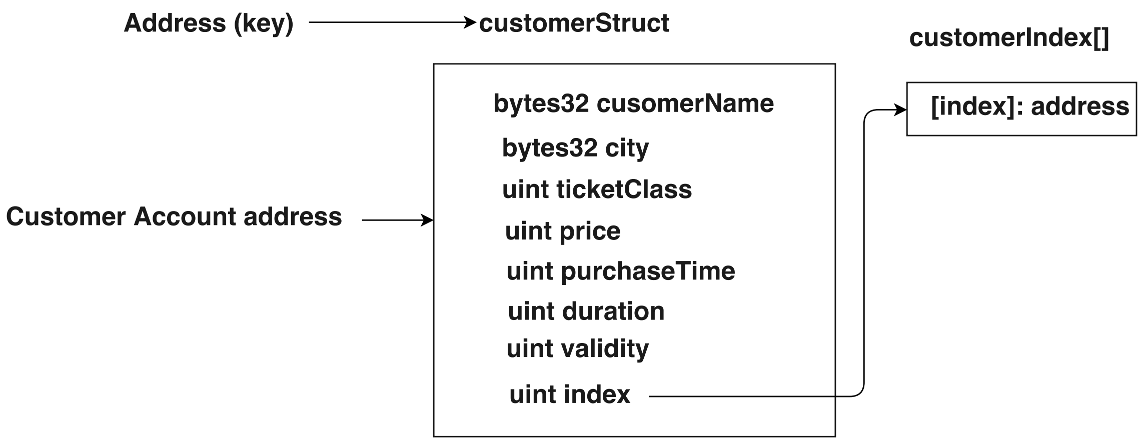


Figure 5.2: Data Model for Ticketing System

The storage of a smart contract is not a database, we are responsible for the internal organization of data at a deep level. The model for data storage presented in this thesis is based on the model presented by Rob Hitchens in [Hit17]. The model presented in figure 5.2 allows us to write data to contract storage like writing data to a table in the database and allow overwriting of data. We use overwriting of data because in blockchain old data is not deleted it still exists on blockchain in the old blocks and we can access it, overwriting just replace the data on the current index. A struct in Solidity allows us to write our own variable, in our the model we use cutomerStruct which has eight fields. We cannot apply database operations on struct only such as the searching the record of a particular customer, counting a number of customers, etc; so keep customer address in an array of addresses and store that array index in the index field of the struct which makes a relationship between struct and the array of addresses. This model enables us to perform the following operations:

1. Make a single entity with a defined set of fields.
2. Insert new records known by a Key.
3. Randomly retrieve records by their keys.
4. Retrieve a record count.
5. Access a list of all the records that exist.
6. Update field(s) in any given record.
7. Key existence validation.

5.3 Requests Pattern for Contract functions using Blockchain RPC Interface

In this section a pattern to access contract functions with web3.js and truffle-contract through blockchain RPC interface. A detail description of different functionalities of the contract along with the allowed actors who can execute the functions and events, parameters description, and responses to the requests will be presented.

function	**addCustomer**
Description	This function allows the purchase of the ticket by adding a record on the blockchain.
Actor	Customer
Call-pattern	contractInstance.addcustomer()
Function parameters	• customerAddress : Customer account address. • customerName : Customer name, which will be displayed on ticket. • ticketCity : The city for which the ticket is valid. • ticketClass : Ticket Class such as Ist or 2nd Class. • ticketDuration : Duration in number of days. • "value:price, from:customerAddress, gas:value" :This composite parameter has three sub-parameters such as "value" represent the ticket fee in wei, "from" the address from which the fee(ticket fee + transaction fee) will be deducted and "gas" shows the maximum allowed transaction cost in term of gas value.
Response	Response is an object which contain transaction hash, receipt object, and logs array.
Error Response	• out of Gas error. • Invalid Address. • Unlock Account error. • Insufficient funds.

Table 5.1: Description of the Contract function *Purchase ticket*

function	**getCustomer**
Description	This function allows the customer to check the validity of the ticket.
Actor	Customer
Call-pattern	contractInstance.getCustomer()
Function parameters	• address : Customer account address.
Response	Response is an array which contain multiple hexadecimal and big-number values
Error Response	• Empty array if no record exist in the storage.

Table 5.2: Description of the Contract function *getCustomer*

Event	**ticketPurchase**
Description	This event allows watching or filter event logs of ticket purchase operation and used to get list of old tickets for a particular address.
Actor	Customer
Call-pattern	contractInstance.ticketPurchase()
Event parameters	• IndexedAddress : The indexed address make the customer purchase event logs searchable based on the indexed customer address. • "fromBlock: 0, toBlock: 'latest'" : A composite parameter used to limit the filtering process by describing starting block and ending block.
Response	Response is an array of objects and each object contain data for a particular ticket
Error Response	• Empty array if no event logs exist for the current address.

Table 5.3: Description of the Contract Event *ticketPurchase*

function	**setTicketPrice**
Description	This function allows the contract owner to set or change the ticket price value in wei.
Actor	Contract Owner
Call-pattern	contractInstance.setTicketPrice()
Function parameters	• price : Ticket new value in wei. • "from:contract owner address, gas:value" : This composite parameter contain two components such as "from", the address from which the transaction fee will be deducted and "gas", which shows the maximum allowed transaction cost in term of gas value.
Response	Response is an object which contain transaction hash, receipt object, and logs array.
Error Response	• out of Gas error. • Invalid Address. • Unlock Account error.

Table 5.4: Description of the Contract function *setTicketPrice*

function	**transferFromContract**
Description	This function allows the contract owner to transfer ether from contract account to EOA.
Actor	Contract Owner
Call-pattern	contractInstance.transferFromContract()
Function parameters	• address : The receiver address. • amount : The amount of ether to transfer from contract, always represented in wei. • "from:contract owner address, gas:value" : This composite parameter contain two components such as "from", the address from which the transaction fee will be deducted and "gas", which shows the maximum allowed transaction cost in term of gas value.
Response	Response is an object which contain transaction hash, receipt object, and logs array.
Error Response	• out of Gas error. • Invalid Address. • Unlock Account error. • Insufficient funds.

Table 5.5: Description of the Contract function *transferFromContract*

function	**getCustomerCount**
Description	This function allows the check the total number of registered customers.
Actor	Customer and Contract Owner
Call-pattern	contractInstance.getCustomerCount()
Function parameters	• none
Response	Response is a bigNumber represents the total number of registered customers.
Error Response	• Return 0 if no customer is registered.

Table 5.6: Description of the Contract function *getCustomerCount*

6 Implementation and Validation

In this chapter, we will follow the specification discussed in chapter 4 and the architecture presented in the chapter 5 to realize blockchain-based ticketing Dapp. The implementation is divided into three phases (i) Private blockchain configuration (ii) Smart contract development and deployment (iii) Front-end application development. In the end validation of the application will be presented in order to check the conformity of the Dapp with the specifications.

