



universität
wien

BACHELORARBEIT / BACHELOR'S THESIS

Titel der Bachelorarbeit / Title of the Bachelor's Thesis

„Implementing a Private Blockchain“

verfasst von / submitted by

Mathias Niehoff

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Bachelor of Science (BSc)

Wien, 2020/21 / Vienna, 2020/21

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

UA 033 521

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Bachelorstudium Informatik

Betreut von / Supervisor:

Dr. techn. Belal Abu Naim

Mitbetreut von / Co-Supervisor:

Professor Co
Second Row

Abstract

This paper describes the basic building blocks yielding a blockchain system. The ledger is distributed among the participating peers which act according to agreed upon standards encoded within a smart contract. A single block contains a list of transactions with the orderer's (miner's) signature. Each block is linked to its previous block by using the transaction data, the previous block's hash and a timestamp as inputs when creating its own hash - that way the blocks are intertwined with each other (hence "chained") guaranteeing the integrity of the ledger. Manipulating an existing block's data in any way yields the side effect of changing its hash and making corrupted blockchains identifiable hence making the blockchain immutable. A block's transactions are hashed within a Merkle tree structure inheriting its various performance benefits. This work also demonstrates my custom implementation and showcases a private consensus algorithm - proof of authority (PoA) - where only the network owner (*admin*) is authorised to order new blocks.

Kurzfassung

Diese Arbeit beschreibt die Grundbausteine einer *Blockchain*. Die Hauptbuch bzw. die Liste der vorhandenen Transaktionen (der *Ledger*) ist auf allen Teilnehmern verteilt um ein peer-to-peer Netzwerk zu schaffen, was das Kernkonzept einer Blockchain darstellt. Alle Teilnehmer agieren ausserdem auf Basis gemeinsamer Regeln und koennen daher einen Konsens finden. Ein Block (in der Kette der Bloecke) beinhaltet eine Liste von Transaktionen die weiters die digitale Signatur ihres Herstellers (*Orderer*) traegt. Jeder Block und dessen *Hash* stehen, anhand von Transaktionsdaten, den Vorgaenger-Hash und einen Zeitstempel, in Verbindung mit dessen Vorgaenger. Dadurch sind alle Bloecke miteinander verknuepft und garantieren die Integritaet der Daten bzw. des Ledgers. Die Manipulation der Daten eines existierenden Blocks fuehrt zur Aenderung seines Hashes und macht die manipulierte Blockchain daher sofort erkennbar. Die Transaktions-Hashes werden ausserdem zu einem Merkle-Baum ghasht um dessen Performanzvorteile, hinsichtlich Zugriffszeit etc., zu nutzen. Weiters zeigt dieses Dokument die Teile meiner spezifischen Implementierung und den dabei verwendeten Konsens-Algorithmus proof of authority (PoA); hierbei ist nur der *Admin* dazu berechtigt neue Bloecke zu generieren.

Contents

Abstract	i
Kurzfassung	iii
List of Figures	vii
Listings	ix
1 Introduction	1
1.1 Origins	2
1.2 Consensus	2
2 Related Work / State-of-the-Art	3
2.1 Bitcoin (public)	3
2.2 Hyperledger Fabric (private)	4
2.3 Ethereum (public)	6
2.3.1 Ethereum 2.0	6
2.4 Others	7
3 Design and Implementation	9
3.1 Access Permission	9
3.2 the App	10
3.3 the Network	11
3.4 the Ledger	14
3.5 persistent, local storage	22
4 Further Discussion	23
4.1 Scalability Trilemma	23
4.2 Forks	24
4.2.1 the DAO Fork	24
5 Conclusion	27
5.1 Future Work	27
Bibliography	29

List of Figures

3.1	UI	15
3.2	proposing a transaction	16

Listings

3.1	generating/issuing a digital identity (key pair)	10
3.2	signing a tx proposal generating digital signature	11
3.3	listening for incoming multicast packets	11
3.4	multicasting a handshake/discovery packet	13
3.5	processing a handshake/discovery packet	14
3.6	receiving/processing a tx proposal (Controller)	16
3.7	transmitting a tx proposal packet	16
3.8	receiving/processing a tx proposal packet	17
3.9	receiving/processing a tx endorsement packet	17
3.10	receiving/processing an endorsed tx proposal	18
3.11	attempting to order a new block	18
3.12	assembling a new block (Smart Contract)	19
3.13	generating a Merkle root	19
3.14	casting a chain sync request	20
3.15	processing and responding to a chain sync request	21
3.16	processing a chain sync response	22

1 Introduction

Democratisation of data is a big feature that comes with blockchains. In the digital age we live in, transparency of critical data, privacy protection, autonomy and the need for maintenance and integrity of the data is a necessity that has to be met. In opposition to a centralised client-server network where the entity behind the server often holds the right to distribute and use the personal data of clients, a decentralised peer-to-peer (P2P) network can tackle such injustices and fill many of the above mentioned needs. In a P2P system each participant acts as a client and a server. Distributing the critical data across all participants obviously removes the single point of failure, enables transparency and also distributes workload. On the other hand it is much harder to secure the full system across all nodes and guarantee data integrity along the way. The participants have to reach Consensus on the critical data i.e. keeping the global state up to date where updates have to be approved by the majority or an appropriate amount of participants. A blockchain extends the P2P concept by paying special attention to the security aspect. Deploying Consensus protocols and making them tolerant to Byzantine Fault (BFT). The Byzantine Generals Problem is based around the situation where multiple parties have to cooperate in order to reach a common goal with the premise of not trusting each other. A system fulfills BFT if the common goal (Consensus) is reached although some (a minority) of the parties are acting counterproductive. For a blockchain system BFT means that the network can still reach Consensus if the number of maliciously acting nodes stays below a certain threshold.

To make a blockchain even harder to tamper with it packs chunks of data into blocks and links them together in an immutable fashion. Any change in already recorded data blocks would propagate through to the most recent block and render the whole ledger invalid/corrupted. The most basic building blocks of a blockchain are the underlying peer to peer network where the participants try to reach agreement on which data is recorded to the ledger. A certain chunk of data (e.g. 1 MB for bitcoin) is assembled into a block and hashed (for performance reasons) into a unique string. This hash also includes the hash of the previous chunk of data. In that way the hash of block n is always linked to the hash of block $n-1$, creating a chain of data blocks. The underlying data structure is quite similar in all of the various blockchain systems out there whereas there exist majorly different approaches when it comes to reaching Consensus within a network. An overall distinction can be made between public and private blockchains. The basic idea of a public network is that everyone with access to the internet can participate wherein a private network only permissioned entities are allowed to join. In the course of this text I will walk you through the implementation of my custom private blockchain.

