Ethereum - Concept, Architecture and Blockchain-based Applications

Bachelor Thesis

Frankfurt School of Finance & Management

presented to

Prof. Dr. Peter Roßbach

and

Andreas Reitz

by

Simon Tim Dosch

Student Id: 4924352

Mercatorstraße 5

60313 Frankfurt a. M.

Cell Phone: +49 176 57721803

E-Mail: sdosch2110@gmail.com

Frankfurt, October 2016

ABSTRACT

Simon Tim Dosch: Ethereum – Concept, Architecture
and Blockchain-based Applications

Blockchains have been around for some time now, first being introduced by the ominous creator of the Bitcoin network. Using a consensus mechanism, these distributed systems are able to retain a synchronous state of information and write changes into interlinked pieces of data called 'blocks', effectively creating a *blockchain*. The use cases range from cryptocurrencies to distributed domain name systems and have one thing in common: they do no longer need a trusted central agency. *Ethereum* aims to expose the blockchain's capabilities to the creativity of programmers. Featuring a virtual machine built from scratch that will reliably execute code on its blockchain, *Ethereum* harnesses the power of generality and aspires to become the world's most widely used platform for blockchain applications. The protocol might also be used by companies to create private or consortium blockchains. Deterministic computation plus built-in authentication and payment mechanisms forebode powerful applications to be built on *Ethereum*.

In this thesis, we want to analyze the validity of this vision, addressing questions regarding security, privacy, scalability and economic feasibility. The legal point of view comes into play when talking about accountability in cases of programmer error or malice. Solutions to these problems such as encapsulation, identity services, *sharding* and a switch to a more economic consensus mechanism are either already implemented or planned. To provide an informed opinion about the capabilities of *Ethereum* at the current state, a prototype was built demonstrating some of the core advantages of blockchain-based applications. The promising results are documented and explained in detail.

Contents

1 Introduction

In recent years we have seen a rise in "cryptoeconomic consensus networks"[1], or blockchains, like Bitcoin, Namecoin or Ethereum. These systems offer the possibility to keep tamperproof records of any data, i.e. a ledger of a cryptocurrency, and send secure transaction across the network. Now their scope is expanding to being able to manage "the back-end of arbitrary stateful applications"[2] using so called *smart contracts*. Possible applications for systems like this are financial services, payment systems, registration of identity, more secure certificate authority systems, voting systems, prediction markets or even entire distributed organisations managed democratically by their shareholders.[3] A look at pages like "State of the ĐApps"[4] shows us that many people are already working hard on developing useful distributed apps on the Ethereum blockchain.

In this thesis, the *Ethereum* blockchain protocol will be explained and insights will be given on its most important features and inner workings. There will be a discussion on possible styles of applications and the implications they might have on existing business models. Integral **achievements** but also **limitations** of the **concept** of blockchains and *smart contracts* will be determined and examined. The most crucial future **challenges** like scalability and privacy also form a central theme.

Furthermore, an in-depth look will be taken at the actual implementation of distributed applications completely on, or at least heavily relying, on the *Ethereum* blockchain. In order to do this, a **prototype** has been developed. The development process will be detailed and the prototype will be explained extensively.

[1] V. Buterin, Ethereum 2.0 Mauve Paper, 2016
[2] V. Buterin, Ethereum 2.0 Mauve Paper, 2016
[3] See V. Buterin, Ethereum 2.0 Mauve Paper, 2016
[4] See EtherCasts, 2015

2 The Ethereum Protocol

2.1 Bitcoin and state transaction functions

The first concept of a blockchain, a distributed database using a consensus mechanism, was introduced in 2008 by an unknown person with the pseudonym Satoshi Nakamoto. In his paper "Bitcoin: A Peer-to-Peer Electronic Cash System"[5], he introduced the first version of the protocol. Since then, the source code has been under continuous development by the BitcoinCore Team[6].

Bitcoin succeeded in creating a distributed network of nodes that collectively processes all transactions of Bitcoins without the need of a central entity. Sending Bitcoin is an immutable act forever recorded in the blockchain database. This allows users to trust that their Bitcoins will always be attributed to the right address, reach their destination and that other users cannot pretend to have more Bitcoin than they actually possess. Another big problem solved by the protocol was the danger of users spending the same money more than one time (*double spending*).

In his white paper "A Next-Generation Smart Contract and Decentralized Application Platform"[7], Vitalik Buterin described Bitcoin as a *state transition system*. Buterin is one of the founders and core developers of the Ethereum Protocol. He explains that, in essence, the Bitcoin blockchain serves a "*state transition function*" between two states. The first state is the initial ownership status of all bitcoins and the second is the result or output from the state transition function. Bitcoins have been reallocated and we have a new ownership status. Formally he defined this with:

```
APPLY (S, TX) -> S' or ERROR
```

Where TX is the state transaction function, S is the old and S' the new state.[8]

It is very important to realize that the Bitcoin protocol is just one of many possible implementations of a blockchain. In its case, a node can determine the amount of Bitcoin

[5] S. Nakamoto, 2008
[6] See W. van der Laan, 2016
[7] V. Buterin, Ethereum White Paper 2014
[8] See V. Buterin, Ethereum White Paper, 2014

available to an address and write Transactions containing sender and recipient as well as the amount to be sent, into a block. This is the special *state transition function* of the Bitcoin blockchain. In order to have a different functionality, one could expand the capabilities of a node to do different computations. Applications like *Namecoin* (distributed domain name system) or *colored coins* (financial instruments and custom currencies) did exactly that and developed their own special blockchain with their own *state transition function,* fitting their needs.

2.2 The Concept of Ethereum

In this section, we will try to get a firm understanding of what exactly *Ethereum* is. The network's properties and mechanisms are going to be illuminated so that later discussions about implications and issues can happen on an informed level.

2.2.1 Introduction the Protocol

The *Ethereum* developers managed to build a blockchain that is capable of all these use-cases and even more that nobody has even thought of yet. Using a built-in, Turing-complete[9] programming language that runs on the Ethereum Virtual Machine (EVM) on each node, programmers are able to build arbitrary *state transition functions*. One can basically expand the standard *state transition function* of the Ethereum blockchain by appending the execution of pieces of code (saved in the blockchain) to it. These pieces of code are commonly called *smart contracts.* Users can call functions from these *smart contracts* by sending transaction to the network. Tapping into the power of generality, Ethereum transformed its blockchain from an application to a platform. Arbitrary *state transition functions* enable Ethereum to be the backbone for any application that wants to capitalize on the unique properties of a blockchain.

[9] Turing-complete languages can store variables and have conditional repetition and jumps. See J. Purdy, 2012

Adding to the appeal of reliable, tamperproof computing, Ethereum relieves developers of two of the most important and critical parts of any web-application:

User Authentication and **Transfer of Funds**.

Ethereum uses its own built-in cryptocurrency *Ether* as a means of compensating its miners, but also as a simple tool to conduct payments in any application that is built on Ethereum. Ether funds are attributed to 20-byte addresses that are basically cryptographic keys. These are used to sign transaction to ensure their validity. For cryptocurrencies such as Bitcoin or Ether this is obviously necessary to prevent fraud, but in Ethereum this mechanism can be used for authorization in any kind of application. Reliable, comprehensive payment and user management that could be used all across the internet is something that is very appealing to web developers.

There are two types of accounts in Ethereum[10]:

- **Externally owned accounts**, secured with a private key and
- **Contract accounts**, controlled by code

"Controlled by code" means that this account is associated with a piece of code called a *smart contract.* Using *function calls* embedded into normal transactions, these *smart contracts* can be addressed by participants in the network. Ether funds can be attributed to both of the described account types. This means that *smart contracts* can receive or even send Ether if they are programmed to do so. Contract accounts can also write into an internal storage. This allows the contracts itself to handle multi-state operations. This basically means that contracts are able to remember the variables they put into the blockchain. Just like any common applications needs to store variables connected to users or the general state of the system, *smart contracts* can do the same. Buterin refers to this as the ability to handle "stateful" applications[11].

[10] See V. Buterin, Ethereum White Paper, 2014
[11] See V. Buterin, Ethereum: Platform Review, 2016, p. 16

Transactions

Let us look at how transactions are handled in the Ethereum blockchain. A transaction essentially consists of these 4 most important fields:

- **Address** of the recipient (this can be any of the two types of accounts)
- **Signature** identifying the sender
- **Amount** of Ether to be transferred with the transaction
- **Data** field (this can contain either code for the deployment of a contract or the instruction to call a specific function of the addressed contract)

Other fields are a unique **transaction hash**, **timestamp**, amount of **gas**, **gas price** and a **nonce**.

The **transaction hash** is necessary to ensure the ability to uniquely identify a transaction anywhere on the blockchain.

The **timestamp** is actually one of the most important parts of a transaction. It gives any application that uses the blockchain the power to exactly determine when changes were made in its system. Like everything else, once in the blockchain, this can't be manipulated or tampered with after the fact.

The fields concerning **gas** are necessary to enable a system of micropayments to reimburse the miners. Every transaction has a maximum amount of gas (representing a certain amount of the built-in currency ether) that can be paid out to miners that process the transaction. If it is just a simple transaction that transfers ether from one account to another, the total gas used will be very low. If the transaction talks to a smart contract, this means code will be run on the blockchain which is more expensive. The gas cap is to ensure that faulty smart contracts resulting in loops will not run forever but only until the transaction's gas runs out.

The **gas price** is a means to prioritize transactions. As the sender, I can set how much I am willing to pay for each gas unit that will be used processing my transaction. Miners will then consider those transactions with a higher gas price before others. This means the higher the gas price the quicker the processing of a transaction.

Finally, the **nonce** is just a simple counter to make sure each transaction can only be processed once.[12]

Messages

"Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment."[13] They are very similar to transactions but can also be created by contracts themselves, enabling them to communicate with each other. When a transaction calls a *smart contract's* function, a message is also created. Unlike transactions, messages are not recorded into the blockchain.

2.2.2 Consensus through Mining

When a transaction is send to a node, it is put into the transaction pool and published to the network. This way, with little delay, all nodes know about all pending transactions at any time. At one point, the transactions inside the network will be processed and the results are written into a new block that will be appended to the current blockchain. Each block contains the hash of the previous block to ensure order. If a transaction is written into a block, this proves that it has been executed. To ensure that participants of the blockchain cannot advertently alter the outcome of transactions, every node will process every transaction and see if its results match the ones other nodes have gotten. In order to compare their results, nodes share them across the network. In order to simplify this process, results are compressed into a data structure called the **merkle tree**. We will take a look at that method later on.

Now all that is left is to **decide which node writes the block first**. This is essentially what *consensus mechanisms* do. Only when participants cannot predict who will "mine" the next block do we have a secure consensus mechanism. In the case of Ethereum (and also Bitcoin), this is done with using a so called "**proof-of-work**" system.[14] It works as follows. The protocol requires the block hash to be under a certain value. Nodes will alter the nonce field of the block they want to mine, in an attempt to guess a combination that yields a

[12] See V. Buterin, Ethereum White Paper, 2014
[13] V. Buterin, Ethereum White Paper, 2014, p.
[14] V. Buterin, Ethereum White Paper, 2014

desired hash. The higher the requirement, the harder it becomes to achieve this. This is called *difficulty*. Ethereum's difficulty is automatically updated to keep the average block time at ~12 seconds no matter the computation power of the network.[15] This is a very good way to ensure a certain randomness in the mining process.

Vitalik Buterin has blogged on the topic of **block intervals**. In his post "On Slow and Fast Block Times" [16], he attempts to refute the claim that block time does not matter. Firstly, it is important to understand that we usually do not trust a transaction immediately after it has been processed and mined into a block. This is because of the possibility that two or more nodes have created a new block simultaneously without knowing about each other because of network delays. The result is a temporary split in the blockchain that will eventually be resolved depending on which chain will produce the next block the quickest. If blocks are again created simultaneously within network latency, the split persists, and so on. It therefore makes sense to wait for a certain amount of time before trusting the execution of a transaction. The question is if this time is inversely proportional to the block interval. That would mean that if my block time is 20 times faster I would have to wait 20 times the number of blocks to be mined in order trust a transaction. Buterin begins to show that slower blockchains display a slightly lower **probability of splits** because it is simply much harder to create blocks, so latency is not as much of an issue. He proceeds to explain that due their lower block interval, faster blockchains are ultimately less likely to revert transactions.[17] So they may suffer from more insecurities in the form of splits shortly after a certain transaction, but they also resolve a potential split faster than the slow blockchain. It becomes a matter of granularity. Buterin shows this in a comprehensible graph.