6.1 Private Blockchain Configuration

As discussed in chapter 2, in private blockchain write permissions are centralized, belongs to one organization. Read permissions may be public or limited to an arbitrary amount. For the configuration of the private blockchain, guidelines are followed which are discussed in [Elo17; Eth17b], with some modifications. In our application demo, we are using a Raspberry Pi device with one laptop as the miner node. The configuration consist of the follwing steps

1. Download ethereum client from https://geth.ethereum.org/downloads/, it has a list of compatible versions for different platforms; download one which is compatible with your platform by checking your device CPU; then install it. To check installation execute " geth version" command in the shell.
2. To start geth in the shell execute "geth" command. The client will start synchronization with the main chain, press CTRL+C to stop it. The default locations for the ethereum contents are
    - Windows:
    - Linux: /.ethereum/keystore
3. In a private blockchain to join the same blockchain, every node has to fulfill the following requirements.
    - A distinct data directory have to be created to store the database and the wallet, which belong to the private blockchain.
    - Every node has to initialize the same genesis file. Genesis file initialization creates the genesis block of the blockchain, which is the first block and does not refer to any block.

- To connect to the same blockchain every node has to join same network id, assign any id except 1,2,3 because they are reserved for the main chain.

The genesis block configuration of our private blockchain is given in the listing below

Listing 6.1 Genesis Block Configuration

```
{
      "nonce": "0x0000000000000042",
      "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
      "difficulty": "0x400",
      "alloc": {},
      "coinbase": "0x0000000000000000000000000000000000000000",
      "timestamp": "0x00",
      "parentHash":
          "0x0000000000000000000000000000000000000000000000000000000000000000",
      "extraData": "0x",
      "gasLimit": "0xffffffff",
      "config": {
            "chainId": 59,
            "homesteadBlock": 0,
            "eip155Block": 0,
            "eip158Block": 0
            }
}
```

4. To initialize the private blockchain in the custom directory we created; the following geth command will have to be executed

Listing 6.2 Private Blockchain Initialization

```
$ geth --datadir ~/IotChain init genesis.json
// IotChain is the directory which will have the private blockchain database and keystore.
```

5. Then create account using the following command at the moment because geth is not in the running state.

Listing 6.3 Account Creation on Private Blockchain

```
$ geth --datadir ~/IotChain account new
Your new account is locked with a password. Please give a password. Do not forget this
    password.

Passphrase:
Repeat passphrase:
Address: {79eeaf45cef97564022d924dc67113170bb1f46c}
```

6. To start geth on the nodes; execute the following command which includes different flags for different functionalities, the node will perform such as the miner node will use "–mine" flag. We can execute the command in the shell or create bash file and put the whole command in the file, then run bash file in the shell. The main difference

between the commands for a miner node and normal node is the "–mine" flag. To manage the miner at run time, follow the guidelines given in 2.7.3. The purpose of

Listing 6.4 Starting Geth for Private Blockchain

```
$ geth --identity "miner1" --networkid 59 --datadir "~/IotChain" --nodiscover
        --mine --rpc    --rpcport "8045" --port "30303" --rpcapi "db,eth,net,web3"
        --rpccorsdomain "*" --unlock 0  --password ~/IotChain/password.sec
```

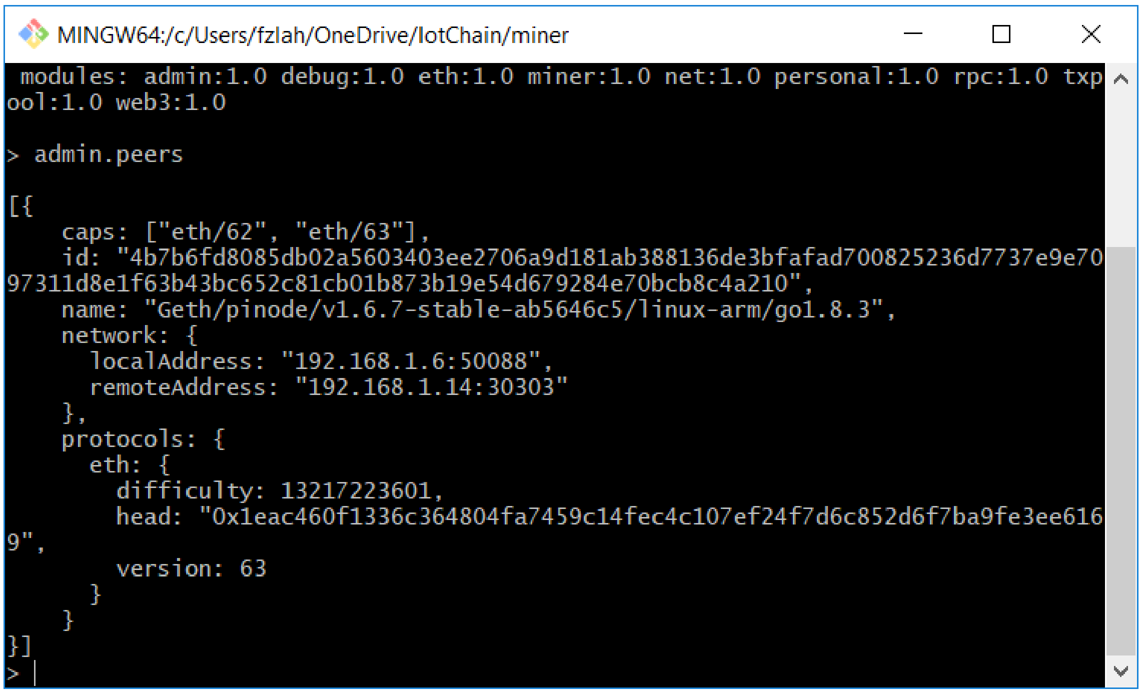
different flag used in the command are

- identity: The name of the node.
- networkid: This is a unique network identifier that distinguishes one network from other networks and is used to join all the nodes of the same network.
- datadir: Folder where private blockchain stores its data.
- rpc and rpcport: It enables HTTP-RPC server on a specific port number, in our case, it is 8045.
- port: The port through which nodes in a network communicate with each other and spread new transactions and blocks.
- rpcapi : It list APIs which can be accessed using RPC port.
- nodiscover: It disables the discovery mechanism.
- mine: It makes the node a miner node, which mine ether and transactions.
- rpccorsdomain: RPC cross-origin request domain, it can be used to restrict to accept RPC calls coming from different domains to the allowed domains only, which are defined here.
- unlock 0 : It unlocks the account at the index 0, mostly it is coinbase account.
- password: The path to the file containing the password of the default account.
- ipcpath: The path to store the filename for Inter Process Communication (IPC) socket/pipe.

7. To pair the nodes we will be using the static-nodes.json file, where we put the enode id, which we get using the command given in 2.6.2. We use this method because there are limited number of nodes to connect and there is no need for discovery mechanism to pair nodes dynamically. Our static-nodes.json file content are given below
8. Restart the miner, that's it, now we have our own private blockchain with fully synchronized nodes.