1.1 Origins

The term Blockchain emerged with the infamous Satoshi Nakamoto in 2008. Nakamoto used it as the underlying technology for implementing the electronic cash system Bitcoin [Nak08]. The paper is considered as the starting signal for the (mainstream) crypto boom. But the basic idea of a distributed P2P ledger/database blooms (at least) since the 1970s. You can read about immutable chaining of blocks in Ralph Merkle's dissertation from 1979 where he describes the data structure now known as Merkle hash tree - or David Chaum's Vault System (1979) which represents a blockchain prototype. These ideas were enhanced by a company named Surety in 1994 when adding time-stamps constructing data. Although these systems partly contained some kind of proof mechanism, Bitcoin was the first to include proof of work (PoW) in both the construction (Mining) of data/blocks and the Consensus process [ea]. Since then blockchain technology has come a long way.

1.2 Consensus

Another differentiation exists regarding accessibility/openness of a blockchain network. With a public (also called permissionless) system anyone can join the network anonymously - that is, for instance, in the case with Bitcoin and (classic) Ethereum. Public access often comes with proof of work Consensus. Here the integrity of the data and security is established by a mathematical puzzle participating miners try to solve. Whoever wins that race has proven validity of the new block/data by contributing appropriate processing power to the network. The PoW mechanism comes with two major parameters. First, the mining rate (or block frequency), which is the average duration it takes to create a new block. Second, the block size, which is the hard-coded disk space a block is allowed to allocate - e.g., 1 MB for Bitcoin blocks. In contrast to the permissionless access of public blockchains there are private (permissioned) systems - e.g., *MultiChain*. Hereby a network owner exists who represents the authority and has control over the network. She also decides who is allowed to participate. [Vuk16]

2 Related Work / State-of-the-Art

Meanwhile one often reads about a generational differentiation regarding blockchains. Bitcoin, for example, is seen as generation 1 providing classic (crypto)currency transactions. With the rise of Ethereum in 2015 the second generation of blockchains was born bringing an additional feature of so called smart contracts. This “self-executing protocol” opened a whole new world of possibilities for blockchain in contrast to the limited use cases bitcoin had to offer - nowadays bitcoin provides a scripting language although it is not Turing-complete unlike Ethereum’s Solidity. Sending transactions to a smart contract triggers certain events based on the inherent code of the contract. The Ethereum Blockchain provides the foundation for nearly all popular Decentralised Application (DApp) currently developed. At last Blockchain 3.0 applications are considered within advanced industries like governments, health care, science, IoT, video games and supply chain [FC19].

2.1 Bitcoin (public)

Considered as the first mainstream application of the blockchain is the bitcoin blockchain. Its native cryptocurrency *BTC* is still by far the most valuable coin in the *cryptoverse*. It is/was intended to act as an electronic cash system without the need for any intermediary like e.g. a bank. It implements a public proof of work consensus mechanism where everyone can join and participate in the regulation of the system. The integrity of the data and the ledger’s true state are established through mathematical calculations (CPUs) and not an third party intermediary.

The PoW consensus mechanism is often referred to as Mining where miners try to solve a cryptographic puzzle by guessing the solution. The more CPU power a miner has available the faster she can generate valid guesses and increase her chance of winning the *mining race*. The guessing process involves finding a hash with a certain number of leading zeroes. The system controls the difficulty of that task by introducing some parameters (nonce) to the input of that hash. Currently it takes around 10 minutes to mine a new block and ultimately confirm the transactions in it. The mining work also yields a reward meaning that new coins are minted. This introduces the concept where miners - i.e. the network maintainers - are economically incentivised to act according to the rules. Mining therefore represents the asset supply as well as the decentralised consensus mechanism. Because every node is checking new transactions based on universally encoded norms the decentralised network completely erases the need for a central authority.

Public Key Infrastructure (PKI) plays a big role in the bitcoin system. The public-private key pair enables participants to prove ownership of BTC by signing transactions with their private key. I will come back to the role of PKI in the implementation section. The structure of how funds are associated with their owners is not like the way we know

from traditional banking accounts but rather a so-called unspent transaction output model. The sum of all available inputs are equal to one's balance. One can think of that sum as the pool of available/spendable/unspent assets. A particular pool/chunk of unspent transactions can only be accessed or spent with a specific private key. The output in turn represents the transfer of ownership from one private key to another - the sender's output is the receiver's input. So as mentioned before there are no accounts per se but rather a chain of transactions dividing chunks of assets and transferring ownership between keys. Note that from a technical point of view a bitcoin wallet does not contain assets but only the keys to access them on-chain. For user convenience reasons the wallet constructs these higher level concepts of account addresses and balances though. (see [Ant18] ch02).

In 2016 the original bitcoin relay network transitioned to the UDP-based Fast Internet Bitcoin Relay Engine (FIBRE) where it introduced *DNS seeds* providing a list of other nodes' IP addresses to handle the exponentially growing number of participants and network load. After installing the bitcoin core reference (including the genesis block) on a local device the node only needs to connect to one peer to start the bootstrap the synchronization process (see [Ant18] ch08). [Nak08]

2.2 Hyperledger Fabric (private)

To demonstrate an opposite approach to blockchain technology we now take a look at a popular private (permissioned) platform and see what barriers it overcomes that public (permissionless) platforms limit. The open source blockchain project Hyperledger was created and is governed by Linux Foundation. An array of companies are working on different solutions - e.g. IBM on Fabric. Hyperledger Fabric is a private - also called permissioned - framework where each participant has to be accepted to join the network. Such a private environment is suited for corporate identities working together fulfilling a mutual demand. It therefore does not need the computationally heavy, power-draining process of mining to reach consensus within the network, instead using a traditional Byzantine Fault Tolerance (BFT) consensus protocol which is based on state-machine replication (SMR). Although there are authority nodes within the network this mechanism prevents single entities from taking over control and keeps the *trustless* dynamic intact. Deploying authoritative nodes in a blockchain environment is often proof of authority (PoA) where these authority nodes are called *validators*. Within the Fabric framework nodes take specific roles like admin, orderer, endorser and peer. Consensus is partially decentralised in the way that multiple nodes like endorsers and orderers take part in the process.

Fabric's basic model consists of the following components;

- assets (can be a digital currency or any other *tokenised* asset),
- *chaincode*/smart contracts (invoke transactions and interact with the blockchain),
- access control layer (CA, MSP) and

- consensus (endorsement policies enforced by selected peers)

A main feature of the network is to construct *channels*. This enables to set up a blockchain between selected members privately.

The above mentioned access permission is implemented using Certification Authority (CA) issuing X.509 certificates and trusted MemberShip Service Provider (MSP) to accept certified digital identities (encapsulated in an X.509 certificate). More details on these components will follow in the implementation section below.

The ledger consists of two parts, the *blockchain* and the *world state*. The blockchain is the immutable log of transactions whereas the world state is derived from the chain and quasi represents the status quo. The world state only changes if an update proposal completes the consensus process (deterministic). The abstraction from of the world state from the chain itself provides efficient storage and retrieval of often queried chain information.