[15] See Ethereum Wiki – Mining, 2016
[16] See V. Buterin, On Slow and Fast Block Times, 2015
[17] See V. Buterin, On Slow and Fast Block Times, 2015

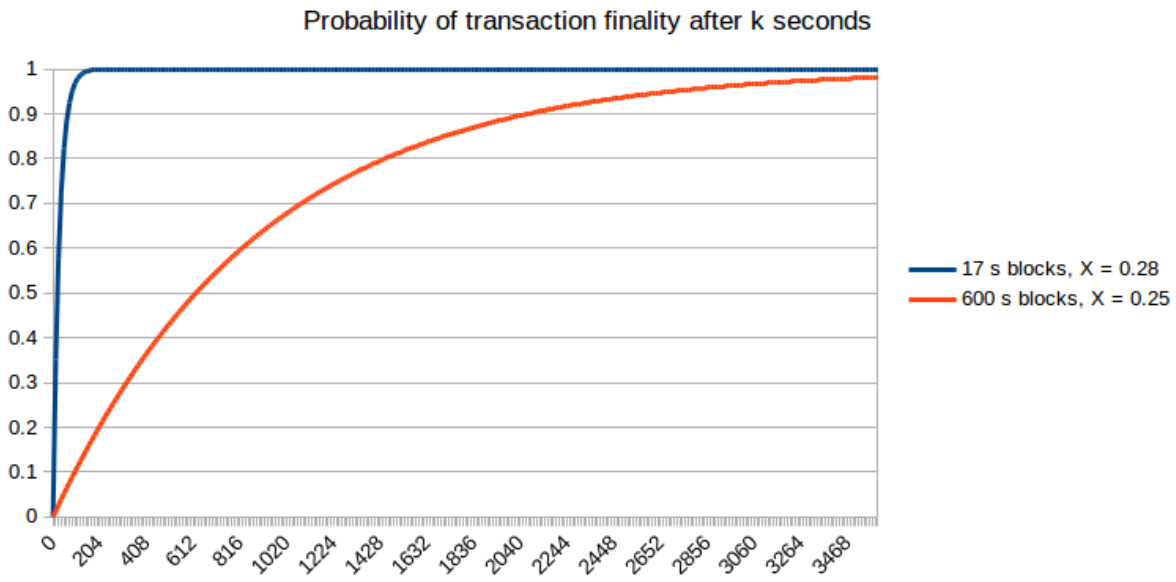


Figure 1, probability of transaction finalty after k seconds[18]

The big **downside of the *proof-of-work*** approach is the high energy consumption. The more participants a network has, the higher the *difficulty* becomes in an attempt to keep the same average block time. This implies an exponential increase in energy consumption as the network grows. Considering that scalability is already a problem because every full node processes every transaction, spending so much computation power on the consensus mechanism seems problematic.

Ethereum attempts to solve this problem by switching to the *proof-of-stake* mechanism in early 2017. To increase the incentive for miners to move to the new network, mining difficulty is going to increase even more over the next months.[19] This means that come January 2017, average block time will be close to one minute and increasing exponentially. This estimate originates from a simulation implemented in python that can be found on the Ethereum Stack Exchange forum[20]. The *proof-of-stake* consensus mechanism will be discussed later on in the chapter “Switch to proof-of-stake”.

[18] V. Buterin, On Slow and Fast Block Times, 2015
[19] See CryptoBond, 2016
[20] See 5chdn, 2016

2.2.3 Merkle Tress

As mentioned before, comparison of the nodes' results is an essential part in the *mining process* of the distributed network. When a node downloads the newest block, it "mines backwards" to check if it would have reached the same outcome. Now this has to happen very quickly as quick transaction confirmation is desirable. This is why blockchains make use of a data structure called merkle trees**.**

Merkle trees are a form of binary tree. Starting with the head node, each node has two leaf nodes. Every node is a hash calculated from the hashes of its two children. This goes all the way down to the nodes where the actual transaction hashes are being stored. Thanks to the unique properties of hashing algorithms, even the slightest changes in a transaction hash will lead to a variation of the head node.

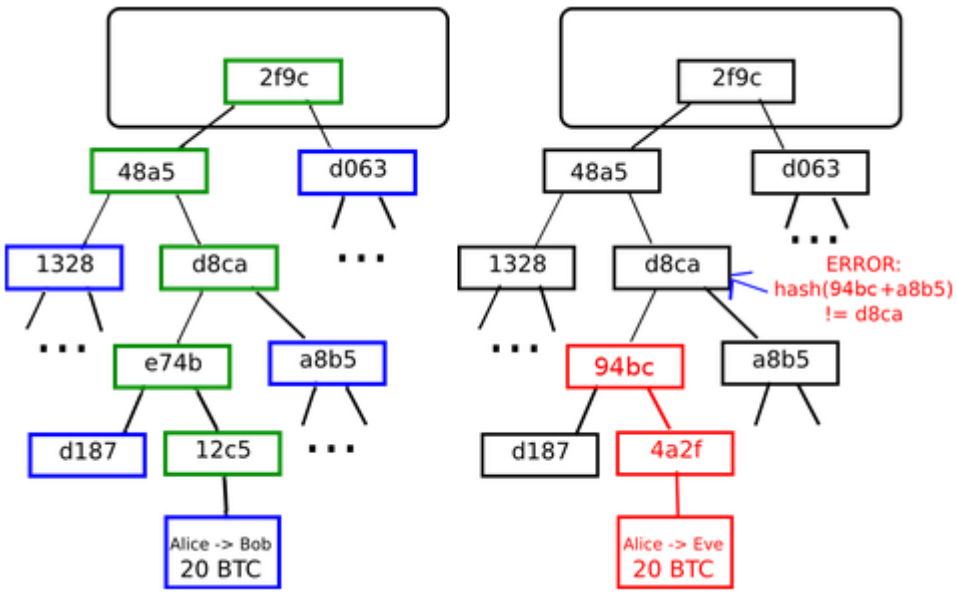


Figure 2, merkle trees – Source: https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-state-transition-function

If a malevolent user tries to alter a transaction embedded into the merkle tree, the whole tree will change and the protocol will discard the block because of it being a completely different block without the necessary *proof-of-work.* The value of this technique might become even more important in the future. So called *light nodes* will only need to download the head of a block containing the root hash of the merkle tree in order to validate a block and then download only those branches of the merkle tree that are of

interest to them[21]. A block header is roughly the size of 200-bytes, containing only a timestamp, a nonce, the previous block hash and the merkle tree root hash[22].

2.2.4 Ethereum Environment

So far, *Ethereum* has been implemented in Go, C++, Python, Java, JavaScript, Haskell, Node and .Net[23], the Go-implementation ***geth***[24] being the one most commonly used.

There is an official *Ethereum Wallet* with a GUI and also a special *Ethereum Browser* called ***Mist,*** that enables the user to directly authorize transaction in the browser using his private keys and a password. Both also work with a privately set-up test-net for development purposes.

For *smart contract* development, there is the online solidity real-time compiler[25] available which allows deploying on a simulated environment and test *smart contracts.* One can also use the ***Ethereum Studio IDE***[26] provided by *live.ether.camp*.

Blockchain applications, also called Distributed Applications or **ĐApps**, are primarily developed using JavaScript, HTML[27] and Less. The popular **Meteor** Framework supports *Ethereum* plugins. The *Mist* browser can also be used to inspect or debug the application.

These ĐApps can talk to the blockchain using i.e. the JavaScript API[28] *Web3.js* that uses JSON RPCs[29] to communicate with the an *Ethereum* node.

During the description of the ĐApp Prototype, we will go deeper into these topics.

Smart contracts

The *Ethereum Virtual Machine* is what separates the *Ethereum* blockchain from other blockchains. It implements the interpreter for the *smart contracts* users can upload into the blockchain. Functions from these contracts can be called by anybody participating in

[21] See V. Buterin, Ethereum: Platform Review, 2016, p. 5
[22] See V. Buterin, Ethereum White Paper, 2014
[23] See EtherPress, 2016
[24] See GethWiki, 2016
[25] See Chris Eth, 2016
[26] See EtherCamp, 2016
[27] HyperText Markup Language
[28] Applications Programming Interface
[29] Remote Procedure Calls

the network. The contracts are basically pieces of code. As mentioned before, there are two types of accounts on the *Ethereum* blockchain, externally owned and contract accounts. Externally owned accounts are controlled by users holding their private key. Contract accounts are managed by their corresponding *smart contracts.* This means that *smart contracts* can send and receive money according to the rules defined in their code. They are also able to store variables and thereby manage stateful operations. So we have a piece of code permanently associated with an *Ethereum address* that can store variables and send and receive money in the form of *ether.*

Solidity

The high level language that is used most for *smart contract* development is called *Solidity*[30]*.* It is very similar to JavaScript semantically, but naturally it is not as developed yet. Solidity files have the filename extension ".sol". When a contract is deployed into the blockchain, Solidity code is compiled into bytecode that the *Ethereum Virtual Machine* interprets. This code is then assigned to a new account. In order to communicate with a *smart contract,* one needs to know its address and its Application Binary Interface, short: ABI. Knowing the ABI, a user knows the names of the *smart contract's* functions, which variables a function needs to receive and which they return.

Solidity supports all the usual value types like Integers, Strings, Booleans and Arrays. Additionally, there is a type "address" which represents a 20-byte *Ethereum* address. We can get the balance of an address with `address.balance` and send money to it using `address.send(ether)`. When a function is called with a transaction, we can access information about that transaction with the `msg` object. With `msg.sender` we can get the address of the sender and `msg.value` will tell us how much *wei*[31] was send along with the transaction. Another special object is the `block` object. Through it, we can find out about the current `block.number` or its blockhash.

Deploying Solidity code to the blockchain happens in form of a normal transaction. The compiled bytecode is send via the *data* field. The *Ethereum* protocol will automatically

[30] See readthedocs.io, 2016

[31] 1 ether = 1 000 000 000 000 000 000 wei, full conversion table in Appendix B

recognize if a contract was send with a transaction, create a new address and deploy the contract. As usual, this costs a certain amount of gas depending on how big the contract is. Once created, a contract can never be altered and will forever remain in the blockchain unless a function was implemented that includes the `selfdestruct` command. It will delete the contract and return all its funds to its creator. Of course a function like that is not desirable when it comes to *smart contracts* that are supposed to manage funds independently and where not even the creator should have the power to withdraw ether that does not belong to him.

Solidity, like all of the software surrounding *Ethereum* is still under heavy development and there is a possibility of another high-level language arising and taking its place.

2.2.5 Deviation from conventional technologies

The *Ethereum* blockchain functions as a distributed database that can be altered with transactions. Once something has happened, it will forever be traceable. So it is a database that cannot be altered retroactively. It is also fault-tolerant. This means that nodes may disagree about the current state of the database. In order to resolve differences, consensus mechanisms can be used. Now this has been done before. There are fault-tolerant databases are also immune to tampering. The main difference lies in the fact that these databases all assume that there is one final, central arbiter of "the truth". Blockchains don't have that assumption. Their consensus mechanisms are designed to work, even when none of the participating parties trust each other. This is called **byzantine fault-tolerance**[32].

[32] See C. Kuhlman, 2016

2.3 Implications and Applications of Ethereum

Having grasped the concept of *Ethereum* a little better, we can now look at the different ways the protocol might be used. There is the one, official, public *Ethereum* blockchain, but there are also possibilities of creating blockchains with some different properties using the same technology. Furthermore, we want to discuss the impact such systems might have on existing business models and the way we use the internet in general.

2.3.1 Private and Public Blockchains

The following properties can be used to classify different kinds of blockchains:

- **Participation**
- **Access**
- **Consensus mechanisms**

Participation

The Ethereum blockchain is a *public* blockchain. This means everybody that installs a node on his machine and has an internet connection is able to **participate**. The very fact that there are no requirements for participation ensures the incorruptibility of the blockchain. As soon as there are restrictions on participation, we talk about a *permissioned* or *private* blockchain. This means that nodes do not look for peers automatically, but rather have to be added to the network manually.

Access

On a *public* blockchain like Ethereum, there are no restriction as to who can read the blockchain. Everything that has ever been done on or put into the blockchain is accessible to everybody. Even though many applications only store hashes in the blockchain that they later use to validate their own database (i.e. Factom[33]), it might be desirable to restrict read- or write-access to a blockchain. This applies especially to companies that deal with very sensitive information, internal accounting or auditing applications. They could

[33] See S. Higgins, 2014

implement *permissioned* blockchains with these properties and restrict public access. A blockchain like this might be open for participation, but an extra layer regulates accessibility.