Listing 6.5 Pairing Nodes using Static-nodes.json on Private Blockchain

```
[
      "enode://70070c8c8d0f3f2e8005dfbaa7f6f42ba82e571b1d140fed0add51c582fce267695819fdd71a66c9
           3a495ec612666ce699d24719149a620d2a3bef9958813269@192.168.1.6:30303",
      "enode://4b7b6fd8085db02a5603403ee2706a9d181ab388136de3bfafad700825236d7737e9e7097311d8e1
           f63b43bc652c81cb01b873b19e54d679284e70bcb8c4a210@192.168.1.14:30303"
]
```



MINGW64:/c/Users/fzlah/OneDrive/IotChain/miner

```
 modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txp
ool:1.0 web3:1.0

> admin.peers

[{
    caps: ["eth/62", "eth/63"],
    id: "4b7b6fd8085db02a5603403ee2706a9d181ab388136de3bfafad700825236d7737e9e70
97311d8e1f63b43bc652c81cb01b873b19e54d679284e70bcb8c4a210",
    name: "Geth/pinode/v1.6.7-stable-ab5646c5/linux-arm/go1.8.3",
    network: {
      localAddress: "192.168.1.6:50088",
      remoteAddress: "192.168.1.14:30303"
    },
    protocols: {
      eth: {
        difficulty: 13217223601,
        head: "0x1eac460f1336c364804fa7459c14fec4c107ef24f7d6c852d6f7ba9fe3ee616
9",
        version: 63
      }
    }
}]
> |
```

Figure 6.1: Private blockchain Peers List

6.2 Smart Contract Development and Deployment

For Smart contract coding in solidity, we are using Visual Studio Code, which is a very flexible and handy code editor for solidity coding. For compilation, testing and deployment we are using Truffle framework, some of its core functionalities are already discussed in 2.10.2. The whole process can be fulfilled in the following steps

1. Open a Console window and run this command "npm install -g truffle". It will install truffle framework on the system.
2. Create a project directory and run the command "truffle init" inside the directory, it will generate three directories and a file called truffle.js as shown in the listing below.

3. truffle.js is the truffle configuration file as shown in the listing below. Description of the network to test or deploy Smart contracts is also mentioned here.

Listing 6.6 Truffle initialization and Directory Structure

```
$ truffle init
Downloading project...
Project initialized.

Documentation: http://truffleframework.com/docs

Commands:

Compile: truffle compile
Migrate: truffle migrate
Test:  truffle test

// The directories and file created are
Contracts directory - Truffle looks for contracts here.
Migrations directory- contains deployment scripts
test directory- test files location
truffle.js file - truffle configuration file.
```

Listing 6.7 truffle.js Configuration

```
// Allows us to use ES6 in our migrations and tests.
require('babel-register')
module.exports = {
      networks: {
            development: {
                  host: 'localhost',
                  port: 8045,
                  network_id: '*' // Match any network id
            }
      }
}
```

4. Now its time to present the Smart Contract code. The listings below show the Smart Contract written in solidity and named TicketContract.sol.

Listing 6.8 Smart Contract for Ticketing System Part-1

```
pragma solidity ^0.4.14;
contract TicketContract {
    // Ticket struct a custom variable.
    struct ticket{
        bytes32 customerName;
        bytes32 city;
        uint ticketClass;
        uint price;
        uint purchaseTime;
        uint duration;
        uint validity;
        uint index;
     }
    // here we map ticket struct to an address, which is analogous to the primary key in table and the struct represent the fields in a table.
    mapping(address => ticket) private customerTicket;
    // address array which store cutomer address
    address[] private customerIndex;
    // owner is the contract owner, it gets value for the first time when contract run, later this variable cannot be accessed , it is used to restrict
    // the access to the smart contract owner operations like changing ticket price, transfering ether from contract etc.
    address public owner = msg.sender;
    uint public ticketPrice ;
    // ticketPurchase event declaration which gets triggered when user purchase a ticket. which we use to filter old tickets.
    event ticketPurchase(address indexed customerAddress, bytes32 customerName, bytes32 city, uint price,uint purchaseTime, uint validity);
    // pricechange event declaration which gets triggerd whenever ticketprice get changed.
    event priceChanged(uint newPrice,uint timeStamp);
    // fromContractToexternal event declaration which gets triggered whenever ether transfer transaction is performed by the contract owner.
    event fromContractToexternal(address reviever, uint amount);
    // function to set ticket price, only contract owner can execute this function.

    function setTicketPrice(uint newPrice) public returns (bool success) {
        if (msg.sender != owner)
            revert();
         ticketPrice = newPrice;
         priceChanged(newPrice, block.timestamp);
         return true;
    }
    // function to get the current price of the ticket
    function getTicketPrice() public constant returns (uint price) {
     return ticketPrice;
    }
```

Listing 6.9 Smart Contract for Ticketing System Part-2

```
         // function to check whether the current customer is registered or not.
        function isCustomer(address customerAddress) public constant returns(bool
            isAvailble) {
            if (customerIndex.length == 0)
                return false;
            return (customerIndex[customerTicket[customerAddress].index] ==
                customerAddress);
        }
        // if a customer address does not exist on our database, then add the
            customer and purchase the ticket.
        function addCustomer(
            address customerAddress,
            bytes32 customerName,
            bytes32 city,
            uint ticketClass,
            uint duration
            ) payable public returns(uint index)
        {
            if (isCustomer(customerAddress) || msg.sender.balance < msg.value) {
                updateCustomer(customerAddress, customerName, city, ticketClass,
                    duration);
            }else {
                uint ticketValue = msg.value;
                customerTicket[customerAddress].customerName = customerName;
                customerTicket[customerAddress].city = city;
                customerTicket[customerAddress].ticketClass = ticketClass;
                customerTicket[customerAddress].price = ticketValue;
                customerTicket[customerAddress].purchaseTime = now;
                customerTicket[customerAddress].duration = duration;
                customerTicket[customerAddress].validity = now + duration;
                customerTicket[customerAddress].index = customerIndex.push(
                    customerAddress)-1;
                // execute the ticket purchase event here.
                ticketPurchase(
                    customerAddress,
                    customerName,
                    city,
                    customerTicket[customerAddress].price,
                    customerTicket[customerAddress].purchaseTime,
                    customerTicket[customerAddress].validity
                    );
                return customerIndex.length-1;
            }
        }
        // funtion to check the current valid ticket for the customer.
        function getCustomer(address customerAddress)
            public
            constant
            returns(bytes32 customerName, bytes32 city, uint ticketClass, uint price,
                uint purchaseTime, uint duration, uint validity)
```