One major innovation that came with Fabric was the *order-execute-validate* architecture. Let's first have a look at the *order-execute* architecture most of the other blockchains are/were using; each peer (within a proof of work (PoW) environment) creates a block containing valid transactions, executing them and then tries to solve the Mining riddle. If the node achieves that it then atomically broadcasts the block to its peers which in turn repeat the process the sender did before broadcasting the block. This is a typical implementation of active state-machine replication (SMR). That approach comes with some significant drawbacks; sequential execution of the transaction on all peers for example threshees the effective throughput. The gas concept in Ethereum (see Ethereum section) tackles this issue but it is not applicable to general-purpose private blockchains without a native coin. So after first executing transactions and then independently ordering them Fabric adds a third phase - *validate* - where the endorsements of the second phase (read-write sets) are again validated to guarantee a persistent state.

An active SMR environment or *order-execute* architecture is *non-deterministic* by nature; transactions are executed *after* reaching consensus arising the possibility to produce chain Forks. If a Fork occurs not all nodes are holding the same state anymore (see further discussion for more on Forks). This issue is addressed by domain-specific programming languages but most of these languages again inherit non-deterministic concepts anyway; e.g. a map iterator in *Go*.

In contrast to most public blockchains' active SMR Fabric implements *passive* replication. Not all peers are meant to or are required to access the smart contract logic; private blockchain data often requires confidentiality and privacy - that's one reason why they even exist. Fabric was also the first blockchain that introduced plug-able consensus. Under the conclusion that there is no "one-size-fits-all" consensus protocol the platform is capable of customising trust model mechanisms. This is possible because the ordering service/nodes are unaware of the application state and do not take part in the execution or the validation of transactions - orderers do not verify semantically. [fab]

2.3 Ethereum (public)

Considered as the second generation of blockchain technology Ethereum introduced smart contracts to its platform. Stemming from Nick Szabo's invention [Sza96] Vitalik Buterin worked on an overlay protocol (Mastercoin) for Bitcoin to implement smart contract functionality. Buterin's proposal was not implemented and so he started a separate project with the help of Dr. Gavin Wood. Ethereum is currently the dominant and most valuable *Altcoin* by market capitalization.

Because the majority of Decentralised Application (DApp) are built on top of the Ethereum Blockchain it is not only a blockchain but an operating system and therefore calls itself the "world computer". A DApp is basically a smart contract running on the (Ethereum) blockchain. For making a transaction or calling a smart contract on the Ethereum blockchain a fee (Gas) has to be paid. The gas price is determined by the miners and adds a security layer to e.g. avoid (D)DoS attacks.

In contrast to bitcoin's unspent transaction output model Ethereum uses an account-based model where two different types exist. On the one hand you have a wallet containing a key pair which is associated with an ordinary user; called externally owned accounts (EOA). On the other hand you have contract accounts which include code for sending transactions and writing into storage. An Ethereum smart contract is written in it's own scripting language Solidity and is triggered when certain events take place in the network.

The process of creating digital identities is decentralised too. Zero-trust data stores (like the Ethereum Claims Registry) make it possible to verify claims without a trusted entity. As mentioned earlier the Turing-completeness of the Ethereum Virtual Machine (EVM) is thanks to the gas fee because each individual instruction of a smart contract is associated with gas. If a function runs out of gas the EVM will terminate and avoid the smart contract from indefinitely consuming resources [12, 13].

2.3.1 Ethereum 2.0

The Ethereum ecosystem is currently upgrading to a new version *Eth2* to make the platform more scalable (faster transactions), secure (against attacks) and sustainable (proof of stake instead of proof of work). Highly fluctuating transaction fees, PoW's environmental impact through enormous energy consumption and increasing disk space required to run a full node makes it unsuitable for mass adoption. Eth2 aims to drastically improve these aspects without sacrificing decentralisation. PoS is *the* major change wherein miners are replaced by validators. Theoretically, staking increases the number of nodes - hence decentralisation - because it eliminates the barrier of expensive hardware mining requires. Validators are incentivised to act benign because they are liable with their stakes. A validator will loose (a portion of) her stake for (deliberately) failing. [eth]

2.4 Others

Cardano (ADA)

Cardano is another popular use case of blockchain technology. It is developed by *IOHK* and boasts itself by following a scientific approach in developing its project and it aims to be a "third generation" blockchain; some see it as a major competitor to Ethereum. Cardano went live in 2015 but it has not implemented all of its features yet hence it has not realized its full potential. For example, only the next phase (Goguen) will introduce smart contract functionality to the platform. After that there will be two phases left which will implement the optimisation of scalability/interoperability and a treasury system regarding governance. Cardano's own proof of stake consensus *Ouroboros* is designed to handle up to 24 000 transactions per second hence tackling the ongoing throughput issue of proof of work systems. [Aca]

Some significant (private) blockchain platforms include IBM Blockchain Platform, Oracle Blockchain Platform, Alibaba Cloud BaaS, Ripple and MultiChain.

Ripple (XRP)

The Ripple network, for example, is controlled by a single company (Ripple). It does not follow bitcoin's core values and instead tries to enhance the current global payment infrastructure. In fact, Ripple is not a blockchain per se, consensus is reached by leveraging validation servers. Because Ripple tries to be the global settlement network where its currency XRP represents the value of exchange its customers are financial institutions; whereas other cryptocurrencies represent an alternative to traditional financial services [10].

Regardless of the technological evolution of the last years all blockchains are based on the same logical building blocks, namely a data model for the blocks, hashing algorithms to generate block and transaction hashes, a consensus mechanism and a P2P protocol.

3 Design and Implementation

This section dives into the technical level of the project and describes the algorithms in use and implementation details. Here are some details on the Technology Stack,

- Programming Language: Java 15
- IDE: IntelliJ IDEA 2020 (CE)
- UI: Command Line Interface
- Logging framework: SLF4J
- Unit testing: JUnit 5
- Repository management: GitLab
- Source code documentation: JavaDoc
- Build management tool: Maven

Modules like the access permission layer and the ledger were mostly inspired by the Hyperledger Fabric principles. Data encryption and integrity is implemented with Public Key Infrastructure (PKI); it is the fundamental piece to guarantee authenticity of transferred data.

3.1 Access Permission

As previously discussed, this blockchain runs in a private environment. Therefore the program starts with running the user through an access permission layer. This includes a Certification Authority (CA) and a MemberShip Service Provider (MSP). The CA issues certificates which determine the exact permissions over resources. The MSP validates the issued certificate. One can think of the CA as a credit card company and the MSP as a store accepting selected cards. In the context of the private blockchain the CA issues a public-private key pair where the public key represents the digital identity of its holder. The MSP then confirms that a specific public key belongs to a specific entity/organisation/user/device/etc. A registration authority (RA) is often added to CA tasks to identify and authenticate applicants and applications. This also includes possible revocations or suspensions of certificates/keys.

The classes *CertificateAuthority* and *MembershipServiceProvider* simulate entire process. The CA issues a key pair (see Listing 3.1) whereas the MSP is verifying the public key

3 Design and Implementation

- the private key must not be shared - and listing all permissioned/accepted identities (public keys). Note that most of these verification methods are of external nature hence done by humans or independent software.