Consensus Mechanisms

Most blockchains use cryptoeconomics to produce consensus.[34] Consensus is necessary because every participant of the network can create a block. This means we need to have a mechanism that tells us which block we should trust. Right now, *Ethereum* currently uses the *proof of work* **consensus mechanism.** This describes the mining process, the creation of new blocks, of the blockchain as it was described earlier in the chapter "Mining". It capitalizes on the laws of thermodynamics to show a specific amount of time has been put into the creation of a certain block.[35] Users therefore know that they can trust the longest chain with the highest amount of work put into it. Another consensus mechanism is *proof-of-stake.* Instead of mining blocks by guessing hashes and therefore committing computing power to the blockchain, participants of the network will pledge money instead. In this case, they will "stake" a certain amount of cryptocurrency that can either grow by being rewarded for honest work or be taken away in the case of malicious altering of transactions and submission of wrong blocks.

As soon as peers are being selected and added in a controlled manner we are dealing with a *private* blockchain. In his blogpost "On Public and Private Blockchains"[36], Vitalik Buterin classifies blockchains into three main categories: *Public*, *Consortium* and *Private* blockchains.

Public blockchains are open to participation for anybody in the world. One can simply install the protocol on a machine, download the current state of the blockchain and start mining and submitting transactions. All blocks can be inspected and evaluated. Public

[34] See V. Buterin, On Public and Private Blockchains, 2015
[35] See V. Buterin, On Public and Private Blockchains, 2015
[36] V. Buterin, On Public and Private Blockchains, 2015

consensus mechanisms implement cryptoeconomic principles following the assumption that the trustworthiness of a participant is proportional to the amount of capital they offer. Be it computing power in the case of *proof-of-work* or actual capital in the case of *proof-of-stake*. Buterin describes these blockchains as "fully decentralized".[37]

Consortium blockchains are a kind of *private* blockchain. This means that participants will be pre-selected. They might be a group of companies cooperating on the blockchain. Especially in the financial market where blockchains could revolutionize the clearing business, these solutions seem very likely. Instead of using standard cryptoeconomic consensus mechanisms, these consortium blockchains could establish a sort of group-sign mechanism. As an example, Buterin proposes 10 of 15 institution have to sign a block in order for it to be valid.[38] Another possibility would be altering accessibility. Making part of the blockchain available to the public would help ensuring trust and would enable easy public auditing. Buterin describes these "hybrid" blockchains as "partially decentralized".[39]

Fully private blockchains are not open to the public. Nodes have to be added manually and no one but the participating nodes can access information on the blockchain. Even though blockchains can provide a certain degree of security and auditability, having all the nodes inside one organisation defies the original idea of trust through reciprocal validation. Having control over more than 50 percent of a blockchain network enables a user to make changes retroactively.

2.3.2 Decentralization of software

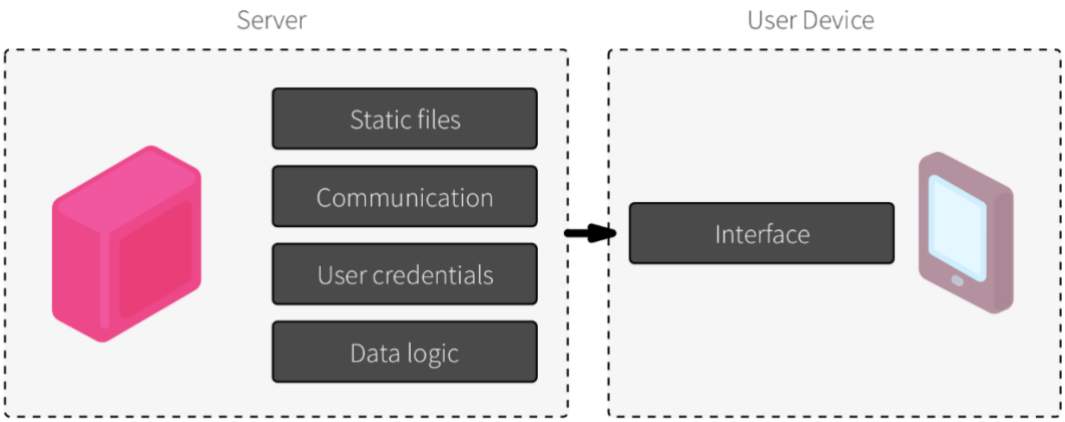
In the next sections, the possible impact of blockchain technologies on the way we use the internet and its services will be discussed.

Over the past years, advances in communication technology have enabled a radical evolution of the architecture of information systems. Coming from an unconnected stage

[37] V. Buterin, On Public and Private Blockchains, 2015
[38] See V. Buterin, On Public and Private Blockchains, 2015
[39] V. Buterin, On Public and Private Blockchains, 2015

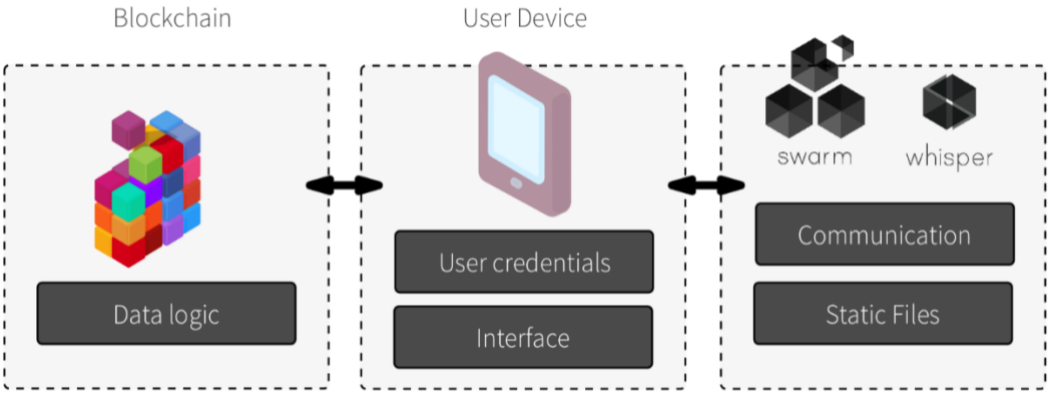
of just many standalone machines, we moved into the era of the internet. As communication speed and security increased, systems became more interconnected. Also, we were able to store less and less content on our respective devices. Online drives and streaming services have made it possible to enjoy content without needing to have a private copy of it (*cloud)*. The same is true for software. We are currently shifting from autonomous desktop applications to web applications. This has long been true for social media, streaming and other services. Nobody would think to download and install Facebook. With things like Office 360 we can see that even the most classic desktop applications are moving into the browser.

Right now, these applications are still hosted centrally. If we work with them, we need to trust the provider like Microsoft. This might be no problem when it comes to relatively innocuous applications like Office, but as soon as money, intellectual property or private information is involved, trust becomes an important aspect to manage. But there are many platforms out there that deal with exactly these entities. Decentralized *deterministic computation* the likes of which can be achieved through public blockchains like *Ethereum* could provide the trust that is needed for these applications. The result would be a shift from server-driven to client-driven applications that use decentralized services for logic and storage. Alex Van de Sande propagates this idea in his blogpost "How to build server less applications for Mist".[40] Right now, even though consumer devices are relatively powerful, most tasks are handled by a central server.



[40] A. Van de Sande, 2016

In the future, consumers could manage their user credentials locally, data logic would be handled by the blockchain and static files of websites could be stored using decentralized solutions. These will be covered under “Distributed Content”.



Tokens

Another very important aspect of this development is the usage of **tokens**. Tokens are just made-up digital entities that *smart contracts* can issue. The *smart contract* keeps track of the ownership status of the tokens. Owning a token may authorize the holder to receive a certain service. Tokens have a value and can be bought using *Ethereum’s* cryptocurrency *ether.* By not having to pay for services with *ether* directly, the market can decide how much a service is worth by determining the value of a specific token. Say I want to store something in a decentralized file system. I just buy tokens from the system equivalent to the price of storing my files and I am done. Maybe the file storage system holds tokens from a service that will send E-Mails to its users in certain situations. Because everything happens in the same blockchain, everybody still only uses one account but can still trust that he will receive the services he paid for.

This new form of immediate and convenient reimbursement of services using tokens could simplify the online market significantly and facilitate cooperation between services that now would require much more overhead.

2.3.3 Immediate interaction

Another big advantage will be the power of the immediacy of user interactions that is possible with blockchain applications. Let us look at some existing business models. YouTube brings creators and consumers of original video content together, Facebook enables people to share a social experience online and platforms like Amazon offer products from millions of different vendors from all around the world. We have seen time and time again that there can be conflicts of interest between the **provider** of the platform and its **users**. Just recently, YouTube made a major adjustment in their policies and forbid the monetization of videos with non-advertiser-friendly content.

Many advocates of blockchains and decentralized applications hope that new applications could be build that will minimize the gap between consumer and creator. Now this doesn't mean that there will be no more company-run platforms like YouTube, but instead it is the financial part that we need to take a look at.

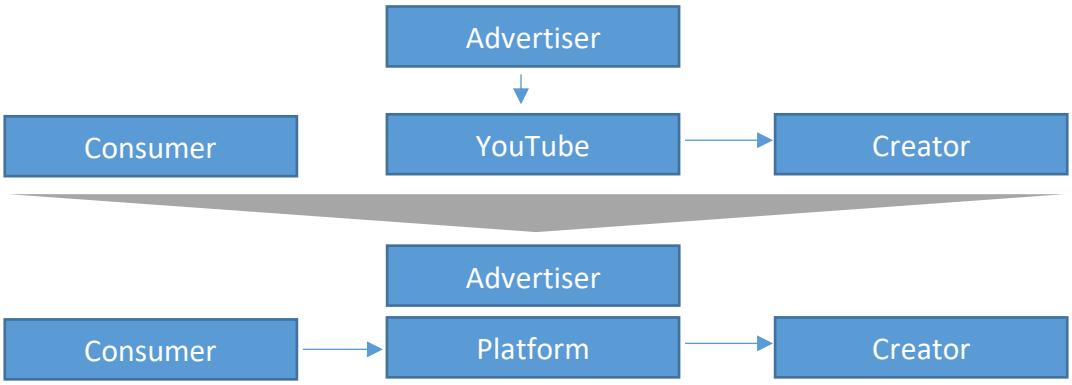


Figure 1, Flow of money determining content

Currently, the flow of money on YouTube is very indirect. Advertisers pay for ads on the platform and successful content creators can profit by letting these ads show before their videos. As we can see quite clearly in this example (Figure 1), the middleman creates a conflict of interest, as the interests of advertisers do not reflect all the interests of YouTube's customers. Content that does not satisfy the needs of both the advertisers and the customers will not be profitable.

On a similar platform that uses a blockchain as backbone, flat-rate or pay-per-use models are very quickly implemented since secure payments are easy to integrate. The consumer could trust that the money he pays goes directly to the creators of his favourite content

and the creator could trust that he is receiving his fair share depending on how successful his videos are. Advertisements are still possible and can contribute to the profitability of content, but without undermining the incentive to create original content for consumers no matter if it is advertiser-friendly or not.

2.3.4 Distributed Autonomous Organizations

When we look at the example presented above, the platform itself would still be controlled by a company. This company can make deals with advertisers and alter the platform when it wants to. As we saw, some decisions that make sense financially for the provider of the platform, cut into the overall benefit perceived by the users. This is why many developers in the community want to build applications "of the people, by the people, for the people". Now this means that changes to the platform will be democratically decided upon by the people invested into the organisation and participating on the platform. Proposing changes and voting on them would be secured by *smart contract* functionalities. For rather simple concepts like a matching platform in the housing or private retail market, this will be very feasible. Again, payments and security will be handled by the blockchain in order to achieve the same kind of trust that a big company would provide the users with. But instead of being dependent on corporate decisions, its users could ensure the prosperous continuation of the platform.

Coming back to the aforementioned evolution of the architecture of information systems, this could be regarded as the next logical step. Just like the power structure in human society, computer programs tend to move from scattered to centric, to decentralized as communication speed and security increased.[41]

Recent events have sparked discussion in the community about how organizations like these should be treated in terms of liabilities in the case of an exploit. In April 2016, somebody (we only know the *Ethereum* address) used the DAO framework developed by Slock.it[42] to create a DAO that acts basically like an investment fund. Everybody that bought a share (in this case called token) of the DAO has the right to vote on propositions on how

[41] See C. Kuhlman, 2016
[42] See C. Jentzsch, 2016

to allocate the money held by it. The profits of the investments will be fairly distributed between the token-holders.