Listing 6.10 Smart Contract for Ticketing System Part-3

```
        {
            if (!isCustomer(customerAddress))
                revert();
            return(
                customerTicket[customerAddress].customerName,
                customerTicket[customerAddress].city,
                customerTicket[customerAddress].ticketClass,
                customerTicket[customerAddress].price,
                customerTicket[customerAddress].purchaseTime,
                customerTicket[customerAddress].duration,
                customerTicket[customerAddress].validity
                );
        }
        // Checkvalidity function to get the validity of the ticket for the repective
             customer address
        function checkValidity(address customerAddress) public constant returns(uint
            validity) {
            return customerTicket[customerAddress].validity;
        }
        function getClass(address customerAddress)
            public
            constant
            returns(uint ticketClass)
        {
            if (!isCustomer(customerAddress))
                revert();
            return(
                    customerTicket[customerAddress].ticketClass
                );
        }
        // if A customer Address already exist then just update the ticket parmeters ,
             simply purhase a new ticket for the current customer.
        function updateCustomer(
            address customerAddress,
            bytes32 customerName,
            bytes32 city,
            uint ticketClass,
            uint duration)
            payable
            public
            returns(bool success)
        {
            if (isCustomer(customerAddress) && msg.sender.balance > msg.value) {
                uint ticketValue = msg.value;
                customerTicket[customerAddress].customerName = customerName;
                customerTicket[customerAddress].city = city;
                customerTicket[customerAddress].ticketClass = ticketClass;
```

Listing 6.11 Smart Contract for Ticketing System Part-4

```
                customerTicket[customerAddress].price = ticketValue;
                customerTicket[customerAddress].purchaseTime = now;
                customerTicket[customerAddress].duration = duration;
                customerTicket[customerAddress].validity = now + duration;
                // trigger ticket purchase event.
                ticketPurchase(
                    customerAddress,
                    customerName,
                    city,
                    customerTicket[customerAddress].price,
                    customerTicket[customerAddress].purchaseTime,
                    customerTicket[customerAddress].validity
                );
                return true;
            } else {
                revert();
            }
        }
        // funtion to get the total number of registered customers.
        function getCustomerCount()
            public
            constant
            returns(uint count)
        {
            return customerIndex.length;
        }
        // fucntion to check a customer at a specific index of customerIndex array.
        function getCustomerAtIndex(uint index)
            public
            constant
            returns(address customerAddress)
        {
            return customerIndex[index];
        }
        // transferFromContract facilitate owner of the contract to transfer ether
            from the contract account to other accounts.
        function transferFromContract(address reciever,uint amount) payable public
            returns(bool success) {
            if (msg.sender == owner && this.balance>=amount) {
                reciever.transfer(amount);
                fromContractToexternal(reciever,amount);
                return true;
            }
        }

    }
```

5. Then in the migrations directory, delete other files except 1_initial_migration.js and create another js file with the name 2_deploy_contract.js with the following contents

Listing 6.12 Migration configuration for contract deployment

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

6. Now open a console in the project directory and run the command "truffle compile". It compile the contract, if there is any compile time errors it will be thrown here.
7. Then execute "truffle migrate –reset" command, during this operation keep miner in running condition. At the completion, it will give a message contract successfully deployed. In the build directory there will be TicketContract.json file. This file contains the ABI which will be used in the front-end application.

6.3 Front-end Application

In this section, we will present the basic configuration for the front-end application, for detail description check the source code. As we discussed earlier we are using Bootstrap, Web3.js, truffle-contract and smart contract ABI. The following list shows how to perform the basic configuration and start using it for the rest of functionalities.

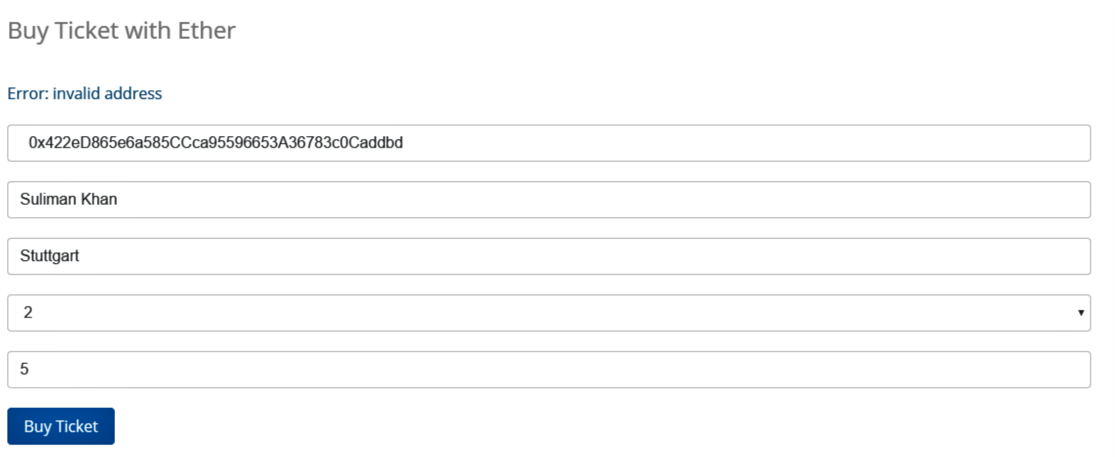
Listing 6.13 Front-end Application Basic Configuration Demo

```
import "../stylesheets/app.css";
import { default as Web3} from 'web3';
import { default as contract } from 'truffle-contract'
import ticketArtifacts from '../../build/contracts/TicketContract.json'
var ticketContract = contract(ticketArtifacts);
// loadPrice() a simple demo function to load ticket current price from the
    blockchain using smart contract
window.loadPrice = function(){
    ticketContract.deployed().then(function(contractInstance) {
        contractInstance.getTicketPrice().then(function(v) {
            console.log(v);
            $("#Price").attr('value',v.toString());
            web3.eth.getBalance(contractInstance.address, function(error, result)
                {
                $("#contract-balance").html(web3.fromWei(result.toString()) + " :
                    Contract Ether | "+ result.toString()+" : Contract wei");
            });
        });
    });
}
```

6.4 Validation

In this section, different snapshots of the front-end application will be presented, which are taken at distinct state of the application, which shows whether our Dapp comply to the specification and design we presented in the previous chapters. The list of snapshots are

1. When there is an error in the account address the following message get displayed.



Buy Ticket with Ether

Error: invalid address

0x422eD865e6a585CCca95596653A36783c0Caddbd

Suliman Khan

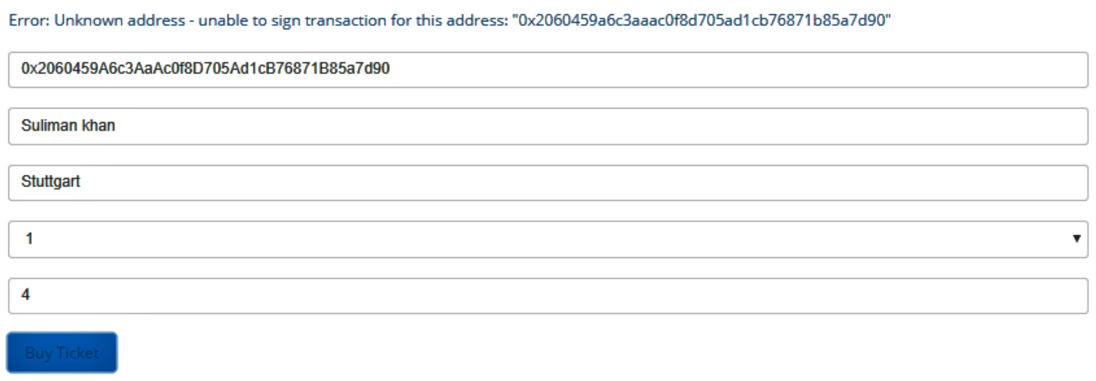
Stuttgart

2

5

Figure 6.2: Invalid Address error while buying ticket

2. If the account keyfile does not exist in the keystore of the node or in the wallet then it throws the following error.