The *java.security* package is used for the key pair generation. The *CertificateAuthority* issues a *KeyPair* object using a *KeyPairGenerator* - its *getInstance* method takes in an algorithm and provider label. In my case that's the Digital Signature Algorithm (DSA) from Sun Microsystems. The *SHA1PRNG* algorithm is used for random number generation and the key size is set to 1024 bits.

```
1 public KeyPair issueCertificate()
2 {
3     // external validation methods IRL
4     try
5     {
6         KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "
SUN");
7         SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "
SUN");
8         keyGen.initialize(1024, random);
9         logger.info("Issuing new digital ID (KeyPair).");
10        return keyGen.generateKeyPair();
11    } catch (Exception e)
12    {
13        logger.error("Issuance failed, Key(Pair) could not be generated."
, e);
14        e.printStackTrace();
15    }
16    return null;
17 }
18 }
```

Listing 3.1: generating/issuing a digital identity (key pair)

After being permissioned a Node instance is created providing the selected port number, username and the issued public key. The key pair (digital ID) is passed into the *Controller.launchApp* method where it is passed onto the *Wallet* class which is responsible for storing the key pair.

3.2 the App

The *Controller* is the central instance handling the business logic. It holds *Wallet*, *Blockchain* and *WorldState* instances - also calling the *UI.display()* when needed. The wallet holds the previously generated key pair hence the user's private and public key. The wallet address is generated by sending the public key through the *Sha256Hasher.hash* method which takes in a string of data and returns a 256-bit hash of that data. This hasher is also used within block headers and merkle trees.

Another essential feature the wallet provides is signing transaction proposals and block headers. This enables the receiving side of a packet to verify that an associated packet was approved (signed) by the sender in its current state. Consequently the receiver can

be sure that the packet and its data is not corrupted. The process of e.g. signing a transaction proposal (TxProposal) is shown in Listing 3.2. The method takes an to be signed object as input parameter, creates a `java.security.Signature` instance, provides that signature with a private key and updates/adds the serialized transaction proposal to the signature instance before signing and returning the serialized representation of the complete signature. Note, because the private key must not be shared, only the wallet classes which have access to the private key can implement signing operations.

```

1 public byte[] sign(TxProposal txProp)
2 {
3     try
4     {
5         // specifies sig algo (DSA) using the digest algo (SHA-1)
6         Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
7         dsa.initSign(keyPair.getPrivate());
8         dsa.update(txProp.stringify().getBytes()); // supplies tx data to
           the Sig object
9         return dsa.sign();
10
11     } catch (Exception e)
12     {
13         logger.warning("Tx proposal could not be signed.");
14         e.printStackTrace();
15     }
16     return null;
17 }

```

Listing 3.2: signing a tx proposal | generating digital signature

The initially created Node is assigned at least one *Role* that determine its rights within the network. Every new node is assigned the role of a peer, meaning she is participating in the network and is passed the access permission. An admin for example is allowed to change the roles of participants i.e. up- or downgrading someone with respect to their rights. An orderer is responsible for ordering (mining) new blocks. An endorser receives transaction proposals and approves/rejects them. After instantiation the node is then used to set up the peer to peer network infrastructure.

3.3 the Network

In the context of this implementation I will refer to a running instance of the program with “node” or “user”.

The bootstrap process starts with discovering peers by setting up the necessary objects - creating a `java.net.MulticastSocket` object (class `MulticastReceiver`) to listen for “broadcast” packets. To guarantee the integrity of the setup the (multicast) port number and multicast host address are defined in a separate file (class `network.Config`).

```

1 @Override
2 public void run()
3 {
4     logger.info("Thread (for receiving multicasts) started.");

```

3 Design and Implementation

```
5
6  try
7  {
8      socket = new MulticastSocket(Config.multicastPort);
9      InetAddress group = InetAddress.getByName(Config.multicastHost);
10     socket.joinGroup(group);
11
12     while (true)
13     {
14         byte[] buffer = new byte[9800];
15         DatagramPacket incoming = new DatagramPacket(buffer, buffer.
length);
16
17         socket.receive(incoming);
18         String received = new String(incoming.getData(), 0, incoming.
getLength());
19         if ("end".equals(received))
20         {
21             break;
22         }
23         Message message = PacketHandler.parseMessage(incoming);
24         switch (message.getMsgType())
25         {
26             case PeerDiscovery:
27                 logger.info("incoming peer discovery from {}. ",
message.getSender());
28                 if (!NetworkInfo.peers.contains(message.getSender()))
29                 {
30                     PortSender.respondHandshake(message.getSender().
getPort(), node);
31                     NetworkInfo.peers.add(message.getSender());
32                     // adds address to chain accounts
33                     Controller.worldState.getBalances().put(message.
getSender().getAddress(), 100.); // adds initial starting balance
for demo purposes
34                 }
35                 break;
36
37             case NewBlock:
38                 Block rcvBlock = PacketHandler.parseBlock(message);
39                 logger.info("receiving new Block # {}. ", rcvBlock.
getHeader().getHash());
40
41                 if (SmartContract.verifyNewBlockSig(rcvBlock))
42                 {
43                     Controller.blockchain.add(rcvBlock);
44                     Controller.worldState.update(rcvBlock.getTxn());
45                     Controller.worldState.getMempool().clear();
46                 } else
47                 {
48                     logger.error("New Block invalid.");
49                 }
50                 break;
51
```



```

52         case Disconnect:
53             NetworkInfo.peers.remove( message.getSender() );
54             break;
55     }
56 }
57
58     socket.leaveGroup(group);
59     socket.close();
60 } catch (IOException e)
61 {
62     e.printStackTrace();
63 }
64 }

```

Listing 3.3: listening for incoming multicast packets

Each peer is associated with an individual port number. During the bootstrap it also creates a `java.net.DatagramSocket` (class `PortReceiver`) on its associated port number and starts listening for packets/messages destined for that particular port. As the name suggests it will be UDP packets which are sent over a *DatagramSocket*. Technically, the *PortReceiver* works like the *MulticastReceiver* with the only exception that it listens on a specific port.

Both the `MulticastReceiver` and `PortReceiver` instances are running an infinite loop listening for packets. Both these processes also run in a daemon thread to enable the program to process incoming messages without stopping the user from interacting with the application.