Unfortunately, the *smart contracts* that made up the DAO turned out to be vulnerable to a very serious exploit. Deficiencies in the "splitDAO"-function that creates childDAOs, made it possible to transfer the same tokens into your new childDAO multiple times. The attacker used this to drain about 150 million dollars' worth of *DAO-Tokens* out of the main DAO into his childDAO.[43] These tokens were now no longer available to the rightful owners. The problem now was that the basic principle of blockchains dictated that no transaction can be reversed. This meant that in order to reverse the damage done and return the tokens to their rightful owners, this principle would have to be violated.

Two sides formed in the discussion. One side was defending the idea that "code is law" and that no matter what the consequences, the blockchain should not be tampered with even if it seems to be for the greater good. The other side was of the opinion that in this early stadium of the technology an intervention would be justified in order to preserve the trust that people had put in it.

Instead of making a decision themselves, the developers decided to fork the blockchain into two chains. One would support the intervention and the other would not. Everybody that participated in the network and operated a node could now decide which chain he wanted to support. This democratic process that, ironically, bears similarity to the voting process in a DAO led to a decision in the matter.

The majority of the miners supported the DAO-Fork and most of the tokens have since been claimed by their previous owners. The old chain is now called Ethereum Classic and will continue to exist as long as some miners will use it. The precedent that this case sets could be dangerous in the future though. The DAO exploit was not a problem of the *Ethereum* platform itself. It was the *smart contract* running on the platform that was vulnerable. Since the platform was not at fault, it would be unreasonable to do a hard fork on the Ethereum blockchain every time a major application that runs on it is exploited.

[43] See D. Siegel, 2016

In the next years, clear rules should be established as to who is liable for failures like the one of the DAO. There are essentially 2 possibilities:

1. Users are responsible. Investing money in cryptocurrency and blockchain applications bears risks and there will be no compensation for eventual losses.
2. Developers are responsible. As the creators of the smart contracts, developers should compensate users for losses suffered due to exploits.

Jurists like Michael Kolain want to see laws concerning these issues implemented. To him it is obvious that just improving the quality and security of smart contracts will not be enough, since exploits will always happen.[44] A legal framework on how to resolve conflicts surrounding blockchain technology seems reasonable and necessary.

[44] See M. Kolain, 2016

3 Achievements and Challenges

3.1 Achievements of the Ethereum ecosystem

Ethereum is often compared to a sort of "world computer"[45], except that it is much slower than one would expect it to be. This is because of the great amount of parallelisation. In contrast to a regular computer this does not mean doing multiple tasks at once, but rather doing the same task multiple times simultaneously. Every computation has to be done by every full node in the network. The nodes thereby ensure the validity of their peers' computations. This decentralized consensus mechanism offers a lot of really positive properties though. *Ethereum* is **byzantine fault tolerant[46]**, has **zero downtime**, and the contents of its blockchain are **resistant to censorship and retroactive modifications**.

As explained earlier, we are moving into an era where applications are becoming more decentralized and closer to an ecosystem than a traditional application. In order to build these applications, a number of challenges have to be mastered. In this paragraph, we will take a more in-depth look at the problems blockchain technology and specifically *Ethereum* can solve.

Gaining attention through the popularity of Bitcoin, it is easy to think that cryptocurrencies are the only application of blockchains. Even very generically speaking though, the blockchain offers very unique properties. We will see that almost every application that operates on an ecosystem level can benefit from these. In his post "Doing Distributed Business", Casey Kuhlman, CEO of Eris Industries offers great insight into this topic. He pinpoints the important issues that are being solved by blockchains or need solving in order to build next generation applications he calls "ecosystem applications".[47]

[45] See T. Patron, 2016
[46] See chapter "Deviation from conventional technologies"
[47] See C. Kuhlman, 2016

3.1.1 Deterministic computation

A blockchain is a system that can process transactions. In doing so, it moves from one state to the next. How exactly transactions are being processed in between states can be described as the *state transition function*. This function must be **trusted to be executed** exactly as programmed every single time, otherwise inconsistencies in the network would arise. *Deterministic computation* is a common term for this. The definition of the *state transition function* is embedded in every node participating in the network. This means that all nodes should reach the same result when processing transactions. The *Ethereum state transition function* includes the execution of code if the transaction asks for it. In the *data* field of a transaction, functions of the code associated with a contract-owned address can be called. If this is the case, this code will be executed as part of the *state transition function.* It is important to realize that not only the node that mines the next block will do this computation. For every block, all transactions and their corresponding smart contract function calls are processed by every single node. This is necessary to prove that computation is actually deterministic. Only when all nodes reach the same result and communicate those to each other, does the block count as verified. This is why one usually waits around 12 blocks before completely trusting an entry in the blockchain.[48] This is also the reason why scalability is a big issue with blockchains as it can never surpass the computation power of a single computer. This will be addressed later on in the chapter "Scalability".

Code distribution

In order to run any application, we usually need to download and install it. In an ecosystem-scale application, every participant needs to know the computational sequences necessary to run the application. This is traditionally solved by installing the software on a device or connecting to a central entity where the application has been installed. By putting logic on the blockchain in the form of *smart contracts*, there is no need to install anything. In order to perform a certain computation, we can just connect to the network and it will reliably and deterministically execute the code. The Ethereum network has often been described as sort of a "world computer" where the EVM is the processor[49].

[48] See tayvano, 2014
[49] See gitbooks.io, 2016

3.1.2 Attribution

Attribution is a very common problem of the digital age. Systems need to reliably attribute actions to users in order to be able to function properly.

Authentication in the real world is managed with documents or passports. Online, we have always had more than one identity. Traditionally, passwords or IP addresses are held by a single entity. This leads to the need for multiple usernames and passwords for every application. Also, it means we have to trust a single arbiter for authentication. For an application that operates on a network, storing password hashes would not make sense. This is why we use cryptographic technology. This way, the network knows exactly who sent a transaction. This can be of tremendous help when building distributed applications, because authentication is already taken care of by the blockchain. Instead of having many different usernames and passwords, one would really only need its private key in order to use any application built on top of the Ethereum blockchain.

Ordering transactions is another important task. The blockchain, which consists of a number of independent nodes, has to distribute pending transactions within itself, process them and write them into a new block, thus moving to the next state. The network has to know the exact order in which transactions have been received. This is what the timestamp are used for. Ordering blocks is achieved by including the last block's blockhash into a new block's header, interlinking them to create the blockchain.

Combined, knowing which user sent which transaction at what time gives the blockchain the power of *attribution.*

3.1.3 Transfer of value

As mentioned in the chapter "Immediate Interaction", the seamless and secure transfer of cryptocurrencies can change the way we interact with applications. Thanks to the immutability of the blockchain, keeping score of ownership (especially when it comes to digital assets) is one of the capabilities of the network. *Ethereum* features the built-in cryptocurrency *ether*. It is used to pay for transaction's *gas* costs and to reimburse miners. Using *smart contracts,* it is very easy to create new currencies or tokens that can represent anything from monetary value to ownership of a painting. Having applications that do not

need to rely on third party payment services and allow almost direct payment seem very attractive. This is one of the achievements that our prototype will show quite neatly.

In the chapter "Decentralization of Software" we already touched on the power of tokenized services. Here, the market would decide the price for tokens that enable the owner to use a certain service. This will again be important in the next section. Without secure and comprehensible transfer of value, a *token economy* would not be possible.

3.1.4 Distributed Content

Following the "world computer" metaphor, computation is not the only important part of a computer. Content has to be stored and distributed efficiently. As mentioned before, storing content on a local device is becoming more and more obsolete. This is why there are a lot of efforts to combine Ethereum with distributed storage technologies.

Projects like Swarm or **IPFS** are combining **content addressable storage** with decentralized networks in order to create a reliable and secure way to store content online. Right now, when we are calling a webpage, we download data from a single server where all the files for that website are being kept. The Secure Socket Layer attempts to ensure that files are not swapped on their way and that the identity behind the website is legitimate. To make entirely sure that we have the right file, we sometimes compare its hash with one that the server provided. Instead of using the location of data to address storage like traditional systems, *content addressable storage* directly uses those hashes of data to find them. IPFS stands for InterPlanetary File System. It uses cryptographic hashes to uniquely identify data and stores it in a decentralized manner using blockchain technology. IPFS just recently announced that they will be using the *Ethereum* blockchain[50]. Content will not be held by all nodes simultaneously, as that would consume too much storage space. Instead, nodes only store the content that they need and duplicates are systematically removed by the protocol. When a node needs a file, it simply sends a request to the network asking who holds the data corresponding to a certain hash. The service will be paid for with special tokens following the idea of a tokenized reimbursement system we touched on earlier.

[50] See J. Young, 2016

To summarize, in order to build trustworthy distributed ecosystems, this is what we need and how we could achieve it:

Problem	Tool
Deterministic computation	Smart contracts
Attribution	Elliptical curve cryptography
Code Distribution	Smart contracts
Distributed Content	Content addressable storage
Ordering	Timestamps

Table 1, Requirements for ecosystem applications and their implementation

Lastly, let us have a look at what kind of **tasks** a protocol like *Ethereum* has to handle from a **technological point of view**.

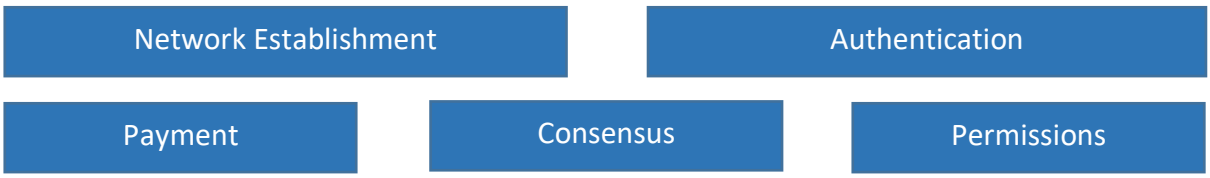


Figure 3, Features of a distributed network

Kuhlman thinks that in order to achieve greater flexibility when it comes to blockchains, these tasks should not all be handled by the same system. He proposes a different model where standardized interfaces enable the substitution of any of those features so that it will be handled by a different system.[51] Leveraging *Docker*, this is exactly what Eris Industries is trying to achieve.

[51] See C. Kuhlman, 2016

3.2 Challenges awaiting the Web 3.0

We have discussed multiple visions of what future applications, networks or storage systems could look like. The term "Web 3.0" is being used a lot to describe this new distributed phase of the Internet.[52] But before we get there, there are still a lot of obstacles to face. Let us take a critical look at the difficulties awaiting the forecited technologies.

3.2.1 Scalability

One of the biggest concerns when it comes to blockchain technology is scalability. On a blockchain like *Ethereum,* every transaction has to be processed by all the nodes participating in the network. The transaction processing capacity of a blockchain will therefore never exceed that of a single node.[53] This is necessary in order to ensure consensus. Even switching to *proof-of-stake* will not change this as nodes will only be relieved of the mining process. If *Ethereum* would become widely used, transaction volume would increase drastically resulting in a growth of the blockchain up to the point where it would be too big to be stored by consumer devices. Having a blockchain with a size of 50 TB would mean that only a few, probably professional, users could actually store the blockchain. All other users would use **light nodes**.

Light Simplified Payment Verification (SPV) nodes, using the power of merkle trees, will only need to download a fraction of the blockchain to verify their transactions by relying on the trustworthiness of the full nodes. Now it would of course be problematic if these full nodes would become fewer and fewer and were only run by big companies who could still afford it. It would undermine the principle of distribution of the blockchain.

The high transaction volume itself is also a problem. Due to risks of miner centralization or non-validation[54], there is a soft cap of 10tx/sec in the *Ethereum* blockchain. In private blockchains the full capacity of ~1000tx/sec could theoretically be reached. It still flails in

[52] See T. Gerring, 2016
[53] See V. Buterin, Ethereum: Platform Review, 2016
[54] Further explanation in the glossary under "miner centralization risk" and "non-validation risk"

comparison to, say, the approximately 80 000tx/sec process by the Shanghai Stock Exchange.