Error: Unknown address - unable to sign transaction for this address: "0x2060459a6c3aaac0f8d705ad1cb76871b85a7d90"

0x2060459A6C3AaAc0f8D705Ad1cB76871B85a7d90

Suliman khan

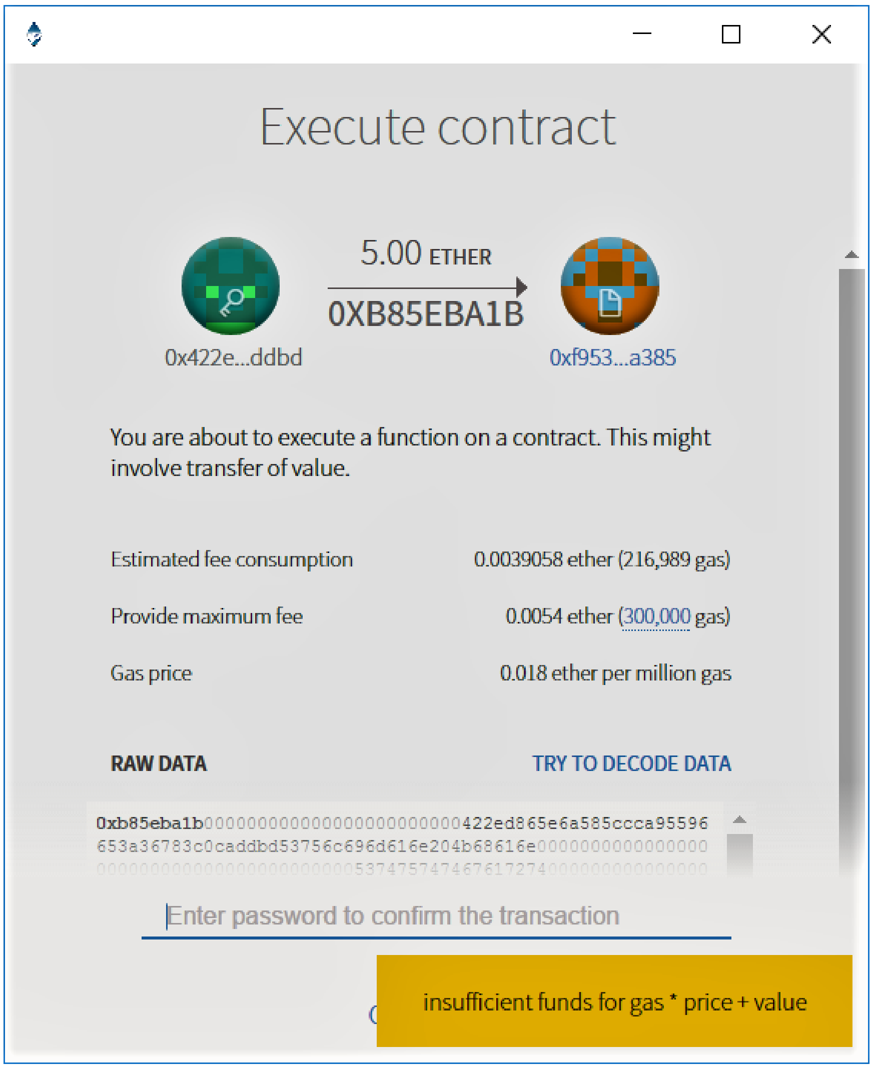
Stuttgart

1

4

Figure 6.3: Unknown Account error

3. When the account has insufficient funds to perform transaction and purchase ticket then the following notification displayed when running application in Mist browser and transaction get failed to complete.



You are about to execute a function on a contract. This might involve transfer of value.

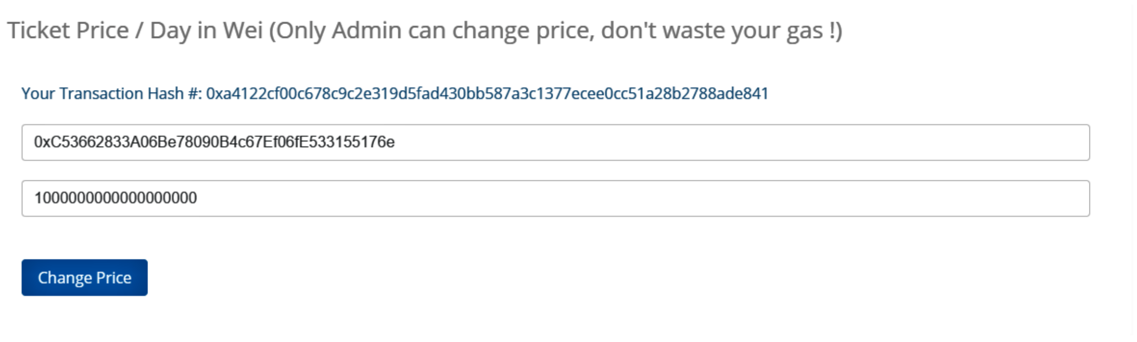
Estimated fee consumption	0.0039058 ether (216,989 gas)
Provide maximum fee	0.0054 ether (300,000 gas)
Gas price	0.018 ether per million gas

```
0xb85eba1b000000000000000000000000422ed865e6a585ccca95596
653a36783c0caddbd53756c696d616e204b68616e0000000000000000
000000000000000000000000537475747467617274000000000000000
```

insufficient funds for gas * price + value

Figure 6.4: Insufficient funds error

4. The first important thing for the contract owner is to set the ticket price after successful contract deployment using the application.


Ticket Price / Day in Wei (Only Admin can change price, don't waste your gas !)