After setting up `Multicast-` and `PortReceiver` the node is requesting a handshake to discover peers. Notice, in Listing 3.4 the transmitted data is packaged into a custom `Message` object containing a custom `MessageType` (enum) so that the receiving side can more easily identify and process the data based on that.

```

1 public static void requestHandshake(Node requester)
2 {
3     try
4     {
5         socket = new DatagramSocket();
6         group = InetAddress.getByName(Config.multicastHost);
7         Message msg = new Message(MessageType.PeerDiscovery, requester
8         , null);
9         byte[] buffer = msg.serialize();
10        DatagramPacket packet = new DatagramPacket(buffer, buffer.length,
11        group, Config.multicastPort);
12        logger.info("handshake request / discovering peers...");
13        socket.send(packet);
14    } catch (IOException e)
15    {
16        e.printStackTrace();
17    } finally
18    {
19        socket.close();
20    }
21 }

```

19 }

Listing 3.4: multicasting a handshake/discovery packet

In case of receiving a handshake (peer discovery) message, the receiver responds (PortSender.respondHandshake) with its own information and adds the sender's address to its local accounts. For demonstration purposes every gets an initial starting balance - see Listing 3.5. It also shows a custom PacketHandler extracting/parsing objects from the received packet i.e. deserializing the byte array of information.

```

1 ...
2     switch (message.getMsgType())
3     {
4         case PeerDiscovery:
5             logger.info("incoming peer discovery from {}.", message.
6               getSender());
7             if (!NetworkInfo.peers.contains(message.getSender()))
8             {
9                 PortSender.respondHandshake(message.getSender().getPort(),
10                node);
11                NetworkInfo.peers.add(message.getSender());
12                // adds address to chain accounts
13                Controller.worldState.getBalances().put(message.getSender
14                ().getAddress(), 100.); // adds initial starting balance for demo
15                purposes
16            }
17            break;
18
19         case NewBlock:
20             ...
21     }
22 }

```

Listing 3.5: processing a handshake/discovery packet

After completing the discovery interaction the node has a set of peers stored in the class network.PeerInfo. Meanwhile the program has instantiated a Blockchain instance containing only the universal genesis block. The freshly created blockchain has to be synchronized with the help of the discovered peers to catch up to the most recent block and become a “fully participating network node”.

3.4 the Ledger

The centerpiece of the application is the blockchain itself of course. Every building block is found in the ledger package. A Blockchain object contains an *ArrayList<Block>* representing the chain of intertwined blocks. The creation of block number 0 - the so-called genesis block - is embedded into the constructor. It uses some hard coded parameter values for e.g. the timestamp to guarantee a consistent block hash across all nodes regardless of the time they are joining.

A block is composed of a *BlockHeader*, a *List<Transaction>* and signature-key pair (SigKey) object. The SigKey class is a utility object holding a signature in byte array

form and the associated public key to simplify the verification processes within the PKI. The sigKey within a block refers to the orderer (miner) of the block. A Transaction contains

- a transaction proposal (TxProposal) defining the sender's address, receiver's address and an amount to be transferred,
- a sig-pubKey pair, referring to the sender's signature,
- a set of sig-pubKey pairs, representing the endorsements of the transaction.

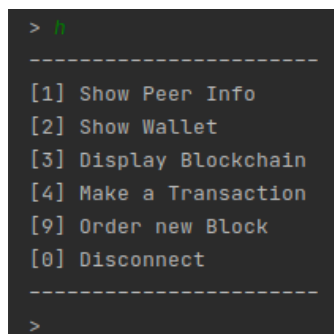
The third field - the block's header - contains the following fields:

- the height or the index number (in the context of a private/deterministic blockchain),
- the merkle root, representing the root of the transactions' merkle tree,
- the preHash, referring to the previous block's hash (making the blocks intertwined),
- a timestamp (time of the block creation) and
- the block's hash.

A block hash is generated by hashing its height, merkle root, preHash and timestamp through Sha256 hasher.

Parallel to the blockchain the Controller also holds a WorldState object storing the accounts and assets of the participating nodes. Because the world state associates participants with accounts my blockchain is leveraging an account-based rather than a UTXO model. Orderer nodes also use the world state to store pending transactions which are not yet ordered into a block (mempool).

After the peer discovery and chain sync is finished the is fully integrated into the network can use its features like accessing blockchain information and making transactions.



```

> #
-----
[1] Show Peer Info
[2] Show Wallet
[3] Display Blockchain
[4] Make a Transaction
[9] Order new Block
[0] Disconnect
-----
>

```

Figure 3.1: UI

The following steps will show a transaction sequence and how the network finds consensus i.e. agreeing on the validity of a transaction and further ordering transactions

3 Design and Implementation

```
> 4
enter receiver's address > 7f18a10a07cc3c71a1a0f2a27f547c0400a4e50fcb0200d4e730a07c0a0220
enter amount > 10
2021-01-04 16:40:14 [Thread-1] INFO PortReceiver:120 - incoming TX Endorsement Message{msgType=TxEndorsement,
```

Figure 3.2: proposing a transaction

into a new block. When starting a transaction the user is confronted by input fields asking her to enter the receiver's address and the amount to be transferred.

The requested transaction data is packed into a TxProposal and temporarily stored in a map of active/pending tx proposals (network.PeerInfo.activeProposals). The proposing node (sender) signs the tx proposal with his private key which is stored within the wallet instance and sends the packet.

```
1 ...
2     case "4":
3         TxProposal txProposal = TxProposal.createTxPropUI(node, wallet);
4         NetworkInfo.activeProposals.put(txProposal, new HashSet<>());
5         // sign tx proposal
6         byte[] sig = wallet.sign(txProposal);
7         PortSender.proposeTx(node, txProposal, sig, wallet.getPub());
8         Thread.sleep(1100); // ...before checking if proposal was
9         endorsed
10        if (NetworkInfo.activeProposals.get(txProposal).size() >= Config.
11        endorsers.size()/2) // 50% endorsement policy
12        {
13            logger.debug("EndorsementStatus " + NetworkInfo.
14            activeProposals);
15            Transaction tx = new Transaction(txProposal, new SigKey(sig,
16            wallet.getPub()));
17            tx.setEndorsements(NetworkInfo.activeProposals.get(txProposal
18            ));
19            PortSender.submitTxProposal(node, tx);
20        } else
21        {
22            logger.info("missing endorsements, forfeiting proposal.");
23        }
24        NetworkInfo.activeProposals.remove(txProposal);
25        break;
26 ...
```

Listing 3.6: receiving/processing a tx proposal (Controller)

Now the actual consensus part begins - the packet is sent to the endorser peers. Again the to be transmitted data is packed into a Message object before serialized and sent from socket to socket as a byte array.

```
1 public static void proposeTx(Node node, TxProposal txProposal, byte[]
2     sign, PublicKey pub) throws IOException
3 {
4     for (int port : Config.endorsers)
5     {
```

```

5      Message<TxProposal> message = new Message<>(MessageType.
      TxProposal, node, txProposal, new SigKey(sign, pub));
6      byte[]          buffer = message.serialize();
7      DatagramPacket packet = new DatagramPacket(buffer, buffer.length,
      InetAddress.getByName("localhost"), port);
8      new DatagramSocket().send(packet);
9  }
10 }

```

Listing 3.7: transmitting a tx proposal packet

An endorser on the receiving side parses the packet and extracts the transaction proposal and the corresponding signature. The endorser calls the integrated smart contract functionality - `verifyTxProposal()` - to enforce the endorsement policy implemented into the network. In my blockchain the `SmartContract` is used to verify transmitted data and blockchain information. It basically exists to define the rules between the participants “carved” in executable code. You can find more information on smart contracts in other sections (about e.g. Ethereum). However this particular smart contract checks if the signature is valid and the sender has enough funds on her account balance to make the transaction possible. If everything is verified by the contract instance the endorser sends an endorsement back to the proposing node in the form of her own digital signature of that proposal. If the contract identifies the proposal as invalid the endorser will not send a response to the proposer of course.

```

1 ...
2     case TxProposal:
3         logger.info("incoming TX Proposal " + message.getPayload());
4         TxProposal txProp = PacketHandler.parseTxProp(message);
5         if (SmartContract.verifyTxProposal(txProp, message.getSigKey()))
6         {
7             logger.info("endorsing proposal; added to mempool");
8             Controller.worldState.getMempool().add(new Transaction(txProp
, message.getSigKey()));
9             SigKey sigKey = new SigKey(Controller.wallet.sign(txProp),
Controller.wallet.getPub());
10            PortSender.endorseTx(message.getSender().getPort(), node,
txProp, sigKey);
11        }
12        break;
13 ...

```

Listing 3.8: receiving/processing a tx proposal packet

In the case of receiving an endorsement the proposer adds the received signature to the set of endorsements of her proposed transaction and also updates the set active proposals.

```

1 ...
2     case TxEndorsement:
3         logger.info("incoming TX Endorsement " + message);
4         TxProposal endorsedProposal = PacketHandler.parseTxProp(message);
5
6         // update endorsement (add sigKey of endorser)
7         Set<SigKey> endorsements = NetworkInfo.activeProposals.get(
endorsedProposal);

```

3 Design and Implementation

```
8      endorsements.add(message.getSigKey());
9      NetworkInfo.activeProposals.put(endorsedProposal, endorsements);
10     break;
11 ...
```

Listing 3.9: receiving/processing a tx endorsement packet

Because every transaction proposal is verified against the current world state and the pending transactions in the mempool the problem of double-spending can be prevented. If the majority of the network will act maliciously (“51 % attack”) the blockchain can get corrupted nodes by reversing transactions and spending them twice.

The proposing node is waiting for a little bit before checking if she received enough endorsements in order to proceed to the next phase. If there are not enough endorsements the node has to drop its proposal and remove it from the set of active proposals. In case there came enough endorsing responses the proposer can proceed to the next step. She packs the proposal and the set of endorsements into a Transaction and sends it to one of the ordering nodes. The transaction is sent to a random ordering node in order to decentralise and distribute the workload.

We have reached the last phase of the peer to peer consensus procedure. All the ordering node is doing in our proof of authority (PoA) context is to verify the incoming transaction regarding all its signatures with the help of the integrated validation/smart contract before storing the incoming transaction in the mempool. The purpose of the mempool is to temporary store accepted but not yet ordered transactions. At this point in time the orderer node can assemble the mempool into a new block using the smart contract to access blockchain information.

```
1 ...
2     case TxSubmission:
3         Transaction submittedTx = PacketHandler.parseTx(message);
4         logger.info("incoming TX Submission " + submittedTx);
5         //
6         if (SmartContract.verifyTxSubmission(submittedTx))
7         {
8             Controller.worldState.getMempool().add(submittedTx);
9             logger.debug("{} is valid, added to mempool.", submittedTx);
10        } else
11        {
12            logger.error("{} rejected.", submittedTx);
13        }
14        break;
15 ...
```

Listing 3.10: receiving/processing an endorsed tx proposal

When attempting to order a new block the application checks if the corresponding node is entitled to do so.

```
1 case "9":
2     if (!node.getRights().contains(Role.ORDERER))
3     {
4         System.out.println("You are not authorised to order blocks.");
5         break;
```

```

6     }
7     if (Controller.worldState.getMempool().isEmpty())
8     {
9         System.out.println("There are no pending transactions.");
10        break;
11    }
12    Block b = SmartContract.orderNewBlock(node);
13    MulticastSender.sendNewBlock(node, b);
14    break;

```

Listing 3.11: attempting to order a new block

Assembling a new block includes some self-explanatory parameters as well as a Merkle root hash of all its transactions. To generate that hash a binary Merkle tree is created.

```

1 public static Block orderNewBlock(Node orderer)
2 {
3     if (!orderer.getRights().contains(Role.ORDERER))
4     {
5         System.out.println("You are not authorised to order blocks.");
6         return null;
7     }
8     MerkleTree merkleTree = new MerkleTree(Controller.worldState.
9 getMempool());
10    String root = merkleTree.getMerkleRoot().get(0);
11    BlockHeader blockHeader = new BlockHeader(
12        Controller.blockchain.getLatestBlock().getHeader().getHeight
13        () + 1,
14        root,
15        Controller.blockchain.getLatestBlock().getHeader().getHash(),
16        System.currentTimeMillis()
17    );
18    SigKey sigKey = new SigKey(Controller.wallet.sign(
19        blockHeader), Controller.wallet.getPub());
20    return new Block(blockHeader, Controller.worldState.getMempool(),
21        sigKey);
22 }

```

Listing 3.12: assembling a new block (Smart Contract)

It takes a list of individual transaction hashes and recursively (see return statement below) runs them through the construct method. Let's examine that function in more detail. if there is only one transaction, respectively hash then that is already the root hash obviously (line 40). Otherwise it starts merging neighbouring hashes (line 45-48); note that if there is an odd number of transaction hashes the last tx hash is merged with itself (line 51-54). In any way the last recursion will see a list with two remaining hashes, merging them and finally returning a list containing only one hash - the (Merkle) root.

```

1 /**
2  * Recursively merges neighbouring hashes from transactions list.
3  *
4  * @param transactions list of all transaction hashes
5  * @return list of merged neighbour hashes => last recursion returns
6  *         single element, the root

```

3 Design and Implementation

```
6  */
7  private List<String> construct(List<String> transactions)
8  {
9      if (transactions.size() == 1) return transactions; // merkle root
10     found
11     List<String> updatedList = new ArrayList<>(); // contains half as
12     much elements after each recursion
13     // merges neighbouring items
14     for (int i=0; i < transactions.size()-1; i+=2)
15     {
16         updatedList.add( mergeHash(transactions.get(i), transactions.get(
17         i+1)) );
18     }
19     // if odd # transaction, last item is hashed with itself
20     if( transactions.size() % 2 == 1 )
21     {
22         updatedList.add( mergeHash(transactions.get(transactions.size()
23         -1), transactions.get(transactions.size()-1)) );
24     }
25     return construct(updatedList); // recursion
26 }
```

Listing 3.13: generating a Merkle root

That unique root hash is now used as an essential part for data integrity. As can be derived from the algorithm above any minor change in the underlying transaction data would lead to different leaf and/or branch hashes and will therefore be propagated through the root. Once there is a generated root hash it is the only thing needed to verify that the underlying data is not corrupted or there were no changes made to it afterwards.

After fetching all the necessary parameters - and hashing them to generate the new block's hash - the block creation is complete; all the transactions are “ordered” into a block. It is then broadcasted to all the other nodes which will do a swift verification of the received block and append it to their local chain. The world state i.e. account balances are updated according to the blocks transactions. This completes the procedure from creating proposing a new transaction to including it into a new block.

There is one scenario left. Rewind to the point after a node discovers its peers when trying to sync her “empty” (only containing the genesis block) blockchain. In a realistic scenario most of the peers are joining the network from the very beginning but rather when there are already some blocks in existence. Even when a node goes offline for some time it has to catch up when she is rejoining the network. We will now go through the process of requesting missing blocks in a secure and scalable way. The requester sends the latest block's header of her local chain to a random peer. This enables the joining node to synchronize even if she is only connected to the one peer. The decentralised (p2p) nature of blockchains comes with the feature that all/some participants have the full ledger.

```
1  public static void requestChainSync(Node requester, BlockHeader
2  blockHeader) throws IOException
3  {
4      Message<BlockHeader> msg = new Message<>(MessageType.
5      ChainSyncRequest, requester, blockHeader);
```



```

4      byte[]          buffer = msg.serialize();
5
6      int toPort = PeerInfo.fetchRandPeer().getPort();
7      DatagramPacket packet = new DatagramPacket(buffer, buffer.length,
8      InetAddress.getByName("localhost"), toPort);
9      new DatagramSocket().send(packet);
10 }

```

Listing 3.14: casting a chain sync request

The receiver of that chain sync request parses and checks the received header. Because in this implementation multi-/broadcast messages are also sent to oneself the receiver initially checks if the request is not coming from oneself. Consequently it fetches the corresponding block based on the received block hash (line 68) to guarantee the integrity of the requester's chain - see Merkle tree section for more details. If there is a block corresponding to the received hash the receiver then goes on to determine the requesting peer's chain height and if she is missing any blocks. In case the requester does not miss any blocks a response is sent with a null payload indicating that the requester is not missing any blocks and is up to date. In case the requester is missing some blocks (line 81-90) the receiver fetches the next block in line and responds with it.

```

1 case ChainSyncRequest:
2     // do not process request from oneself
3     if (!message.getSender().equals(node))
4     {
5         BlockHeader rcvBlockHeader = PacketHandler.parseBlockHeader(
6         message);
7         logger.info("incoming chain sync request, {}.", rcvBlockHeader);
8
9         Block localBlock = Controller.blockchain.fetchBlock(
10        rcvBlockHeader.getHash());
11        if (localBlock == null)
12        {
13            logger.warn("requester sent invalid or corrupted block.");
14            break;
15        }
16
17        int chainHeight = Controller.blockchain.getChain().size();
18        int bestHeightPeer = rcvBlockHeader.getHeight() + 1;
19        int noMissingBlocks = chainHeight - bestHeightPeer;
20
21        if (noMissingBlocks == 0)
22        {
23            // send info (null block) that new peer's chain is up to date
24            logger.info("{}'s chain is up to date.", message.getSender());
25        }
26        ;
27        PortSender.respondChainSync(message.getSender().getPort(),
28        node, null);
29    } else
30    {
31        // respond with next block in line

```

3 Design and Implementation

```
28         logger.debug("{} is {} block(s) behind.", message.getSender()  
29         , noMissingBlocks);  
30         Block nextBlock = Controller.blockchain.getChain()  
31             .get(localBlock.getHeader().getHeight() + 1);  
32         PortSender.respondChainSync(message.getSender().getPort(),  
33         node, nextBlock);  
34     }  
35     break;
```

Listing 3.15: processing and responding to a chain sync request

At the point of receiving a response to a chain sync request the node checks if there is a payload (block) contained in the message. If not she knows that her chain is up to date - otherwise she parses the block, adds it to her local chain and updates the world state accordingly. The node then sends another chain sync request until she reaches blockchain's status quo.

```
1 case ChainSyncResponse:  
2     if (message.getPayload() == null)  
3     {  
4         logger.info("Sync response contains no block, local chain is up  
5         to date.");  
6     } else  
7     {  
8         Block nextBlock = PacketHandler.parseBlock(message);  
9         Controller.blockchain.add(nextBlock);  
10        Controller.worldState.update(nextBlock.getTx());  
11        PortSender.requestChainSync(node, Controller.blockchain.  
12        getLatestBlock().getHeader());  
13    }  
14    break;
```

Listing 3.16: processing a chain sync response

3.5 persistent, local storage

In the *util* package you can find a *Serializer* and a *Deserializer* class which is responsible for storing and retrieving information about the node, its key pair and the blockchain locally. It is achieved by implementing Java's *Serializable* interface and *java.io* Input- and OutputStream objects respectively. This functionality completes my custom interpretation of the basic building blocks of a private blockchain system.

4 Further Discussion

Let's now dig a little deeper into some approaches blockchains are implementing and also explore some scams and failures we have witnessed so far.

The proof of work (PoW) approach for example stipulates that there will be an finite supply (around 21 million) of coins and with its decreasing issuance rate it (theoretically) resists inflation; it cannot be printed excessively like we currently see with fiat money in times of economic crisis as governments try to maintain economic standards as good as possible - risking to (hyper)inflate the fiat currency. Among other properties that is why Bitcoin is seen as superior and more trustworthy than fiats.

Another big issue could arise faster than expected - resilience to quantum computing. A quantum computer using Grover's algorithm could crack Public Key Infrastructure (PKI).

A blockchain is not always the better alternative to a traditional database. One has to estimate the suitability based on the use case and sector. For example, a common database is preferred where low latency or high transaction rate is of major and decentralisation of rather low importance.

There are loads of modification solutions, like Sharding, trying to overcome the well-known drawbacks a lot of blockchains are facing. But note that these solutions influence the equilibrium between security, scalability and decentralisation. [FC19]

4.1 Scalability Trilemma

As we have examined the different approaches like proof of work and proof of authority one could already derive some compromises coming with different implementations. A highly decentralised system using proof of work is much harder to scale and usually harder to secure than a more centralised one leveraging PoA or PoS consensus. This challenge is often referred as the decentralisation *trilemma*. The term is coined by Ethereum's Co-Founder Vitalik Buterin which stated that a blockchain in its purest form can only fulfill two of the three criteria. Full decentralisation assumes that networks are permissionless and censorship-resistant where one could argue that these properties are not given within a private blockchain. Although the more entities taking part in reaching consensus the longer it takes process a transaction hence hindering scalability. Technical factors like the fixed block size in Bitcoin (1 MB) also hinders the system from scaling. Although there are (proposed) solutions like the *Lightning Network* to resolve transactions between parties off-chain; such solutions come with their own drawbacks and vulnerabilities.

In principal, PoW does a pretty good job in respect to decentralisation and security. Although one could already criticize that even these properties are in danger of falling

4 Further Discussion

apart. First of all, with growing popularity and adaption proof of work consumes enormous amounts of energy currently equalling the consumption of whole states. Secondly, because of the high difficulty setting in popular PoW blockchains nowadays (Bitcoin, Ethereum) there is no realistic chance of winning the mining race and being rewarded. One has to join a *mining pool* where hashing power is bundled and rewards are distributed among members. Generally, that is not a problem but we can observe that the growing power of these pools are leading to a centralisation of mining. As of mid 2018 around 84 % of all Ethereum blocks were mined between the five biggest mining pools; see [Ale18] further details. One has to consider the economic factor as well. Mining requires hardware and power which is - on average - more affordable in China than in Europe for example [pow]. Having a major chunk of the network maintainers within one national boarder makes it vulnerable/dependent on political and economic changes within that country.

One could argue that - because of the aforementioned problems - proof of work is not sustainable for mainstream adaption.

4.2 Forks

An issue with proof of work protocols is that Forks can occur. Let's further investigate that event with regards to some notable occurrences like in Ethereum in 2016 (Ethereum Classic). Bitcoin e.g. also witnessed a fork in 2017 leading to the emergence of *Bitcoin Cash*. With a fork the network transitions into different states. There are different type of forks. Firstly, there is a *transient fork* where two (or more) miners create a valid new block without knowing that someone else already created one too. This will be resolved by just continuing to mine blocks until one of the partitioned blockchains will become the longest one; and will be accepted as *the* right one. Secondly, there deliberate Forks to implement protocol updates called *hard* and *soft forks*. If an update/change is backward-compatible we are talking about a soft one; if that is not the case then there is a hard fork.

4.2.1 the DAO Fork

The infamous *ETC-ETH* fork unfolded in 2016 involving a collection of smart contracts called *DAO* (Distributed Autonomous Organization). DAO was a crowdfunding platform for Ethereum projects. Unfortunately, there was a vulnerability in one the DAO's contracts that was exploited to an estimated amount of about \$ 50 million worth of ether. The bug lied within a DAO contract, not Ethereum itself making the contract perfectly valid from the platform's point of view. Consequently, a hard fork was proposed to erase the attacker's transactions. This obviously arose a fundamental debate among developers regarding the core values of the platform. In conclusion, the Ethereum Blockchain partitioned into ETC which records the attacker's transactions and ETH which deleted the attacker's transactions.

In the short-term, ETC suffered from enormous exodus of nodes which abruptly plummeted the hash rate - hence nearly no blocks were mined. The PoW's difficulty

adjustment took nearly two days before continuing in normal procedure. The long-term statistics show that most of the miners transitioned to ETH where difficulty and transactions per day were substantially higher than ETC's. To evaluate the real reasons is a topic for another study because there are multiple factors like mining reward, media attention or competition to consider. [LK17]

5 Conclusion

A peer-to-peer is constructed using sockets from the *java.net* package. Secure communication over the P2P network is guaranteed through Public Key Infrastructure (PKI). Each packet is digitally signed with the sender's private key; the receiver can therefore examine if the data was manipulated in between.

Of course there are still limitations to this simplistic approach of creating and combining the basic building blocks of a blockchain. Multiple instances of the application can only run locally. At the current state of development there is no possibility to connect/synchronise from another device or network.

5.1 Future Work

Obviously, this custom implementation is nowhere near the level of a sophisticated state-of-the-art blockchain. Let's emphasize a few steps which would bring this blockchain to the next level.

Implementing a *thread pool* would enable the receiver to handle multiple requests at a time - making the whole application more scalable with growing number of participants. In that regard a thread pool brings significant performance gains because it reuses threads and they do not have to be created.

As indicated in the previous section the next meaningful step would be to implement remote communication - it should also be possible for devices/nodes to connect from a different network.

Generally, private blockchains are not vulnerable to 51 % attacks where ill-intent participants take control over the majority of the network and consequently are able to manipulate the ledger. My custom implementation would only suffer from such kind of attack if the majority of endorsers are conspiring. But because they are acting independently from each other this is highly unlikely to happen. Within the context of a public and proof of work blockchain the attackers would have to gain the majority of the computing power where the cost of such an attack grows with number of participants.

Testing the robustness of the application goes beyond the scope of this project hence is difficult to estimate. I assume that there is a possibility to successfully flood the endorsers with transaction proposals because at the moment there is no balancing or load distribution mechanism on the endorsement process.

Key management is another important security concern which has to be addressed. Currently, a user's key pair (and private key) is just stored locally within the wallet object. A first step would be to encrypt all of the locally stored data (at least the key pair).

Governance of the blockchain currently lies solely in the hand of the admin(s). It is also desired to decentralise that aspect with democratising the influence of decision making

5 Conclusion

although within the context of private blockchains it makes no sense for the admin/owner to share that right with others because of e.g. privacy reasons.

Bibliography

- [Aca] Binance Academy. What is cardano (ada)? <https://academy.binance.com/en/articles/what-is-cardano-ada>.
- [Ale18] Alethio. Are miners centralized. a look into mining pools. <https://media.consensys.net/are-miners-centralized-a-look-into-mining-pools-b594425411dc>, 2018.
- [Ant18] Andreas M. Antonopoulos. Mastering bitcoin. <https://github.com/bitcoinbook/bitcoinbook>, 2018.
- [ea] Alan Sherman et. al. On the origins and variations of blockchain technology. <https://www.chaum.com/publications/Origins%20of%20Blockchain%2008674176.pdf>.
- [eth] Ethereum 2.0. <https://ethereum.org/en/eth2/>.
- [fab] Hyperledger fabric docx: Key koncepts. https://hyperledger-fabric.readthedocs.io/en/release-2.2/key_concepts.html.
- [FC19] et al. Fran Casino. A systematic literature review of blockchain-based applications. <https://www.sciencedirect.com/science/article/pii/S0736585318306324#s0010>, 2019.
- [LK17] et. al. Lucianna Kiffer. Stick a fork in it: Analyzing the ethreum network partition. <https://dl.acm.org/doi/pdf/10.1145/3152434.3152449>, 2017.
- [Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. <https://bitcoin.org/bitcoin.pdf>, 2008.
- [pow] Average electricity procecs around the world: \$/kwh. <https://www.ovoenergy.com/guides/energy-guides/average-electricity-prices-kwh.html>.
- [Sza96] Nick Szabo. Smart contracts: Building blocks for digital markets. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/L0Twinterschool2006/szabo.best.vwh.net/smart_contracts_2.html, 1996.
- [Vuk16] Marko Vukolic. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. https://link.springer.com/chapter/10.1007%2F978-3-319-39028-4_9, 2016.

Acronyms

BFT Byzantine Fault Tolerance. 1, 4

CA Certification Authority. 4, 5, 9

DApp Decentralised Application. 3, 6

EVM Ethereum Virtual Machine. 6

MSP MemberShip Service Provider. 4, 5, 9

P2P peer-to-peer. iii, 1, 2, 27

PKI Public Key Infrastructure. 3, 9, 23, 27

PoA proof of authority. i, iii, 4, 23

PoS proof of stake. 6, 7, 23

PoW proof of work. 2, 3, 5–7, 23, 24, 27

SMR state-machine replication. 4, 5

Glossary

Consensus The majority agrees (to an update) on the valid global state. 1, 2

Fork Nodes append different blocks to their local chain corrupting the mutual global state. Nodes have to agree on which is the valid one. 5, 24

Grover's algorithm A quantum algorithm that consistently solves unstructured search problems or finds the input to black box functions. 23

Mining The computational effort of trying to create a valid block by solving a cryptographic puzzle. 2, 3, 5

Sharding Partitioning the blockchain into smaller chunks ("shards") to increase performance. Each shard stores its own state and transaction history. 23

unspent transaction output A chain of ownership associated with a digital signature. The UTXO represents one's funds. 4, 6