In his platform review of *Ethereum,* requested by the R3 Consortium, Buterin offers two solutions: **sharding** and **state channels**.[55]

He describes **state channels** as "a generalization of Bitcoin-style 'lightning-networks', which seek to conduct most transaction off the blockchain between parties directly, using the blockchain only as a kind of final arbiter in case of disputes".[56]

Sharding seems to be the solution that will be implemented soon. In his mauve paper, Buterin describes how the sharding technique will start to be realized in the next release, *Ethereum 2.0.*[57] In the realm of blockchain, parallelization means replicating rather than splitting. Sharding attempts to bring the power of parallelization to blockchains by splitting the blockchain into multiple "shards". So the state of the blockchain will no longer reside on a node as a whole but rather be held partially by many nodes. Transactions will be routed to the nodes that hold the state they will affect. Since the computation power of a blockchain actually decreases with the number of nodes[58], it only makes sense to split it up and have the different "shards" communicate with each other.

3.2.2 Privacy

Another big challenge is the issue of privacy. Due to the nature of consensus mechanisms, every node processes every transaction in the network. This means that all information surrounding a transaction is public knowledge. It is important to distinguish between two kinds of security concerns, *authenticity* and *privacy*. The former can be provided by blockchains. There is cryptographic authentication, deterministic computation and an immutable state that can be audited. Buterin suggests that if a company's greatest concern is that of privacy, in the sense that it wants to protect certain data, the best solution would still be a secure server and not a blockchain.[59] Having a private blockchain that only me as

[55] See V. Buterin, Ethereum: Platform Review, 2016
[56] V. Buterin, Ethereum: Platform Review, 2016
[57] See V. Buterin, Ethereum 2.0 Mauve Paper, 2016
[58] See V. Buterin, Ethereum 2.0 Mauve Paper, 2016
[59] See V. Buterin, Ethereum 2.0 Mauve Paper, 2016

a company can read, would mitigate the problem, but one party having control over all nodes defeats the original purpose of a blockchain. The result would be a blockchain that would offer neither *authenticity* nor increased *privacy*.

The first solution that comes to mind, would be to simply encrypt all information that is held on the blockchain as well as all transactions so that users can only decipher those that are dedicated to them. This technique is called *cryptographic obfuscation*. The problem is that obfuscating the entire state of the blockchain would likely lead to an unbearable increase in processing time.[60]

Buterin goes on to describe some of the most promising and feasible solutions that would bring a certain amount of privacy to *Ethereum*.

One approach are solutions that try to mitigate the possibility of users being able to link transactions and thereby finding out about the activities of other users. One could implement a technique where a new account is used for every transaction. Another would be multiple participants joining their funds into one transaction so that other users are not able to determine exactly what happened. This is called ConJoin. Other techniques include "ring signatures, ZK-SNARKs and secret sharing".[61]

There is one problem that all these solutions have in common. As a financial institution i.e., I not only want to but have to know certain private information about the people that conduct business on my platform. Nowadays, the bank knows all kinds of private information about me such as name, ID number, age, address or other accounts. This is necessary to satisfy legal regulations. Right now, *Ethereum* addresses are anonymous, with no additional information attached to them. **Obfuscating one's actions on the blockchain is hard to combine with the need to "know your customer".** A solution would be to have this information on the blockchain as well, with a cryptographic mechanism implemented that enables users to prove their association with certain private information when necessary. Buterin points out that the challenges of implementing such a system would be

[60] See V. Buterin, Ethereum: Platform Review, 2016, p. 35
[61] See V. Buterin, Ethereum: Platform Review, 2016, p. 38

rather legal then technical, as every institution is traditionally required to handle their own customer data[62]

3.2.3 Security

Naturally, a system that allows users to publish code that all participants are forced to execute does come with certain security risks. Ethereum decided to develop its own Virtual Machine with a focus on high security. The EVM has practically no way of communicating with its host computer. Opcodes only interact with system variables defined in the *Ethereum* state, like memory, stack, storage or blockchain info such as block number or timestamp.[63] The Go, C++ and Python implementations have been heavily audited for security flaws and issues have been fixed. So far, except for a recent transaction spam attack[64], *Ethereum* has been running without any major issues. Infinite loops are prevented by the need for gas to execute *smart contracts.*

The DAO-Hack was more an issue of the *smart contracts* themselves. This topic is discussed in depth in the chapter "Distributed Autonomous Organisations". To solve problems arising from programmers' errors or intentional misrepresentation of *smart contract's* function, Buterin suggest two solutions[65]. The first being high-level measures like professional *smart contract* audits by law-firms and programmers and also forced dialogues whenever a transaction is performed. These dialogues would contain detailed information about the transaction that the user is about to authorize and already exist in the *Ethereum* browser *Mist*. The second proposed solution focuses on including *formal verification* into the development process of *smart contracts* to prevent mistakes prophylactically. *Formal verification* describes the technique of using algorithms to test assumptions made about other pieces of code.

[62] See V. Buterin, Ethereum: Platform Review, 2016, p. 7
[63] See V. Buterin, Ethereum: Platform Review, 2016, p. 10
[64] See V. Buterin, Transaction spam attack: Next Steps, 2016
[65] See V. Buterin, Ethereum: Platform Review, 2016 p. 11

3.2.4 Limitations of smart contracts

Smart contracts are one of the key elements of the *Ethereum* blockchain. They enable programmers to design arbitrary *state-transition-functions* to be executed on the blockchain. There are many things that can be achieved through *smart contracts*, but it is imperative to understand their most crucial limitations before developing a ĐApp.

Gideon Greenspan, founder and CEO of Coin Sciences (MultiChain) wrote an interesting article about this topic.[66] He highlights three main points, one of them evolving around the issue of confidential data. Since this topic was covered in the last paragraph, we will focus on the other two.

Calling external services

One of the first ideas that come to mind when learning about blockchain is to develop a contract that behaves in a certain way depending on **external factors**. For example, am application where I can bet on the weather the next day or the result of a soccer game. The problem with these kind of applications is that they require information from outside the blockchain. Since the blockchain needs to be deterministic, this presents a serious problem. Say a contract would consult a standardized web service about the soccer result and use this information to determine its behaviour. Every node processes the transaction, executes the code and calls the web service. The web service can only answer to one call at a time. This is fine as long as the response for every call will be the same. But what if for some reason the web service shuts down halfway through the request. Half the nodes would get no valid result while the other half executed the contract using the received results. This sort of un-deterministic behaviour would endanger the blockchain and is therefore not possible. Instead, so called *oracles* could be used that would reliably publish information into the blockchain for it to be used by *smart contracts.*

Another problem would be calling an external API. Naturally, the API should not be called individually by every node as that would cause the called function to be executed multiple times and nodes would get different results depending on whether the call was successful or not. Here, the solution would be using so called *EventLogs*. Using *EventLogs*, *smart contracts* can indicate to applications that monitor these logs that a certain event has taken

[66] See A. Greenspan, 2016

place. This means instead of calling the API, the service would monitor the blockchain for a specific *EventLog* and then execute the call itself.

Both these solutions require an outside party that needs to be trusted, thus the trust created by the blockchain does not apply here. In the end, interactions with the blockchain are limited to simple write (*oracle*) and read (*EventLogs*) operations.

Enforcing payments

The other application often mentioned for *smart contracts* is that of automated financial products. A bond that automatically and deterministically pays out coupons sounds very practical but actually is not. In order for the contract to be able to reliably make those payments, all of the money would have to be held by the contract itself. This would mean it would not be accessible by any other party during that time period, defeating the original purpose of a bond. If the money would not be held by the *smart contract* directly, there would be no guarantee for successful automated payments.

3.2.5 Switch to proof-of-stake

As mentioned, the *Ethereum* network plans to switch to *proof-of-stake* in early 2017, because of the high energy consumption and the need for indefinite mining reimbursements required by *proof-of-work* mechanisms. Vitalik Buterin, the creator of Ethereum, recently published a paper called "Ethereum 2.0 Mauve Paper", where he describes his proposed solutions to *Ethereum'*s problems like efficiency, throughput and scalability, *proof-of-stake* being the solution to the first one. Let us try to understand what he proposes.

The *proof-of-stake* consensus mechanism is not about mining, but rather about **validation** itself. Every validator owns a stake in the network. This is represented by a certain amount of money or in this case cryptocurrency that they pledge. The more a validator pledges to the network the more value is attributed to his validation of a block. A stake can be viewed as a sort of collateral that ensures the trustworthiness of a validator. When a participant of the network validates a block, it means that he processed all the transaction to the best of his ability. This is due to the fact that a malicious validator's stake would be seized by the protocol in case of wrong validations.

Ethereum will use the so called *Casper* implementation of the *proof-of-stake* mechanism. A "Casper contract" will be created that will keep track of all the validators. It is essentially a normal *smart contract*, with two exceptions. Calling the contract becomes a fixed "part of the process for validating a block header".[67] Also, the contract is part of the genesis block. Validators can send money to the *Casper smart contract* and thereby buy a stake in the network. The contract will randomly select a validator for the next block. If this validator is not available, the next in line will create the block instead. Successful validation will be rewarded by increasing the amount "staked" by the validator. Penalties will be given for dishonest creation of a block and deducted from the stake.

A problem here is that randomness is actually really hard to achieve on a blockchain. *Deterministic computation* being one of the cornerstones of the *Ethereum protocol,* it is only reasonable that pure randomness is not allowed in *smart contracts.* If validators achieve different results executing a *smart contract* due to randomness, the protocol could not achieve consensus. Of course there are ways around this, like using the hash of the current block. It is not predictable yet still identical for all validators for that time period. Using the current blockhash is just a very easy example and while it does provide some sort of *pseudo-randomness,* it can be manipulated by miners. This issue will be discussed later in the chapter "The contract", where a detailed description of the prototype's *smart contract* will be given. Buterin links to one of his own texts, "Validator Ordering and Randomness in PoS"[68], where he analyses several better ways to achieve *pseudo-randomness.*

The *Casper smart contract* uses the RANDAO as a source of randomness for selecting the next validator.[69] It capitalizes on the unpredictability of group behaviour by including all participants in the process of *random number generation* (RNG). The selection of the next validator is also weighed by the amount that a validator has pledged to the contract. This means that the validation of a participant that "staked" a lot of ether is valued higher than that of one that only "staked" a small amount. Of course this makes sense as a higher stake means the validator has more to lose and demonstrates a high interest in keeping the blockchain clean, because he wants to keep his money. To conclude, the *Casper smart*

[67] V. Buterin, Ethereum 2.0 Mauve Paper, 2016
[68] V. Buterin, Validator Ordering and Randomness in PoS
[69] See Y. Qian, 2016

contract possesses a list of validators and randomly selects who will mine the next block. The risk of being penalized or losing ether because of an attack incentivises validators to act in a responsible way. The described mechanism is the foundation for a new trust model, but without the expensive process of mining.

3.3 Timeline of Ethereum

Let us examine the brief but eventful history of *Ethereum*[70] [71].

Ethereum was first mentioned by founder Vitalik Buterin in 2013 and formally described in the Ethereum White Paper. At the beginning of 2014, it was officially announced at the North American Bitcoin Conference in Miami. The first Proof of Concept was released shortly after. Dr. Gavin Wood joined Buterin in his efforts and published the Ethereum Yellow Paper in April 2014. From July 22nd to September 2nd, an *ether* Crowdsale kick-started the project financially. In the following year, the first official release, *Olympic,* was released in May, followed by *Frontier* in July and finally *Homestead* on March 14th 2016. It is planned get the next release, *Metropolis,* out this year. Along with some minor improvements, it will feature the first official release of the *Mist* browser. It will be the gateway for non-technical users to interact with existing smart contracts. After that, the switch to *proof-of-stake* will take place with the *Serenity*

[70] See T. Gerring, 2016
[71] See J. Donnelly, 2016

release expected to happen in early 2017. The planned implementation of a faster *Ethereum Virtual Machine 2.0* written in WebAssembly is also expected to happen that year. The following planned releases *Ethereum 2.0* and *3.0* have not been named yet and will mostly be dealing with improving scalability of the *Ethereum* blockchain.

4 Developing a Ðistributed Application

As part of this thesis, a small prototype was developed that uses the *Ethereum* blockchain to implement a simple lottery game. A detailed description of the application as well as a depiction of its creation process will be given in the following paragraphs. Positive and negative aspects uncovered during development will be highlighted and explained. We will start by exploring the general concept of the prototype and talk about the ways it showcases key advantages of the *Ethereum* blockchain. Next are the setup of the *private test-net* and creation of the *smart contract* the application will use. The application itself and the framework that was used will be examined and in the end, a number of future consideration and other things worth mentioning will be talked about.

4.1 The Idea

The most promising use-cases for blockchain-based applications all have one aspect in common. They are dealing with easily measurable information. As mentioned in the chapter “Limitations of smart contracts”, dealing with information outside of the blockchain can be difficult. Even the result of a football game would have to be provided by a trustworthy third party. There is no deterministic way for the blockchain to get such data on its own.

That is why this prototype does not deal with such information, but rather attempts to show the blockchains ability to create trust. What was built is basically an online lottery. Users can buy tickets and the application will draw a winner who is then able to claim the whole jackpot. The game is played in rounds, each round lasting a certain number of blocks (the easiest way to estimate time on the blockchain). When the “decision-block” has been reached, a winner is determined. The likeliness of winning the jackpot is proportional to the amount of *ether* sent to the contract. If my balance is 20% of the jackpot, I have a 20% chance to win. All this will be done using *Ethereum’s* built in cryptocurrency *ether*.

Assuming the user is able to understand the *smart contract* language *Solidity*, he will be able to verify that the creator of the contract and the application has no way of accessing the funds associated with it. He will see that the drawing mechanism is indeed fair and that he will be able to claim the *ether* in the jackpot if he wins. Without the usage of the

Ethereum blockchain, no one would dare using an online lottery he has never heard of, but now the user is able to trust the application.

The application is designed to work within the *Ethereum Browser "Mist"*. *Mist* offers the ability to send signed transaction directly in the browser. While the user will be able to see the application with live data, using a different browser will render the interactive elements useless. An error pane will alert the user to this fact. The decision to design the application for usage with *Mist* was a conscious one, as easy authentication and transfer of funds are two of the most important advantages intended to be showcased with this prototype. Embracing the whole *Ethereum* ecosystem as intended by its creators seems the only reasonable way of assessing it properly.

This means in order to use the ĐApp with the working title "Lethery", the user needs to download *Mist*. It is available for Linux, MacOS X and Windows and includes a built-in *geth*-node that will download and manage the blockchain. The user can then create an account using the Ethereum Wallet ĐApp and start using other ĐApps with it.

4.2 The Blockchain

The first thing necessary when developing a distributed application for *Ethereum* is setting up a private test-blockchain. By using a private blockchain, we can avoid having to download large amounts of blockchain data. We also will not need to spend real money on *gas* when deploying or testing our *smart contract*. The *Mist* browser can very easily be used with a private *test-net*. That way we can deploy *smart-contracts* using the Wallet GUI[72] if we want to. There also exists a public test-net named "Morden" where developers can test their contracts in a realistic environment without having to use real money.

In order to start a private blockchain, we need to download an *Ethereum* client implementation. The Go-implementation *geth* was an obvious choice as it is the one most commonly used. All development was done on a virtual machine (VM), to have a clean working environment. Ubuntu was chosen as an operating system for the VM. It is light-weight and was supported by *geth* from the beginning. Installing *geth* is fairly simple:

[72] Graphic User Interface

```
> sudo apt-get install software-properties-common
> sudo add-apt-repository -y ppa:ethereum/ethereum
> sudo add-apt-repository -y ppa:ethereum/ethereum-dev
> sudo apt-get update
> sudo apt-get install ethereum
```

A file has to be created using the JSON[73] format. It contains the genesis block, the first block of our blockchain. It sets certain properties like difficulty or gas-limit of the blockchain we are about to create (difficulty will be automatically adjusted to achieve the desired average block interval[74]). It might look like this and we call it "gen.json":

```
{
    "coinbase": "0x0000000000000000000000000000000000000000",
    "difficulty": "0x20000",
    "extraData": "0x",
    "gasLimit": "0x2FEFD8",
    "mixhash": "0x00000000000000000000000000000000000000647572616c65787365646c6578",
    "nonce": "0x0",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "timestamp": "0x00"
}
```

Initiating the blockchain will set this file as the genesis block:

```
> geth init gen.json
```

It is useful to run *geth* as a daemon so the blockchain does not halt when the terminal is closed and we can use the *geth* console while also monitoring the log separately. The tool "Screen" was used to achieve this. Now we can start *geth* by simply running a script:

```
#!/usr/bin/env bash
echo "Starting geth"
cd ~/
screen -dmS geth /usr/bin/geth --verbosity 3 --rpc --rpcapi 'web3,eth,debug'
       --rpcport 8545 --rpccorsdomain 'http://localhost:3000' --nodiscover --mine
```

Figure 4, startup.sh (line-brake only for visibility reasons)

We can see that *geth* is given certain starting parameters. Verbosity refers to the logging level. Then we tell it to provide certain RPC-APIs on port 8545 to the domain *http:localhost:3000* which is where our application is going to be. An important parameter is `--nodiscover`. If set, *geth* will not look for peers automatically. If we want additional nodes to work on our blockchain, we are going to have to add them manually, making our

[73] JavaScript Object Notation
[74] See V. Buterin, Ethereum: Platform Review, 2016, p. 4

blockchain truly private. Lastly, we want *geth* to immediately start mining, so we can start sending transactions to it. This parameter was not there the first time *geth* was started, because every node needs a *coinbase*, which we have to define first. The *coinbase* is the address that will receive *ether* rewards when a block is mined by our node. We can create accounts using the *geth* console and set the coinbase before we start mining. Typing `screen -r` will connect us to the *screen* session that was just started by running the start-up script. In another console window, we can attach to the running *geth* instance using `geth attach`. Now we are in the *geth* console and can set the *coinbase*:

```
> personal.newAccount('password')
> eth.coinbase = eth.accounts[0]
```

Once the *coinbase* is set, we can start the mining process and our blockchain is ready to go. We can now access it using RPC and we can open *Mist* to see if it will automatically detect and connect to our private test-net as it should.

4.3 The Contract

The blockchain is the base for every distributed application. Using *Ethereum's* ability to run code in the *Ethereum Virtual Machine*, we are able to create arbitrary *state-transition-functions* that fit our application's needs. This is the *smart contract*. *Solidity* is the high-level language that is most commonly used to write *smart contracts*. Although one should keep a local copy for security reasons, the online *real-time solidity compiler* is a very good tool to quickly implement *smart contracts*. Instant feedback when errors are made and a standardized interface that lets you test your contract's functions make it a very useful tool. The aforementioned *Ethereum Studio* was not yet used in this project.

Let us now take an in-depth look at the format and content of the *smart contract* that was developed for this prototype. The first thing we see is the definition of some global variables. The first one being *owner* which is of the type *address* and will be set to the creator of the contract in constructor function.

```
3  contract Lethery{
4
5      address owner;
6      uint256 price = 1000000000000000000;
7      uint256 nonce = 0;
8      uint256 decisionBlock;
9      uint256 interval = 100;
```

Price is needed to convert the received value from *wei* to *ether* so that the contract does not deal with numbers that are too high for a 256-bit unsigned integer. Unsigned Integers can hold the same amount of numbers as signed Integers, but go from 0 upwards[75]. The type `uint` is a synonym for `uint256` and could also be used here[76]. The *nonce* is just a utility that helps with *random number generation* (RNG). The *decisionBlock* is the number of the block where the next drawing will be possible. *Interval* is the length of a round in blocks. Right now it is 100 which equals approximately 24 minutes. This is good for testing purposes but can of course be changed.

```
11      struct Round{
12          address winner;
13          uint256 winNr;
14          uint256 roundNr;
15          uint256 jackpot;
16          address[] addresses;
17          mapping (address => uint256) contributions;
18      }
19      mapping(uint256 => Round) rounds;
20      uint256 rn = 0;
```

Next, a *structure* named *Round* is being defined. Similar to a Java-Object, attributes can later be accessed using `Round.attribute`. The data of every round will be stored in a structure like this. Using an *array* to store *addresses* that contributed and the amount that they contributed in a *mapping* makes it easy to access information later. The second *mapping* is used to store *rounds*. The current round number is stored in the *uint256 rn*.

```
23      function Lethery(){
24          owner = msg.sender;
25          //set blocknumber of first drawing
26          decisionBlock = block.number + interval;
27      }
```

[75] See Fabricio, 2012
[76] See readthedocs.io, 2016

This function has the same name as the contract itself which makes it the *constructor*. The constructor is the function that is called when the contract is created. It sets the variable *owner* to the sender of the message that created the contract. Additionally, it sets the first decision block to the current block number plus the predefined interval.

```
29      function commit() payable{
30          //amount of full ether commited
31          uint256 amount = msg.value/price;
32          //record amount and address
33          rounds[rn].contributions[msg.sender] += amount;
34          rounds[rn].addresses.push(msg.sender);
35          //increase jackpot by amount
36          rounds[rn].jackpot += amount;
37          //return excess ether
38          msg.sender.send(msg.value%price)
39      }
```

The commit function is how money can be sent to the contract. Notice that after the name of the function there is an identifier called *payable*. It indicates that this function is able to receive money with a transaction. Before the introduction of the *payable* identifier, one could send *ether* in every transaction, no matter which function was being called. This meant that *ether* that was accidentally sent to a contract address could not be retrieved if the contract did not have a function to do that. So now, when *ether* is sent with a transaction that calls a *smart contract* function, an error will be thrown if the function is not *payable*. In the *commit* function, the funds received are being converted from *wei* to *ether* and excess is sent back to the sender. The amount that was committed and the sender's address are added to the *contributions mapping* of the current round. Since we also want to have an iterable object of all the addresses that committed *ether*, we push the sender's address in the *addresses array*. Also, the *jackpot* is adjusted to reflect the changes. Normally, and understandably, a user would hesitate to send money to a service like this, but thanks to what was described as "deterministic computation" in the chapter "Achievements of the Ethereum blockchain" he can trust that the *ether* he sent will actually be registered in the contract.

The ***drawWinner*** function is how the smart contracts determines the winner of the current round. *Smart contracts* have the ability to do deterministic computation. They cannot,

however, execute functions automatically. That means that right now there is no way to implement an automatic drawing process directly in the *smart contract*. In this prototype, this problem is circumvented by keeping track of the next *decisionBlock* and allowing anyone to call the *DrawWinner* function as soon as it is reached. A good improvement might be to reimburse the caller for the *gas* costs. Another solution would be to have an external program calling it automatically when it is time.

```
42      function drawWinner(){
43          if (block.number < decisionBlock){
44              return;
45          }
46          if(rounds[rn].jackpot == 0){
47              return;
48          }
49          uint etherCount = 0;
50          uint addressCount = 0;
51          rounds[rn].winNr = rand();
52          while(etherCount < rounds[rn].winNr){
53              address contr = rounds[rn].addresses[addressCount];
54              etherCount += rounds[rn].contributions[contr];
55              addressCount++;
56          }
57          rounds[rn].winner = rounds[rn].addresses[addressCount-1];
58          decisionBlock = block.number + interval;
59          rn++;
60      }
```

The way the drawing process works is quite simple. After checking if the *decisionBlock* has been reached and if the jackpot contains *ether*, the function calls the *private* function *rand* to get a random number. Just like in other object-oriented programming languages, *private* means that the function can only be called from within the same class/contract. We will examine the *rand* function later in this paragraph. The random winning number will be a number from 0 to the amount of the jackpot. The function then iterates through all the contributions, adding them together successively. Once the winning number has been reached or exceeded, the address associated with the last contributions is declared the winner. The new *decisionBlock* is set and the round number is incremented by one. The winner of this round can now be read by calling another function called *getWinnerOfRound* and the owner of the winning address can claim the jackpot.

This is the *rand* function. It provides the *drawWinner* function with the winning number.

```
62      function rand() private returns (uint){
63          var blockHash = uint256(block.blockhash(block.number));
64          nonce++;
65          return uint(sha3(blockHash+nonce))%(rounds[rn].jackpot);
66      }
```

As discussed in the chapter *Validation*, blockchains only allow *pseudo-randomness,* as real randomness would conflict with *deterministic computation*. An easy way to achieve *pseudo-randomness* is using the blockhash of the current block. Here, the blockhash is increased by our contracts nonce after which the sha3 hash of the result is being calculated. Using the modulus operator, we get a number between 0 and the jackpot of that current round. While this method is very easy, it bears a substantial risk. A miner that mines the new block would have an incentive to discard it if it doesn't make him the winner, thus increasing his chances to win. It would be possible for him to check the result of the drawing and then decide whether he wants to submit the block or not. Of course this only slightly increases his chances to win as there are many other miners out there who might mine the block in question, but it is still a problem making the game less fair. To address this issue, a switch to more reliable sources of *pseudo-randomness* is advisable. One option would be the use of the RANDAO that was also mentioned in the chapter "Validation". The RANDAO is a DAO harnessing the unpredictability of group behaviour to generate random numbers which other *smart contracts* can then use. It was also suggested by *Ethereum* developer Nick Johnson and is definitely something to look at if the ĐApp was deployed on the real *Ethereum* network. For this prototype though, using the current blockhash will suffice for the moment.

The last big function is called *redeem* and can be called by the winner of a round to collect on his winning.

```
68      function redeem(uint256 round) {
69          if (msg.sender != rounds[round].winner){
70              return;
71          }
72          //send money and dont forget to multiply with price
73          if(!rounds[round].winner.send(rounds[round].jackpot*price)){
74              return;
75          }
76          rounds[round].jackpot = 0;
77      }
```

It tests if the sender of the message equals the winner of the round that is given by the sender. If it checks out, the function proceeds to pay out the jackpot of that round to its winner and the jackpot is set to zero. The pay-out was designed following the “pull”-principle in order to increase transparency for the winner. Again, the amazing thing here is that authentication is provided by the *Ethereum* network. We can trust that if we receive a transaction from a certain address it could only have been sent by the holder of its corresponding private key. This is what was described as *attribution* in the chapter “Achievements of the Ethereum blockchain”.

A very important convention can also be observed here. The return value of the low-level call that sends the jackpot to the winner is instantly used in the if-statement. So when the call fails, the function will return and not delete the jackpot. This is a precaution to prevent attacks that exploit the **callstack depth limit** of *Ethereum*. The callstack depth is fixed at 1024 frames and if the call that was supposed to send the *ether* is the one that exceeds this limit, it will fail. A maleficent user could trick others into calling the function at the end of a very long call stack. If the contract would not check if the *send* call was successful and still erase the jackpot, the money would be lost.

So far, all the functions (except the private *rand* function) involved operations that altered the state of the blockchain by changing variables. All these functions need to be called via transactions and require *gas* to be executed. For a web application we need to be able to update certain data continuously and it would be very tedious to have to pay *gas* when making simple “read”-requests. This is why Solidity has included the ***constant*** identifier. It indicates that no state change is happening inside a function and that it can be called

without consuming gas. When talking about the *Web3.js* API in the chapter "The App" we will see how it handles *constant* functions differently. The *Lethery.sol* contract has six *constant* functions providing the following information:

Balances of an address in a round	The **jackpot** of a round
The current **round number**	The **winning address** of a round
The **winning number** of a round	The next **decision block**

Table 2, constant functions of the Lethery.sol contract

A good example is the function *getMyBalance*, which can be seen here:

```
79      function getMyBalance(address a, uint256 round) constant returns(uint){
80          return rounds[round].contributions[a];
81      }
```

The function receives an address from the sender and the number of the round in question and returns and amount in *ether*. An interesting thing to mention is that unlike JavaScript, Solidity requires its functions to declare the data types they return. This function takes the received variables and accesses the *rounds mapping* to get the *Round* object from which it then gets the contribution of address *a* using the *contributions mapping.*

This last function called *remove* should not be included in the final contract but is useful for testing purposes.

```
107     function remove(){
108         if(msg.sender == owner){
109             selfdestruct(msg.sender);
110         }
111     }
```

If the sender is the owner/creator of the contract, this function allows him to destroy the contract. None of its functions will be able to be called after its destruction. All remaining funds held by the contract are sent to the address indicated, in this case the sender of the transaction.

4.4 The App

After seeing the capabilities of the developed *smart contract*, we will now look into how we can use it in a web application. After setting up the private blockchain, developing the *smart contract* and deploying it on the blockchain via the *Ethereum Wallet*, the application can now talk to it using JSON-formatted Remote Procedure Calls (RPCs), as allowed during configuration.

The Meteor framework was used to develop the app itself. It is a very lightweight framework with which JavaScript-based web applications can be quickly built and tested. To edit the relevant files, the Atom text editor for Linux was used. It also is open source and easy to use. As time was of the essence, these tools seemed like the right choice. The Meteor framework uses templates to simplify developing, but the code that will be shown here can be easily understood without being familiar with Meteor. Git is used as a version control system with the repository publicly accessible on GitHub[77].

The folder structure is quite simple. We have the *Lethery.sol smart contract* and the *meteor* folder which contains all the files concerning the app itself. There are meteor packages especially designed for *Ethereum*. They include certain libraries or stylesheets that can then be used in the project. The most important package has to be *ethereum:web3* which supplies the *Web3.js* API that our app. We will use it to communicate with the blockchain using RPCs. *ethereum:dapp-styles* predefines certain html-elements to match the design of the *Mist browser* and the *Ethereum Wallet ĐApp*, which is quite useful. Other packages offer easy access to information about *blocks*, *accounts* and other elements of the *Ethereum* ecosystem. Once all these packages are added, we can start programming.

In the *client* folder, we find the *lib* folder, *main.html, main.js* and *main.less*[78]. The first thing that will happen when the app starts is all the scripts inside the *lib* folder will be loaded. This is why the *init.js* file is in there. It creates a new Web3 instance and loads the *smart contract* that the app wants to interact with.

[77] https://github.com/simonDos/Lethery
[78] see glossary entry “Less“

```
if(typeof web3 === 'undefined')
  web3 = new Web3(new Web3.providers.HttpProvider('http://localhost:8545'));

contractAddress = '0x842cBBd6CE9C2Dd4B038BEA23654fa61501376c5';
contractABI = '[ { "constant": false, "inputs": [], "name": "commit", "outputs":
lethery = web3.eth.contract(JSON.parse(contractABI)).at(contractAddress);
```

The contract's application binary interface (ABI) continues outside of this screenshot. It details all the functions of a *smart contract* and is therefore quite long. It is important to parse it to JSON format before using it. The last command creates an object named *lethery* that is capable of calling all the functions defined in the *smart contract*.

As mentioned before, when talking about the contract code, *Web3* distinguishes between regular and *constant* functions, the difference being that *constant* functions will be executed as message calls instead of transactions (see chapter "Ethereum"). This means that we can call *constant* functions as often as we like, enabling a more responsive user interface for the application. The way this is done can be seen in one of these asynchronous requests using the *lethery* object.

```
var getNextDecision = function(template) {
  lethery.getDecisionBlock(function(e,r){
    if(!e) {
      var currentBlock = EthBlocks.latest.number;
      if(r.c[0] < currentBlock){
        r = 'now';
      } else{
        r -= currentBlock;
      }
      TemplateVar.set(template, 'nextDecision',r);
    }
  });
};
```

This function is automatically called every second and handles the result of the asynchronous call *getDecisionBlock* with a callback function. It checks for an error *e* and then sees if the result *r* is smaller than the current block. It then sets the *TemplateVariable nextDecision* to *'r'* or *'now'* accordingly. The current block is obtained using one of the other imported libraries, *EthBlocks*. In the *onCreated*-part of the main template the function is first called initially and then the interval is set.

```
  getNextDecision(template);
  this.nextDecisionIntervalId = setInterval(function() {
    getNextDecision(template);
  }, 1000);
```

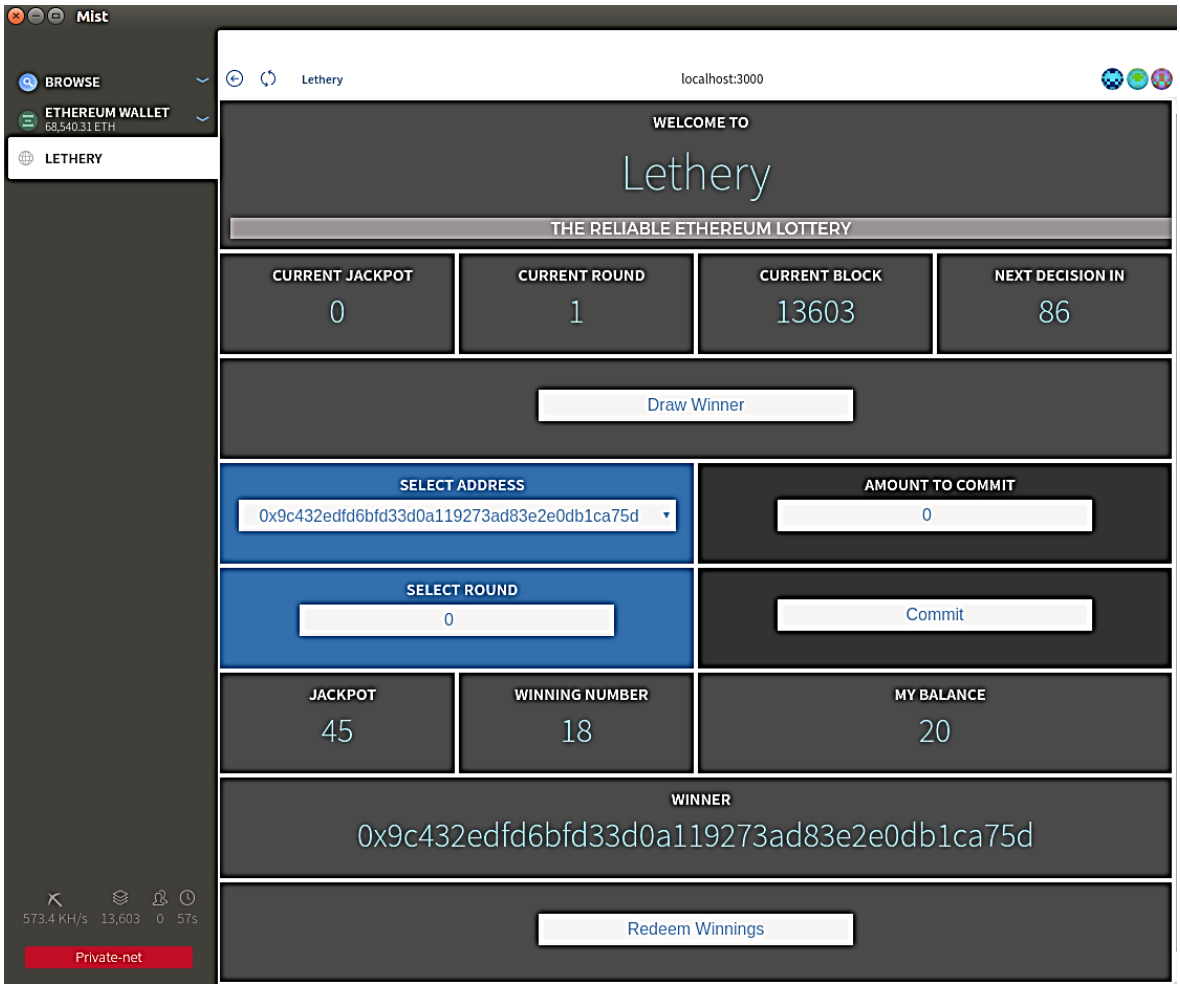
In the *main.html* this is referenced by using two pairs of curly brackets and the *TemplateVar* object. This way, the information can be directly linked in the HTML code.

```
37      <div class="infobox small">
38        <h3>Next Decision In</h3>
39        <p id="decision" class="bigInfo">{{TemplateVar.get 'nextDecision'}}</p>
40      </div>
```

Because we use the *Mist browser* as a base for the application, information about authorized accounts can be requested. In *Mist*, when browsing a ĐApp, the user can reveal none to all of his accounts to allow certain information to be used. This application uses that information to populate a dropdown menu from which the user can then select an address he wishes to use. This is done in the following function using the *Web3.js* API again.

```
29  var getActiveAccounts = function(template) {
30    web3.eth.getAccounts(function(e,r){
31      if(!e) {
32        r = r.toString();
33        activeAccounts = r.split(',');
34        //populate select
35        var options = '';
36        for(var i=0;i<activeAccounts.length;i++){
37          options += '<option value"'+activeAccounts[i]+'">'+activeAccounts[i]+'</option>';
38        }
39        $('#addressSelect').append(options);
40      }
41    });
42  };
```

Whenever a transaction is triggered by the user interface, *Mist* will automatically open up a window where the user confirms his identity and intention by typing in his password. So far, there was no need for that. With the received information from the blockchain, the user gets an overview of relevant data without having to sign a transaction yet.



This is a screenshot of the ĐApp with some informational panes cut out to make it fit better. All the information visible here has been obtained from the smart contract by calling *constant* functions. The user can dynamically change these calls by selecting different round numbers or addresses.

It gets interesting though when a user interaction triggers an event that requires a signature. As can be seen at the top right of the screenshot, the user has authorized three different accounts to be seen by the ĐApp. He can now choose with which one he would like to commit some *ether* to the current game. He selects an address, adjusts the amount he wants to commit and presses the “Commit”-Button. This triggers a function that can be seen here.

```
136  Template.mainboard.events({
137    'click .sendEther'(event) {
138      var value = $('#amount').val();
139      if(addressSelected != 0){
140        lethery.commit({from: addressSelected, value: web3.toWei(value, 'ether')}, function(e,r){
141          if(!e) alert("Commit successful !\nChanges will be visible after next block is mined");
142        });
143      } else {
144        alert("You need to select an address!");
145      }
146    },
```

The function *commit* can be examined in the chapter "The Contract". It requires no variables to be sent to it, yet it receives a lot of information as can be seen in the screenshot. The first variable is a JSON object with two elements, *from* and *value*, indicating the address from which the transaction is supposed to be sent and the amount of *wei* to be sent with it. The selected address is obtained by a global object that is updated whenever the user changes the selection. The value is obtained using jQuery[79] to read the value of the element with the *id* "amount" in the *main.html* file. The second variable is, again, a callback function specifying what will happen once a response was received. If no address has been selected, the user is given an alert reminding him to do that. Once the transaction was successfully sent, the user receives a message alerting him to the fact that changes might take a minute to show in the user interface, as they are only written into the blockchain when the next block is actually mined.

4.5 Considerations

Having seen all the most important parts that make up the application and its testing environment, one can assert that build a working ĐApp is certainly possible at this state of the *Ethereum* network. It will definitely become more streamlined with the proliferation of helper *contracts* like the RANDAO or other *smart contracts* that will provide standardized functionalities. Existing examples of this would be registering a *smart contract* in the GlobalRegistrar[80] service or having it inherit certain basic functions by importing other *smart contracts*.

[79] JQuery is a very common JavaScript Library that simplifies client-side scripting
[80] See G. Wood, 2015

As a general design principle, it is advisable to keep in mind that computation on the blockchain is quite expensive. So when developing an application, one always needs to ask the question what really needs to be done on the blockchain and what can be realized on other layers of the system. Most of the time it really is a small array of functionalities that ensure the reliability of a whole application.

Using the web3 API evidently greatly simplifies dealing with the blockchain and *smart contracts*. It handles the construction of transactions and communication with the blockchain via JSON RPCs. *Mist* is responsible for dealing with authorization using the built-in *Ethereum Wallet*.

Applications that do not use a local *Ethereum* node are also possible. They can interact with a remote node by signing transactions before sending them. With a JavaScript library called *ethereum-tx.js*[81], transaction can be signed offline according to *Ethereum* standards to be then sent to a node. Before real light nodes become reality, this would be a way to bring *Ethereum* to smaller devices like mobile phones that could not handle a local node due to space constraints[82]. One issue that is being dealt with in these kind of projects is that, right now, communication via RPCs is still using http instead of https, which would make cross-server communication insecure.

In the end, the prototype shows how easy it can be to build trustworthy applications from scratch capitalizing on the powers of the blockchain. *Lethery* does not need proprietary user accounts and passwords. It also does not need the users bank account information or collaboration with any other payment service. The user can rely upon the fact that the application will handle his contributions just like it is programmed to do and all the information regarding every round ever played will be available in the blockchain where it cannot be tampered with. All of these advantages are achieved by the *Ethereum* network.

[81] See wanderer, 2016
[82] See Pacs, 2016

5 Retrospection and Outlook

Ethereum is a very promising platform, that has the potential to change the way we use the internet forever. A reliable, distributed authentication system shared by important applications, the convenient transfer of funds, the ability to let the market decide the price of a service using tokens or building entire organisations managed democratically by individuals all over world are some of the great advantages that would come with it. At the end of the day, blockchain technologies will surely improve systems in the financial and accounting sector, but how far the revolution goes has yet to be seen. As pointed out, serious limitations when communicating with other systems make many use cases seem unlikely to be realized. Our prototype demonstrated some of the clear achievements of the *Ethereum* ecosystem, but also quickly encountered difficulties like randomness or automation.

Ethereum is still in a very early stage, but we have seen that many of the most crucial concerns are being recognized and currently addressed by its developers. Most notably, the planned switch to a *proof-of-stake* consensus mechanism will make the blockchain much more economic. After that, addressing issues like scalability and privacy, but also legal questions will be biggest challenges to master in the future.

Maybe in a couple of years, thanks to smart wallets and DAOs, embezzlement will be a thing of the past, we will be able to build platforms that work for its users instead of its creators and consumers can reclaim control over their personal data. There is no doubt that, if *Ethereum* succeeds, it will sound the bell for a new era of the internet.

TODO

Appendix (CODE, Technologies used (with versions))

Source index

Image captions + refereces

Glossary (mist browser, …)

Eidesstattliche versicherung!

improve description text on website!

examine contract code on webpage + link to github

Appendix

Glossary

[Insert glossary here]

Miner centralization

Non-validation

Less

Appendix A: Table of used technologies

TODO

Technology	Version
geth	
Meteor	
Screen	
Atom	
Mist browser	
Ethereum Wallet	
Solidity real-time compiler	

Table 3, used technologies and versions

Appendix B: Ether conversion table

1 Ether =	
wei	1 000 000 000 000 000 000
Kwei	1 000 000 000 000 000
Mwei	1 000 000 000 000
Gwei	1 000 000 000
szabo	1 000 000
finney	1 000
ether	1
Kether	0.001
Mether	0.000 001
Gether	0.000 000 001
Tether	0.000 000 000 001

Table 4, Ether conversion table

Appendix C: Code of the prototype

Insert screenshots here

Bibliography

5chdn. (2016, May 10). *When will the difficulty bomb make mining impossible?* Retrieved from Ethereum Stackexchange: http://ethereum.stackexchange.com/questions/3779/when-will-the-difficulty-bomb-make-mining-impossible

Buterin, V. (2014, Apr 13). *A Next-Generation Smart Contract and Decentralized Application Platform.* Retrieved from Github: https://github.com/ethereum/wiki/wiki/White-Paper#bitcoin-as-a-state-transition-system

Buterin, V. (2014, Nov). *Ethereum White Paper.* Retrieved from https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-state-transition-function

Buterin, V. (2015, Aug 07). *On Public and Private Blockchains.* Retrieved from https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

Buterin, V. (2015, Sep 14). *On Slow and Fast Block Times.* Retrieved from https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/

Buterin, V. (2016, Sep 16). *Ethereum 2.0 Mauve Paper.* Retrieved from http://vitalik.ca/files/mauve_paper3.html

Buterin, V. (2016, Jun 14). *Ethereum: Platform Review.* Retrieved from https://static1.squarespace.com/static/55f73743e4b051cfcc0b02cf/t/57506f387da24ff6bdecb3c1/1464889147417/Ethereum_Paper.pdf

Buterin, V. (2016, Jul 27). *Transaction spam attack: Next Steps.* Retrieved from Ethereum Blog: https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/

Buterin, V. (2016). *Validator Ordering and Randomness in PoS.* Retrieved from http://vitalik.ca/files/randomness.html

CryptoBond. (2016, Sep 12). *What is the Ethereum Ice Age?* Retrieved from CryptoCompare: https://www.cryptocompare.com/coins/guides/what-is-the-ethereum-ice-age/

Donnelly, J. (2016, Mar 14). *Ethereum Blockchain Project Launches First Production Release.* Retrieved from CoinDesk: http://www.coindesk.com/ethereum-blockchain-homestead/

Eth, C. (2016). *Browser Solidity.* Retrieved from GitHub: https://ethereum.github.io/browser-solidity/#version=soljson-latest.js

EtherCamp. (2016). *Ethereum Studio.* Retrieved from https://live.ether.camp/eth-studio

EtherCasts. (2015, Jun 12). *State of the DApp.* Retrieved from http://dapps.ethercasts.com/

EtherPress. (2016). *Ethereum Implementations.* Retrieved from Ether Press: https://ether.press/ethereum-implementations/

Fabricio. (2012, Jan 28). *The real difference between 'int' and 'unsigned int'.* Retrieved from Stackoverflow: http://stackoverflow.com/questions/9045436/the-real-difference-between-int-and-unsigned-int

Gerring, T. (2016, Jul 11). *Building The Decentralized Web 3.0.* Retrieved from Ethereum Blog: https://blog.ethereum.org/2014/08/18/building-decentralized-web/

Gerring, T. (2016, Jul 11). *Cut and try: building a dream.* Retrieved from Ethereum Blog: https://blog.ethereum.org/2016/02/09/cut-and-try-building-a-dream/

GethWiki. (2015, Aug 15). *go-ethereum.* Retrieved from GitHub: https://github.com/ethereum/go-ethereum/wiki/geth

gitbooks.io. (2016). *What is Ethereum?* Retrieved from Gitbooks: https://ethereum.gitbooks.io/frontier-guide/content/ethereum.html

Greenspan, G. (2016, Apr 17). *Why Many Smart Contract Use Cases Are Simply Impossible.* Retrieved from http://www.coindesk.com/three-smart-contract-misconceptions/

Higgins, S. (2014, Nov 19). *Factom Outlines Record-Keeping Network That Utilises Bitcoin's Blockchain.* Retrieved from CoinDesk: http://www.coindesk.com/factom-white-paper-outlines-record-keeping-layer-bitcoin/

Jentzsch, C. (2016, Jul 12). *Slock.it DAO Framwork.* Retrieved from GitHub: https://github.com/slockit/DAO

Kolain, M. (2016, Sep 07). *Lehren aus dem DAO-Hack.* Retrieved from http://www.golem.de/news/lehren-aus-dem-dao-hack-wieso-smart-contracts-die-erwartungen-enttaeuschen-muessen-1609-123020-5.html

Kuhlman, C. (2016, Jun 05). *Doing Distributed Business.* Retrieved from Eris Industries: https://db.erisindustries.com/eris/2016/06/05/ecosystem-applications/

Laan, W. v. (2016). *Team.* Retrieved from BitcoinCore: https://bitcoincore.org/en/team/

Lion, T. (2016, Jul 31). *Randao.* Retrieved from GitHub: https://github.com/randao/randao/blob/master/README.md

Ma, T. (2016, Sep 02). *Ethereum Wiki - Mining.* Retrieved from GitHub: https://github.com/ethereum/wiki/wiki/Mining

Nakamoto, S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System.* Retrieved from Bitcoin.org: https://bitcoin.org/bitcoin.pdf

Pacs. (2016, May 30). *An Ethereum API for Android App Developers.* Retrieved from Medium: https://medium.com/@pacs_IT/an-ethereum-api-for-android-app-developers-3f46b820f8f6#.pd8d4bump

Patron, T. (2016, Mar 13). *What's the Big Idea Behind Ethereum's World Computer?* Retrieved from CoinDesk: http://www.coindesk.com/whats-big-idea-behind-ethereums-world-computer/

Purdy, J. (2012, Jan 29). *What makes a language Turing-complete?* Retrieved from Stackexchange: http://programmers.stackexchange.com/questions/132385/what-makes-a-language-turing-complete

Qian, Y. (2016, Jul 31). *RANDAO.* Retrieved from https://github.com/randao/randao/blob/master/README.md

readthedocs.io. (2016). *Solidity.* Retrieved from https://github.com/ethereum/solidity/blob/develop/docs/index.rst

Sande, A. V. (2016, Jul 12). *How to build server less applications for Mist.* Retrieved from Ethereum Blog: https://blog.ethereum.org/2016/07/12/build-server-less-applications-mist/

Siegel, D. (2016, Jun 25). *Understanding The DAO Attack.* Retrieved from CoinDesk: http://www.coindesk.com/understanding-dao-hack-journalists/

tayvano. (2014, Apr 14). *What number of confirmations is considered secure in Ehtereum?* Retrieved from Ethereum Stackexchange: http://ethereum.stackexchange.com/questions/319/what-number-of-confirmations-is-considered-secure-in-ethereum

wanderer. (2016, Jul 17). *ethereumjs-tx.* Retrieved from GitHub: https://github.com/ethereumjs/ethereumjs-tx

Wood, G. (2015, Apr 7). *Registrar ABI.* Retrieved from GitHub: https://github.com/ethereum/wiki/wiki/Registrar-ABI

Young, J. (2016, Sep 24). *IPFS Protocol Selects Ethereum Over Bitcoin, Prefers Ethereum Dev Community.* Retrieved from The Cointhelegraph: https://cointelegraph.com/news/ipfs-protocol-selects-ethereum-over-bitcoin-prefers-ethereum-dev-community