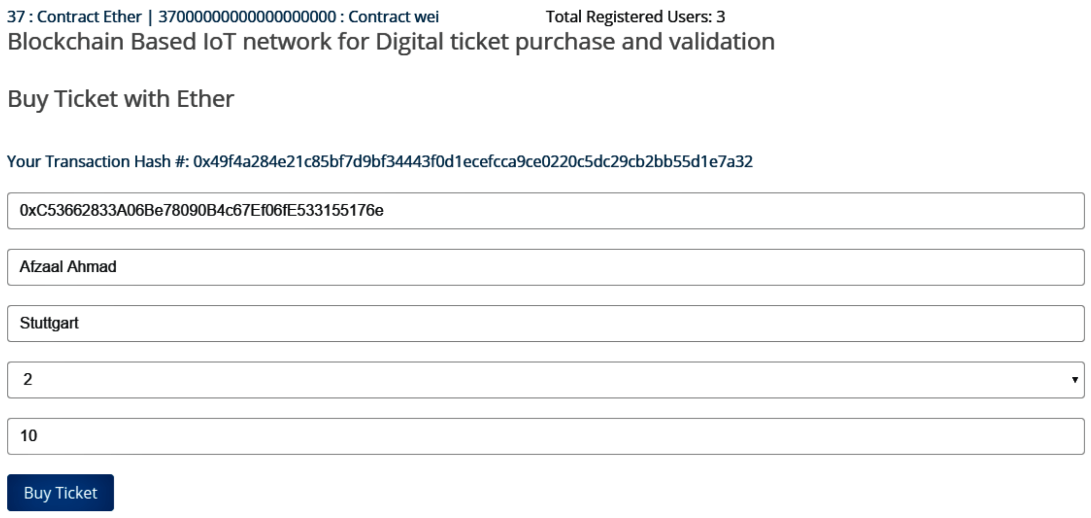
Your Transaction Hash #: 0xa4122cf00c678c9c2e319d5fad430bb587a3c1377ecee0cc51a28b2788ade841

0xC53662833A06Be78090B4c67Ef06fE533155176e

1000000000000000000

Figure 6.5: Ticket Price Change in wei

5. When all the above errors do not exist and ticket price is set, then the application perform the transaction for the ticket purchase and when the transaction get mined we get the transaction hash as given in the figure below.



37 : Contract Ether | 37000000000000000000 : Contract wei   Total Registered Users: 3

Blockchain Based IoT network for Digital ticket purchase and validation

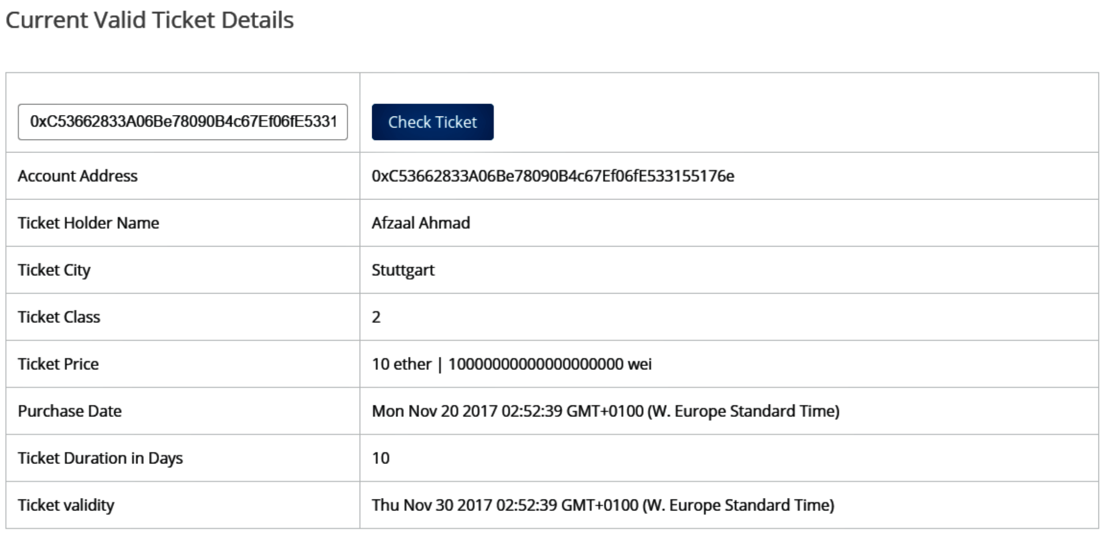
Buy Ticket with Ether

Your Transaction Hash #: 0x49f4a284e21c85bf7d9bf34443f0d1ecefcca9ce0220c5dc29cb2bb55d1e7a32

0xC53662833A06Be78090B4c67Ef06fE533155176e
Afzaal Ahmad
Stuttgart
2
10

Figure 6.6: Successful purchase transaction hash

6. Now to check the current valid ticket, just use account address no need for keyfile, account unlocking or funds. This action can be performed on any available node just with the account address as given below.

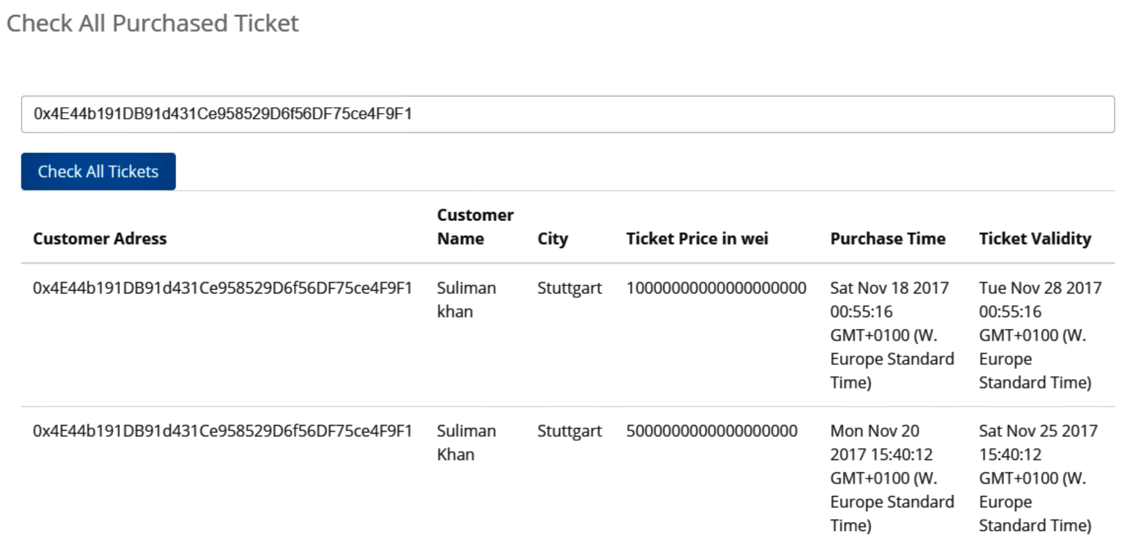


Current Valid Ticket Details

0xC53662833A06Be78090B4c67Ef06fE5331	Check Ticket
Account Address	0xC53662833A06Be78090B4c67Ef06fE533155176e
Ticket Holder Name	Afzaal Ahmad
Ticket City	Stuttgart
Ticket Class	2
Ticket Price	10 ether | 10000000000000000000 wei
Purchase Date	Mon Nov 20 2017 02:52:39 GMT+0100 (W. Europe Standard Time)
Ticket Duration in Days	10
Ticket validity	Thu Nov 30 2017 02:52:39 GMT+0100 (W. Europe Standard Time)

Figure 6.7: Check Current Valid Ticket

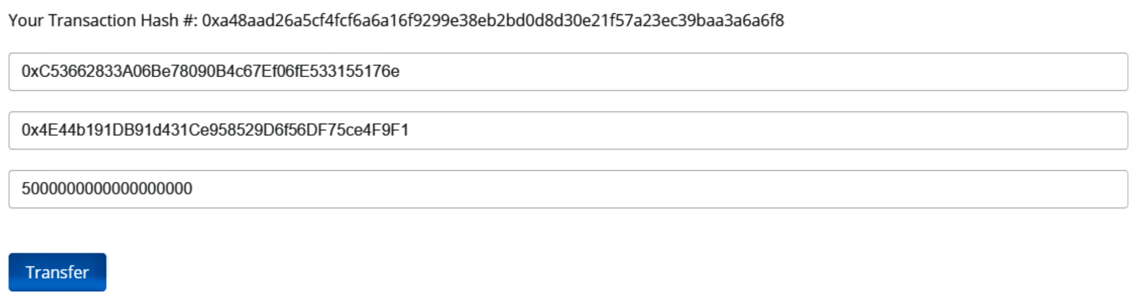
7. To check the list of current and old tickets just use account address similarly as for ticket validity check in the designated area as given in the figure below, But before that remember for the testing purpose, we omit the restriction to purchase a ticket when there is no valid ticket available. Normally a customer cannot buy a new ticket if he owns a valid ticket.


Check All Purchased Ticket
0x4E44b191DB91d431Ce958529D6f56DF75ce4F9F1
Check All Tickets

Customer Adress	Customer Name	City	Ticket Price in wei	Purchase Time	Ticket Validity
0x4E44b191DB91d431Ce958529D6f56DF75ce4F9F1	Suliman khan	Stuttgart	10000000000000000000	Sat Nov 18 2017 00:55:16 GMT+0100 (W. Europe Standard Time)	Tue Nov 28 2017 00:55:16 GMT+0100 (W. Europe Standard Time)
0x4E44b191DB91d431Ce958529D6f56DF75ce4F9F1	Suliman Khan	Stuttgart	5000000000000000000	Mon Nov 20 2017 15:40:12 GMT+0100 (W. Europe Standard Time)	Sat Nov 25 2017 15:40:12 GMT+0100 (W. Europe Standard Time)

Figure 6.8: Check Ticket History for a customer

8. To transfer ether from contract account to EOA, only contract owner can utilize this specific component of the application as given in the figure below. The first address is the contract owner address the second is the receiver address and the third value represent the amount in wei to transfer.


Your Transaction Hash #: 0xa48aad26a5cf4fcf6a6a16f9299e38eb2bd0d8d30e21f57a23ec39baa3a6a6f8

0xC53662833A06Be78090B4c67Ef06fE533155176e

0x4E44b191DB91d431Ce958529D6f56DF75ce4F9F1

5000000000000000000

Figure 6.9: Ether Transfer from Contract Account to EOA

The snapshots presented above verify that the implementation of the Dapp completely comply with the specifications and design presented in the previous chapters.

7 Conclusion and Future Work

Blockchain technology gets a hype in the Information Technology (IT) world because of its security, immutability, transparency, and decentralization as best defined in [Blo17] that "Blockchain is shared, immutable ledgers for recording the history of transactions. It fosters a new generation of transactional applications that establish trust, accountability, and transparency- from contracts to deeds to payments". But there is a lot of misconceptions about blockchain technology which are also discussed in [Ban17] such as some people think smart contract carry the same legal value as a normal contract but it's not true, the smart contract is a piece of code which resides on blockchain and execute upon meeting a certain condition. Similarly, most of the people think of cryptocurrencies when they hear the word blockchain, but blockchain is not cryptocurrency it is the underlying technology used for the cryptocurrencies and it can be used for any other non-financial applications as well. In this thesis, an effort is made to clarify from the basic blockchain concept to Dapp development based on the blockchain technology.

In chapter 2, the fundamentals of blockchain technology is presented in reference to ethereum blockchain. After studying these fundamentals most of the misconceptions about blockchain get cleared and will enable a person to start some development work on blockchain technology. In chapter 3, some of the latest related work has been presented, in which an effort is made to use blockchain for some real-world applications beyond cryptocurrencies. In chapter 4, we presented a scenario along with its model, which includes integration of IoT devices via a Dapp, which can be used for purchasing transport tickets using ether and validating the ticket on any node of the blockchain network without accessing a centralized server. In chapter 5, the architecture of the system is presented which shows all the required component needed for the realization of the system and a detail description of the possible requests and responses needed to perform all the required functions. In chapter 6, step-by-step guidelines from the private blockchain configuration to Dapp development and validation are presented.

Future Work

Despite the functionalities provided by blockchain, it certainly has some limitations, which should have to be tackled such as storing data on public blockchain is not free and not cheap as in the traditional systems; running a single line of code on blockchain cost something. The blockchain is only designed for the key-value pair, not large files or a large amount of other data format. Some efforts are made which results in a hybrid solution which uses centralize storage system with blockchain by utilizing its cryptographic functionalities by giving away the decentralization. Such systems are a step towards security improvements but do not pay back the benefit of decentralization. The blockchain is currently only use full nodes, the lighter version will soon be available which will really push blockchain usage in IoT because IoT devices are always limited in resources. So in our view, any effort in future to solve the storage problem without giving away decentralization like other solutions discussed previously, will be a step forward in the utilization of blockchain technology.

Bibliography

[Ban17] A. Banafa. *12 Myths about Blockchain Technology*. Aug. 10, 2017. URL: https://ahmedbanafa.blogspot.de/2017/08/12-myths-about-blockchain-technology.html?utm_source=datafloq&utm_medium=ref&utm_campaign=datafloq (cit. on p. 99).

[BBG+17] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, R. Sirdey. "Towards Better Availability and Accountability for IoT Updates by Means of a Blockchain." In: *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. Apr. 2017, pp. 50–58. DOI: 10.1109/EuroSPW.2017.50 (cit. on pp. 51, 53, 54).

[Bit17] Bitcoin. *Bitcoin Developer Guide*. 2009-2017. URL: https://bitcoin.org/en/developer-guide#block-chain (cit. on p. 23).

[Blo17] I. Blockchain. *What is blockchain*. 2017. URL: https://www.ibm.com/blockchain/what-is-blockchain.html (cit. on p. 99).

[BRSS17] T. Bocek, B. B. Rodrigues, T. Strasser, B. Stiller. "Blockchains everywhere - a use-case of blockchains in the pharma supply-chain." In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. May 2017, pp. 772–777. DOI: 10.23919/INM.2017.7987376 (cit. on pp. 51, 52).

[But15] V. Buterin. *Merkling in Ethereum*. Nov. 15, 2015. URL: https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/ (cit. on pp. 22, 24, 25).

[CD16] K. Christidis, M. Devetsikiotis. "Blockchains and Smart Contracts for the Internet of Things." In: *IEEE Access* 4 (2016), pp. 2292–2303. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2566339 (cit. on p. 15).

[CNSK15] M. Crosby, Nachiappan, P. and Sanjeev Verma, V. Kalyanaraman. *BlockChain Technology Beyond Bitcoin*. Oct. 16, 2015. URL: http://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf (cit. on p. 15).

[Cou17a] T. Coulter. *Truffle*. Ed. by T. Coulter. 2015-2017. URL: http://truffleframework.com/docs/ (cit. on p. 47).

[Cou17b] T. Coulter. *Truffle-contract*. 2017. URL: https://github.com/trufflesuite/truffle-contract (cit. on p. 48).

[CVM17] M. Conoscenti, A. Vetrò, J. C. D. Martin. "Peer to Peer for Privacy and Decentralization in the Internet of Things." In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. May 2017, pp. 288–290. DOI: 10.1109/ICSE-C.2017.60 (cit. on p. 15).

[DKJ17] A. Dorri, S. S. Kanhere, R. Jurdak. “Towards an Optimized BlockChain for IoT.” In: *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*. Apr. 2017, pp. 173–178 (cit. on pp. 56, 57).

[DKJG17] A. Dorri, S. S. Kanhere, R. Jurdak, P. Gauravaram. “Blockchain for IoT security and privacy: The case study of a smart home.” In: *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. Mar. 2017, pp. 618–623. DOI: 10.1109/PERCOMW.2017.7917634 (cit. on p. 57).

[DPKS17] V. Daza, R. D. Pietro, I. Klimek, M. Signorini. “CONNECT: CONtextual NamE disCovery for blockchain-based services in the IoT.” In: *2017 IEEE International Conference on Communications (ICC)*. May 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7996641 (cit. on pp. 54, 55).

[Elo17] S. Eloudrhiri. *Create a private Ethereum blockchain with IoT devices*. Ed. by S. Eloudrhiri. Feb. 24, 2017. URL: http://chainskills.com/2017/02/24/create-a-private-ethereum-blockchain-with-iot-devices-16/ (cit. on p. 85).

[Eth17a] Ethereum. *Ethereum White Paper*. 2013-2017. URL: https://github.com/ethereum/wiki/wiki/White-Paper (cit. on pp. 20, 21, 23, 26, 29, 38, 41–43).

[Eth17b] Ethereum. *Ethereum Homestead*. 2016-2017. URL: http://ethdocs.org/en/latest/introduction/index.html (cit. on pp. 27–47, 85).

[HCK17] S. Huh, S. Cho, S. Kim. “Managing IoT devices using blockchain platform.” In: *2017 19th International Conference on Advanced Communication Technology (ICACT)*. Feb. 2017, pp. 464–467. DOI: 10.23919/ICACT.2017.7890132 (cit. on p. 16).

[Hit17] R. Hitchens. *Solidity CRUD- Part 1*. Feb. 19, 2017. URL: https://medium.com/@robhitchens/solidity-crud-part-1-824ffa69509a (cit. on p. 79).

[Inn17] O. Innovation. *Oakens Innovation*. Ed. by O. Innovation. 2017. URL: https://www.oakeninnovations.com/ (cit. on p. 16).

[KM17] E. Karafiloski, A. Mishev. “Blockchain solutions for big data challenges: A literature review.” In: *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*. July 2017, pp. 763–768. DOI: 10.1109/EUROCON.2017.8011213 (cit. on p. 16).

[Mar17] J. T. Mark Otto. *Bootstrap Docs*. 2011-2017. URL: http://getbootstrap.com/docs/4.0/getting-started/introduction/ (cit. on pp. 48, 49).

[Mur17] M. Murthy. *Tools and Technologies in the Ethereum Ecosystem*. Mar. 20, 2017. URL: https://medium.com/blockchannel/tools-and-technologies-in-the-ethereum-ecosystem-e5b7e5060eb9 (cit. on p. 29).

[Nak08] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (cit. on pp. 15, 19, 23).

[OCo17] C. O'Connor. *What Blockchain means for you, and Internet of Things*. Feb. 10, 2017. URL: https://www.ibm.com/blogs/internet-of-things/watson-iot-blockchain/ (cit. on p. 15).

[SD16a] M. Samaniego, R. Deters. "Blockchain as a Service for IoT." In: *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. Dec. 2016, pp. 433–436. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData.2016.102 (cit. on pp. 58–60).

[SD16b] M. Samaniego, R. Deters. "Hosting Virtual IoT Resources on Edge-Hosts with Blockchain." In: *2016 IEEE International Conference on Computer and Information Technology (CIT)*. Dec. 2016, pp. 116–119. DOI: 10.1109/CIT.2016.71 (cit. on p. 16).

[slo17] slock.it. *slock.it Solutions*. 2017. URL: https://slock.it/solutions.html (cit. on p. 16).

[Tea16] W. developer Team. *Whisper*. Ed. by R. Trinkler. 2016. URL: https://github.com/ethereum/wiki/wiki/Whisper (cit. on p. 47).

[Tea17a] E. developer Team. *Patricia Tree*. Ed. by S. Matthew. 2013-2017. URL: https://github.com/ethereum/wiki/wiki/Patricia-Tree (cit. on p. 25).

[Tea17b] S. developer Team. *Solidity*. 2017. URL: https://solidity.readthedocs.io/en/develop/ (cit. on p. 43).

[tea17] W. developer team. *web3.js - Ethereum JavaScript API*. 2017. URL: https://github.com/ethereum/wiki/wiki/JavaScript-API (cit. on p. 46).

[W3C16] W3C. *HTML & CSS*. 2016. URL: https://www.w3.org/standards/webdesign/htmlcss (cit. on p. 49).

[Wik17a] Wikipedia. *Bootstrap (front-end framework)*. 2011-2017. URL: https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework) (cit. on p. 48).

[Wik17b] Wikipedia. *jQuery*. 2006-2017. URL: https://en.wikipedia.org/wiki/JQuery (cit. on p. 49).

[Wik17c] Wikipedia. *Cascading Style Sheets*. 1996-2017. URL: https://en.wikipedia.org/wiki/Cascading_Style_Sheets (cit. on p. 49).

[WOO17] D. G. WOOD. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER EIP-150 REVISION*. Aug. 7, 2017. URL: https://ethereum.github.io/yellowpaper/paper.pdf (cit. on pp. 27–29, 38, 43).

All links were last followed on November 01, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature