universität wien

BACHELORARBEIT / BACHELOR'S THESIS

Titel der Bachelorarbeit / Title of the Bachelor's Thesis

„Implementation of a private blockchain system“

verfasst von / submitted by

Samuel Mitterrutzner

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

Bachelor of Science (BSc)

Wien, 2022 / Vienna, 2022

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:	BSc Computer Science
Betreut von / Supervisor:	Univ.Prof.Dr. Wolfgang Klas
Mitbetreut von / Co-Supervisor:	Dr. Belal Abu Naim

Abstract

Blockchain is a very new and innovative technology that has received a lot of attention in recent years. As this technology finds its way more and more into user's everyday life, it becomes even more important to understand the concepts behind it. This thesis therefore aims to design and implement a private blockchain from scratch. Thereby important ideas will be brought closer and alternatives will be discussed. The implemented blockchain is based on the PoA consensus algorithm, which defines exactly how the individual nodes communicate with each other and make decisions.

Contents

List of Figures

Listings

1. Introduction

In recent years, Blockchains have managed to enter our everyday life. Therefore, several applications have been developed over time that even go far beyond the concept of cryptocurrencies - from DApp to NFT marketplaces and blockchain-based games. Much of it is still in its infancy, but the technology seems to be becoming more and more established.
The popularity is largely due to the success of cryptocurrencies such as Bitcoin [18] or ETH [25]. They were also the first revolutionizing applications of the blockchain technology that enabled a financial system without a trusted bank as a third party. Despite cryptocurrencies blockchains have some interesting properties:

- No trusted Third-party:they do not require a trusted thirth-party and instead work as a closed system.
- Immutability: Through cryptography and signatures, the data stored in the blockchain is very secure which makes it the ideal place to store immutable and sensitive data.
- Decentralized: The different nodes that maintain the blockchain work independently of their geographical location.

However, to better understand this complex system, this thesis is fully dedicated to the implementation of a private Blockchain. The goal is to provide a complete design, with all the building blocks needed for a private blockchain, as well as the corresponding implementation. My blockchain system is based on the idea of Twitter, a system that allows people to share their personal opinions with others. The blockchain ensures that tweets cannot be modified once they are saved and that tweets are only saved with valid signatures. Therefore only users with valid key-pairs are able to post tweets.
In the course of this thesis, I will discuss several important topics that are necessary to implement a private blockchain. Specifically, my blockchain is based on the PoA consensus algorithm, which defines which blocks are added and which are rejected. During the design process, it was important to find the right tradeoffs to get a satisfying result. This is also the chapter where I discuss the data model, the basic structure of my project, and the general architecture. After the design is discussed, I elaborate on the most important parts of my implementation. This is also the part where I explain a lot of code snippets to give the best possible impression of my implementation. This should make it easier to understand and reproduce my work. At the end any limitations of my implementation will be discussed and the whole work will be summarized.

2. Related Work

This chapter provides a background for important topics that will be covered throughout this thesis as well as discusses related researches and studies that are conducted in the field of private blockchains.

2.1. Blockchains in General

What exactly is a blockchain? A blockchain in the simplest definition is a distributed database (more precisely a ledger) that is maintained by different peers, also called nodes. Those nodes maintain the blockchain by mining and validating new blocks. Usually, when a node mines a new block for the blockchain, the miner is rewarded by getting some amount of cryptocurrencies from the blockchain. Since everyone can participate and help mining blocks, nodes in a public blockchain don't know who can be trusted. For that reason, there needs to be a trust-less process in place that lets them agree on what blocks are valid and should be added to the ledger. This is exactly what the consensus algorithm does: provide a trustless process to achieve an agreement.
There are several consensus algorithms for public blockchains: One such algorithm is PoW (Proof of Work) which was introduced by Satoshi Nakamoto in 2008 [18] and is the first as well as still the most popular consensus algorithm among blockchains. To reach a consensus one of the peers has to find a hash that has a predetermined number of leading zeroes.
Another example would be PoS (Proof of Stake) which addresses some of the shortcomings of PoW such as better energy efficiency. This results in reduced hardware requirements and lowers the barrier to entering and participating. The algorithm relies on the assumption that users with the biggest stake are interested in the success of the blockchain, where the stake is usually the number of coins a particular node holds. [23] And that's also how the miner of the next block is selected: proportionally to its stake.
A private blockchain is essentially the same as a public blockchain with the difference that in private blockchains the members are restricted and in most cases controlled by one or a few organizations. [20] Another key difference is that since the nodes can trust each other they can make use of more performant consensus algorithms with higher throughput.

2.2. Ethereum ecosystem: Clique and Aura

A relatively new family of consensus algorithms for private blockchains is PoA which was originally proposed by the co-founder of Ethereum Gavin Wood in 2015, as part of the ecosystem for private networks [14]. PoA relies on 2 assumptions [14]:

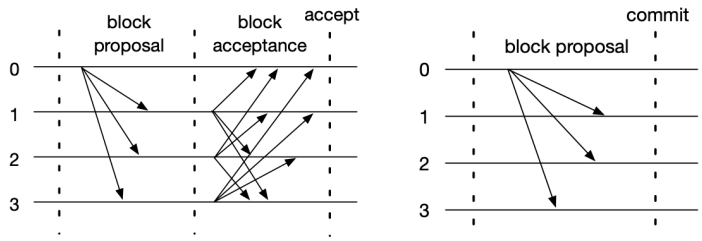
- There exists a set of N nodes that can be identified by a unique id
- At least *N/2 + 1* of those nodes can be trusted

The consensus in PoA algorithms relies on a *mining rotation* schema, that is widely used by blockchains to fairly distribute the block mining among a set of known nodes. [14] It works as follows [11]:

- There exists a fixed number of nodes with known identities.
- The block creation is done in a fixed time interval (other literature also refers to this fixed interval as "turn") that ensures that in this interval the block mining and validation can be done before a new block has to be created. In this interval, a mining-leader is selected to propose a new block that the other nodes will then check.
- If the current node that should mine the new block does not mine the block in the specific interval, it misses the next round.

[11] proposes a fixed order for the selection of the current mining leader. However, for my implementation I made use of a simple scheduling algorithm: Round Robin which selects the next node that should mine a new block.

As the algorithm can assume that more than half of the nodes can be trusted there is no need for solving a computationally expensive problem. This gives PoA the advantage that the block creation happens in fixed intervals and is therefore predictable. In addition, this also requires much less computational power and is therefore cheaper.



(a) Aura (b) Clique

Figure 2.1.: Depicts the different messages for Aura and Clique. In both cases the node with id 0 is selected as the current leader. [14]

2.2.1. Aura (Authority Round)

Aura is the PoA algorithm implemented in Parity [8], which is a Rust-based Ethereum client. [14] In Aura there are two rounds before a new block is added to the ledger: *block proposal* and *block acceptance*. Looking at the Figure 2.1 it can be seen that the rounds

are separated by the dashed horizontal lines. In addition, this figure shows clearly that the first round is simply the process where the current leader proposes a new block that is broadcasted to all the other nodes, while the second round is simply accepting or denying the validity of the proposed block.
The nodes in Aura are synchronized within the same UNIX time *t*. The selection of the current-leader is done with this simple formula: l = s *mod* N, where *l* is the id of the next leader, *s* is the current step and *N* is the total number of nodes. The current step *s* is calculated using the constant *step_ duration*: s = t / step_duration. [14]

2.2.2. Clique ("Inner circle")

Clique is the PoA algorithm implemented in Geth [2], which is a GoLang-based Ethereum client. Clique uses a different approach than Aura: it only needs one round to add a new block to the ledger. In the context of Clique such a round is called *epoch* and it is identified by a prefixed sequence of blocks. [14]
The main difference between Aura and Clique is that by applying Clique in every *epoch* there can be *N - (N/2 + 1)* nodes that are allowed to propose a new block, where one of them is selected as leader. However, this leads to the downside that there can exist different forks of the blockchain at the same time which increases the complexity of the algorithm. To address this issue the likelihood of a fork is limited because the proposed blocks of non-leader nodes are delayed by a random time. By doing so, the likelihood that all the other nodes receive the proposed block from the leader node first increases. [14, 22] But for the case that forks do happen they are resolved using the GHOST protocol [21, 25].

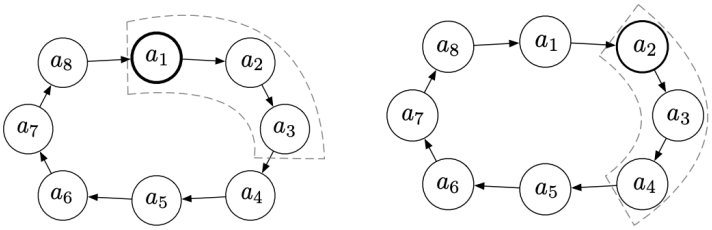


Figure 2.2.: Selection of nodes that are allowed to propose a block: *8 - (8/2 +1) = 3* [14]

2.3. VeChainThor Blockchain

VeChainThor is the most popular blockchain that makes use of the PoA algorithm. The VeChain-team proposed a solution that follows the main goal of "Building a trust-free and distributed business ecosystem platform to enable transparent information flow, efficient collaboration, and high-speed value transfers." [12].

According to, [12] the four main reasons why enterprise and large consumer-focused applications are not yet making use of the blockchain are the following:

- **Governance model**: Decentralization is a big part of blockchains but at the same time makes them harder to upgrade.
- **Economic model**: Blockchains usually are very volatile and unpredictable regarding their cost.
- **Combination of technologies**: To provide real value, the expertise of just blockchain experts is not sufficient. There is also the need of including experts of the following fields: IoT, Big Data and AI.
- **Regulation and changes**: Blockchains are still in a very early stage and a lot of changes can happen.

I used a similar approach as the VeChain team when it comes to the government model of the blockchain: "Neither total centralization nor a total decentralization are the correct answer, but a balance of both." [12] The VeChainThor Blockchain introduces a deterministic pseudo-random process and a concept that assigns an "active" or "inactive" status to a particular node, in order to select who is going to produce the next block. Since VeChain uses the timestamp inside the DPRP, the value can differ from node to node and this could result in multiple "active" nodes. In order to prevent this issue of having multiple nodes creating the same block I added a load-balancer that selects an active node in a fixed time interval. By introducing a more centralised approach the problem of having multiple "active" nodes and therefore also multiple versions of the chain is not a problem anymore. That's the case because every node knows who was selected by the load-balancer as active and therefore also knows from whom it will receive the next block to validate.

3. Design

This chapter provides a description of the concepts integrated in the system and goes into more detail about the design decisions I made for setting up the implementation of the private blockchain.

3.1. Big Picture

3.1.1. Concept / Idea

The idea of my private blockchain is to allow users to express their opinion on any topic that interests them. In short, my system is similar to Twitter, but without the social media character such as, comments, and followers. Another difference is that my application is backed by a private blockchain. This has the advantage that tweets cannot be changed once they are saved and they can be verified by all other users.
To implement such a system, the following functionalities must be available for the users: they should be able to register, as well as log in, post tweets, and query tweets.
From the blockchain perspective, it is important to verify that every tweet is valid. Valid means that if the tweet belongs to person "A" it can only be added to the ledger if the tweet actually does come from person "A" otherwise it would not be added to the ledger.

3.1.2. Consensus Algorithm

The consensus algorithm for my private blockchain implementation is Aura and comes from the Proof of Authority consensus algorithm family. The reason why I chose Aura over Clique is that Aura does not have a problem with multiple forks of the ledger. This makes it easier to implement and also allows for more efficient resource management because a block is created by one node only. This also comes with the drawback that to reach an agreement the nodes have to exchange more messages and also need two rounds [14].
Looking at the Figure 3.1 it is possible to see a simplified example of how the blockchain achieves consensus. First, the user has to generate its key pair and log in. Once he is authorized he can start posting tweets. Those tweets are then broadcasted to all the validation nodes. The Authority node then selects one node that creates a new block out of the three validation nodes (in this case *:ValidationNode1*). This node (also called leader) then creates a new block and sends it to all the other validation nodes, which then validate the block and send it to all the other nodes along with their validation result. Once *N/2 + 1* nodes agreed that the block is valid it will be added to the ledger.

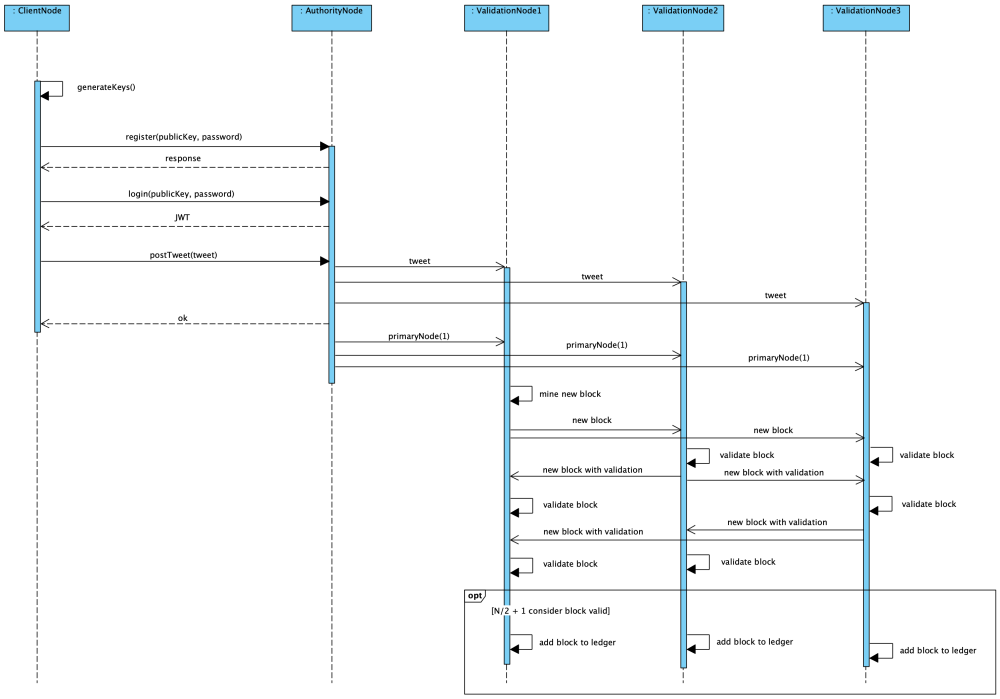


Figure 3.1.: Sequence diagram that demonstrates how the blockchain achieves consensus.

3.2. Architecture

In the implementation of my blockchain system, the first step was to design the model classes. This comes with the advantage that all the data which is going to be sent within my system is already known. After defining the model classes, I tried several approaches: A layered architecture where the ledger - the core of my blockchain - is located at its center and everything else is built on top of it. However, I dropped this approach after some iterations, because all my services were coupled with others and dependencies arose which should not be present.
After some failed designs, I took a step back and defined what I actually expect from my architecture. I wanted an architecture that allows me to make fast iterations with no code duplication and high SoC. Therefore, my next approach was to create a component-based architecture that divides all processes into independent components: Authority Node, Validation Node, and Client Node. However, by doing so the problem appeared that all these components need to access the model classes, which causes a lot of code duplication and results in a very inflexible data model. So I have defined another component: A ledger, which contains all model classes, hash algorithms, key pair management, and data structures. This component can then be used by everyone else to avoid code duplication. I had a similar problem when I started modeling the communication between validation nodes and the authority node. The issue with code duplication appeared also here but in

addition to that, the SoC principle was also violated, because it's not the responsibility of the different nodes to implement a way of communicating with each other. They make use of it but should not provide their implementation. That's why I added another component, which implements the communication among the validation nodes as well as the communication between the validation node and the authority node.
A visual approach to my architecture can be seen in Figure 3.2.

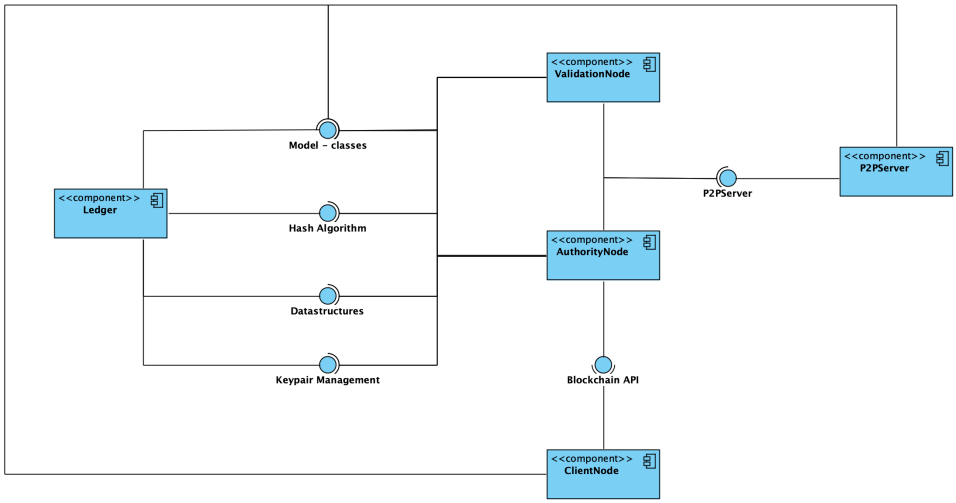


Figure 3.2.: Component Diagram of the high level architecture

3.3. Main building blocks

3.3.1. Ledger - Module

As the name suggests, this module contains everything that has to do with the ledger. This is the place where data structures, model classes, and everything related to encryption and decryption are located. By separating all this logic in a separate module it is very easy to share code. That's the case because every module can import another module if needed. This comes in very handy as both the validation nodes and the authority nodes need the logic for encryption and decryption as well as most of the model classes.

Data structures

Since the validation of tweets is a fundamental building block of the whole system, a data structure that can enable a faster verification in necessary. And one of the fastest and most common ways to check for integrity is to construct a merkle tree and check the root node. The merkle tree was developed by Ralph Merkle for digitally signing datasets with

the goal of fast verification of data consistency. [17] It is essentially a binary - tree that is perfectly balanced with the Leaf-nodes that contain the transactions [24] or in this case contain the tweets. More about the construction and validation of merkle trees follows in chapter 4 Implementation.

Data - Model

The data model I have chosen is kept rather simple: By looking at the data structures of popular blockchains such as Bitcoin and Ethereum, I noticed that both of them had similar approaches. Both of them store the hash of the block along with the hash of the previous hash and the root of the merkle tree [18, 16]. If the implementation is based on such a data model, it comes to a major benefit: Immutability. The hash of the current block is calculated as follows: *SHA256(previousHash + merkleRoot + timestamp)*. As the hash of the previous block is included it becomes very hard to modify a block once one is added to the ledger. Since the block-hash value can change completely, once one of the attributes mentioned previously changes, other nodes can very easily detect modifications in a block just by calculating the hash.
To be able to store the merkle tree there was a need to define a hierarchical entity that stores a reference of the left and right - node as well as a reference to a tweet. As can be seen in Figure 3.3 the *MerkleNode* table has a recursive relationship that creates a parent-child hierarchy which is exactly what is needed to store a tree-like data structure.

To persist the tweets of the different users, the blockchain stores along with the content of the tweet, the public key and a signature. The public key is stored to easily query tweets of a specific user and to verify that the signature was created with the matching private key. The signature is a 2048-bit long string that was created by signing a tweet with a private key and serves as proof of the authenticity of a tweet.

My initial approach was to use a No-SQL Database to store all this information but it turned out to be not an optimal decision. Before talking about why an RDBMS is better suited for the data model outlined above let's look at the advantages of No-SQL (specifically MongoDB [7]) first: Horizontal scaling, fast key-value lookups and Dynamic Schema. My first draft wanted to take advantage of the fast key-value lookups by storing the complete merkle tree in a single document and using the hash value as the primary key (called *_id* in MongoDB [7]). The big problem with this approach was that there is a document size limit of 16Mb, which requires the blockchain to limit the tweets per block and that would result in a smaller throughput. Another idea was to use manual references (storing the string value of an *_id*) but this wasn't a good approach either because querying the data became less performant and also harder because of the hierarchical structure of the data.

By using PostgreSQL it became much simpler to store and also query the data. Also, drawbacks such as fixed schema aren't such a big deal as the schema does not change. It may become harder to extend it in the future but still not impossible.

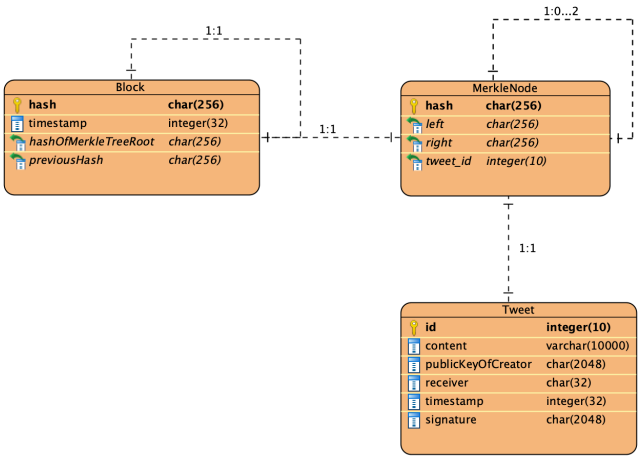


Figure 3.3.: Entity relationship diagram of the data model.
The *yellow-key* means that the column is a primary key, *green-arrow* means that the columns is a foreign key.

3.3.2. P2PServer - Module

The next module that is used by multiple other modules is the P2PServer - Module. It contains everything related to the communication between the authority node and the different validation nodes. The communication is realized by implementing the broker pattern.

Broker Pattern

The Broker pattern allows for the isolation of the communication-related code from the rest of the distributed system. It comes with the benefit that all the communication-related code is in one place, so there is no code duplication and good separation of concerns. [26]. A Node can easily participate in the Peer-to-Peer Network by simply providing an Implementation of the *Callback*-interface and invoking the *register(callback)* method of the *P2PServer*-class. The *Callback* implementation contains the logic of how a node reacts to a specific type of massage.

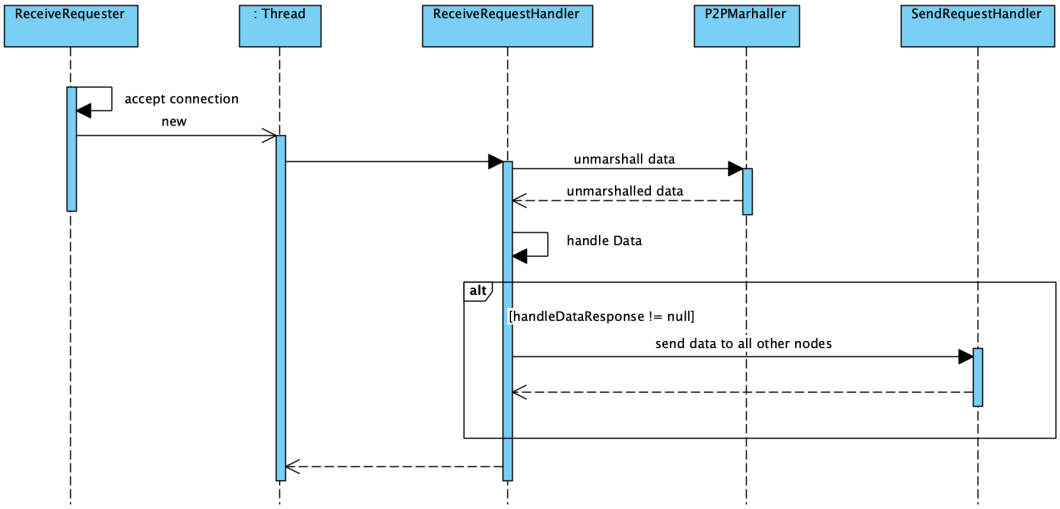


Figure 3.4.: Sequence Diagram of how messages are processed.

3.3.3. Authority Node

The Authority Node is the main access point to the blockchain for the users. It provides a rest endpoint for authentication, posting, and fetching tweets. The authority node has also another very important task which is load balancing: Since the consensus algorithm used in this blockchain is PoA, there has to be a process in place that selects the next validation node that creates a block. This is done by the authority node by implementing a round-robin scheduling algorithm.
The Authority node could be best described as an oracle. An oracle is a bridge between the blockchain and other external systems [15]. Introducing oracles comes with some benefits and drawbacks. As described before the Authority node is used for load balancing which allows me to better distribute the work of the validation nodes and also prevents multiple nodes from creating the same block. But this comes at a cost: Decentralization gets lost to some degree when connecting the blockchain to a single point of failure. This problem is also called the "oracle problem" [15]. The issue of using a more centralized approach is a big concern in public blockchains but not as much in private blockchains, assuming that it will increase the performance.

To authorize the HTTP requests the Authority node uses JWT's (JSON Web Token). A JWT is returned from the Authority Node once a user successfully logs in. Users then can use this token to authorize their HTTP requests and be able to post and fetch tweets. Using those tokens has the benefit that a user has to log in with a password first to be even able to use his key pair to post tweets. By doing so there is an extra layer of security for the users in case their private key is stolen.

3.3.4. Validation Node

The validation nodes make up the core of the blockchain system. They are the ones that maintain the blockchain by mining new blocks and validating proposed blocks. To reach a consensus on what blocks are valid or invalid, they implement the Aura protocol of the PoA consensus algorithm. A simplified example of how a new block is added to the ledger can be seen in Figure 3.1.

Block Mining

Since the consensus algorithm doesn't require a validation node to solve a complex puzzle the process of mining a block is very straightforward. The nodes need to construct a merkle tree, fetch the hash of the last block added to the ledger and then apply the hash function to the block. That's everything that a node needs to do in order to propose a block.

Block Validation

For a block to be considered valid by a validation node it needs to fulfill three Properties:

- All Tweets must be valid: A tweet is considered valid if the signature was created with the correct private key. Figure 3.5 shows the process of validating a single tweet.
- The Merkle Root must be correct: The merkle root is correct if the merkle root stored in the block is equal to the merkle tree root that a validation node gets when constructing a merkle tree with the tweets that are included in the current block.
- Previous Hash must be known: The validation node always checks if the previous hash value that is stored in the block is the last block that was added to the ledger.

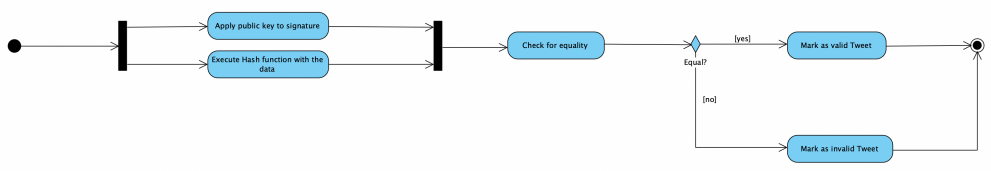


Figure 3.5.: Activity Diagram that shows how a tweet is validated.

3.3.5. Client Node

The client node is not part of the blockchain anymore. It can be seen as a third-party application that makes use of the blockchain system. It includes a simple CLI that lets a user interact with the blockchain by making use of the provided REST-API of the authority node. The CLI provides commands to generate key-pairs, login, register, post

tweets and fetch tweets based on their public key or based on a hash value of another tweet.

4. Implementation

In this chapter, the concrete implementation of the design, described in the previous chapter, will be explained. To map the architecture as best as possible, a multi-module Maven [6] project was used, where a module corresponds to a component in the design.

4.1. Technology Stack

One of the first decisions I had to make was which technologies I wanted to use. This decision was fundamental and very hard to undo, so I spent a lot of time choosing the best stack possible. It required me to make trade-offs between what I had the most experience in and what would be the optimal choice to fulfill the needs of a private blockchain system.

4.1.1. Programming Language

Before talking about Frameworks or libraries, I had to determine what programming language to use. For me, there were two potential candidates for the project: Typescript and Java. Both languages support object-oriented programming and are statically typed. However, there is a clear winner for me and that is Java. The main reason for this decision is that Java is the language with better performance and allows me to make use of Java's multi-threaded features.

4.1.2. Frameworks and Libraries

Once the decision on the programming language was made, it was a matter of selecting a framework that would simplify setting up a rest endpoint, implementing authentication, and establishing a connection to the database. Several frameworks in the Java environment meet these requirements. The best known and most used is Spring Boot [9]. Since there are a lot of best practices and already established solutions for known problems such as authentication, I decided to use Spring Boot in combination with Spring Security.
In addition to this framework the following libraries were used:

- java-jwt [3]: Is a library from auth0 that provides an implementation of JWT.
- lombok [5]: Makes the development process easier by providing simple annotations that avoid boilerplate code.
- javax [4]: Used for marshaling from Java representation to XML and the other way round.

- testcontainers [10]: Used for testing. Testcontainers provide lightweight instances of common databases.

For hashing and key-pair generation the Java security library was used.

4.1.3. Build Tools

To run the project locally, it was necessary to run separate database instances at the same time, since each single validation node is connected to its database. To fulfill this requirement I resorted to a very popular tool: Docker. Docker allows me to build containers, where a container is an abstraction at the app layer that packages code and dependencies together [1], in which a database can run.

4.2. Blockchain System

4.2.1. Cryptography, Hash-Algorithm

One of the most fundamental parts of my system in terms of security is cryptography. For my blockchain to be able to uniquely identify and verify users, as well as validate whether the tweet is from the person in question, a public-key cryptosystem is used. I decided to use the RSA (Rives-Shamir-Aldeman) [19] algorithm implementation of the java-security library, simply because the algorithm is one of the oldest and most established.
The hash algorithm on which the entire system is based on, is the implementation of the java security library of SHA-256 (Secure Hash Algorithm). The reason for this decision is that this algorithm has never been cracked and many other well-known blockchains such as Bitcoin rely on it [13].

4.2.2. Merkle Tree

As described in the previous chapter, Merkle trees are used to increase the efficiency of validation. The following code example 4.1 shows exactly how such a tree is created. First, all tweets that should be stored in this Merkle tree are converted to MerkleNodes and then finally inserted into the tree. The tree is created from the bottom up so that the MerkleNodes that contain a tweet are at the bottom. The tree is created in a way such that the parent always contains the following hash value *SHA256(left.hash, right.hash)*. Since the smallest change in the input can completely change the output of SHA-256, it is possible to detect modifications very easily.
The validation of a Merkle tree is done by creating one with the same tweets and then comparing the two roots. If they are the same the tree is correct and if they are not the tree is not correct.

```
1 private MerkleNode buildTree(List<MerkleNode> messageNodes,
    MerkleTreeSaveCallback callback) {
2     if (messageNodes.size() == 1) {
3       this.root = messageNodes.get(0);
```

```
4
5       if (callback != null) {
6         callback.save(this.root);
7       }
8
9       return messageNodes.get(0);
10    }
11
12    // keeps track of the parent nodes that are created in each iteration
       of the while loop
13    List<MerkleNode> currentParentLevelNodes = new ArrayList<>();
14
15
16    // keeps track of the child notes in order to create the
17    // corresponding parent nodes
18    List<MerkleNode> currentChildrenNodes =
19        new ArrayList<>(messageNodes);
20
21    // once only one node is left in currentChildrenNodes the tree
22    // is complete and the last node is the root node
23    while (currentChildrenNodes.size() > 1) {
24
25      // if the callback exists the data is saved as specified
26      // by the callback
27      if (callback != null) {
28        currentChildrenNodes.forEach(callback::save);
29      }
30
31      // iteration where the parent nodes are created
32      for (int i = 0; i < currentChildrenNodes.size(); i += 2) {
33        MerkleNode left = currentChildrenNodes.get(i);
34        MerkleNode right = null;
35        if (i + 1 < currentChildrenNodes.size()) {
36          right = currentChildrenNodes.get(i + 1);
37        } else {
38          right = left;
39        }
40        MerkleNode parent = new MerkleNode(left, right, null);
41        currentParentLevelNodes.add(parent);
42      }
43
44      // for the next iteration the parent nodes become the children
45      // nodes because we need to generate their parents
46      currentChildrenNodes = currentParentLevelNodes;
47      currentParentLevelNodes = new ArrayList<>();
48    }
49
50    // if the callback exists the data is saved as specified by
51    // the callback
52    if (callback != null) {
53      currentChildrenNodes.forEach(callback::save);
54    }
55
```

```
56      this.root = currentChildrenNodes.get(0);
57      return root;
58  }
```

Listing 4.1: Merkle Tree Construction

4.2.3. Block Mining / Validation

After the cryptographic foundations have been laid with RSA encryption and SHA-256 and the Merkle tree have been implemented, mining a block is very simple. Creating a MerkleTree consisting of the current tweets and fetching the hash of the previous block from the database is everything that has to be done. Validating a block is a bit more complicated than mining but still relatively simple. All three conditions described in 3.3.4 Block Validation have to be checked, which is exactly what the code snippet in 4.2 does.

```
1  public boolean validateBlock(BlockWithValidation block) {
2      // ckeck if all tweets are valid. To validate a tweet the signature
3      // is check by using the SHA256withRSA Signature instance provided by
4      // the java security library
5      boolean areTweetsValid = block.getTweets()
6          .stream()
7          .allMatch(TweetValidator::isTweetValid);
8
9      MerkleTree merkleTree = new MerkleTree();
10     merkleTree.build(block.getTweets());
11     boolean isMerkleTreeValid = merkleTree
12         .getRoot()
13         .getHash()
14         .equals(block.getHashOfMerkleTreeRoot());
15
16     boolean previousHashKnown = repository.findByHash(block.
       getPreviousHash()).isPresent();
17
18     log.info("Block valid? Merkle tree {}, tweets {}, previous hash known
        {}", isMerkleTreeValid, areTweetsValid,
19       previousHashKnown);
20
21     return isMerkleTreeValid && areTweetsValid && previousHashKnown;
22 }
```

Listing 4.2: Block validation implementation

4.2.4. Peer To Peer Network

Before we can dedicate ourselves to the consensus algorithm, a P2P network must first be available so that the individual nodes can communicate with each other. Due to the consensus algorithm blocks have to be created and validated in a certain interval which implies that the communication has to be multi-threaded. In this way a validation node can start mining the block directly after it has been selected as the leader and does not have to wait until other messages are finished processing. To make the communication

multi-threaded I defined two classes which do exactly that: *ReceiveRequester* and *ReceiveRequestHandler*. The *ReceiveRequester* starts a new thread with a *ReceiveRequestHandler* instance every time a connection to another node is accepted. The processing of the request is then taken over by the *ReceiveRequestHandler* and continues in another thread. In this way the *ReceiveRequester* can continue accepting new connections while the *ReceiveRequestHandler* is processing the request.

```
1  public class ReceiveRequester implements Runnable {
2    private final ServerSocket serverSocket;
3    private final P2PCallback callback;
4    private final ExecutorService handlers;
5    private final List<ConnectionDetails> connections;
6
7    private volatile boolean isRunning = true;
8
9    public ReceiveRequester(ServerSocket serverSocket, P2PCallback callback
       , List<ConnectionDetails> connections) {
10     this.serverSocket = serverSocket;
11     this.callback = callback;
12     this.connections = connections;
13     this.handlers = Executors.newFixedThreadPool(10);
14   }
15
16   public boolean terminate() throws InterruptedException {
17     isRunning = false;
18     handlers.shutdown();
19     return handlers.awaitTermination(5,
20         java.util.concurrent.TimeUnit.SECONDS);
21   }
22
23   @Override
24   public void run() {
25     log.info("P2P Server is ready to receive requests on {}:{}",
       serverSocket.getInetAddress().getHostName(),
26       serverSocket.getLocalPort());
27     while (isRunning) {
28       try {
29         handlers.submit(
30             new ReceiveRequestHandler<HashableEntity>(
31                 serverSocket.accept(), callback, connections
32             )
33         );
34       } catch (JAXBException | IOException e) {
35         e.printStackTrace();
36       }
37     }
38   }
39 }
```

Listing 4.3: Implementation of the *ReceiveRequester* class label

Message Processing

My P2P network defines three different types of messages that can be sent: TWEET_RECEIVE, BLOCK_RECEIVE, PRIMARY_NODE_SELECTION. The message is then processed differently based on its type. How exactly the processing of the different message types looks like is defined by an implementation of the *P2PCallback* interface. The code in 4.4 shows how the incoming messages are handled.

```
1  private Optional<? extends Envelope<? extends HashableEntity>>
       handleMessage(Envelope<T> envelope) {
2      switch (envelope.getType()) {
3        case BLOCK_RECEIVE:
4          BlockWithValidation blockWithValidation =
5              (BlockWithValidation) envelope.getData();
6          return callback.onBlockReceived(
7              new Envelope<>(blockWithValidation, envelope.getType()));
8        case TWEET_RECEIVE:
9          Tweet tweet = (Tweet) envelope.getData();
10         return callback.onTweetReceived(
11             new Envelope<>(tweet, envelope.getType()));
12       case PRIMARY_NODE_SELECTION:
13         ConnectionDetails primaryNode =
14             (ConnectionDetails) envelope.getData();
15         return callback.onPrimaryNodeSelected(
16             new Envelope<>(primaryNode, envelope.getType()));
17       default:
18         return Optional.empty();
19     }
20 }
```

Listing 4.4: Implementation of the message handling located in the *ReceiveRequestHandler*-class

How to connect?

As described in 3.3.2, connecting to the P2PServer is very simple. All you need is an instance of the P2PServer class that needs a list of all the connection details of the other nodes and a ServerSocket. Then simply call the method *register(P2PCallback callback)* from the P2PServer instance and as a result, the respective node is part of the P2P network.

4.2.5. Proof of Authority

As the previous topics have been all implemented, we can finally turn to the heart of any blockchain, the consensus algorithm. As already described in 3.1.2 my implementation uses the Aura protocol of the Proof of Authority consensus algorithm. Therefore, it is required to implement three processes in order to understand how the individual validation nodes react when they receive a proposed block or are selected as the primary node and to detect how exactly the selection of the primary node works.

Two of these processes are taken over by the validation nodes by providing an implementation of the P2PCallback interface. When looking at code in 4.5, those are the methods that are then called by the *ReceiveRequestHandler* in 4.4.

```
1  /**
2   * Interface that contains a callback method for every @{@link
       EnvelopeType}
3   */
4  public interface P2PCallback {
5    /**
6     * The callback method that is called when a tweet is received.
7     * If the method returns an envelope then this envelope will be sent
8     * to all the other nodes.
9     *
10    * @param tweetEnvelope envelope contain {@link EnvelopeType#
       TWEET_RECEIVE}
11    */
12   Optional<Envelope<Tweet>> onTweetReceived(Envelope<Tweet> tweetEnvelope
       );
13
14   /**
15    * The callback method that is called when another node send a block
16    * that was mined and needs verification. If the method returns an
17    * envelope then this envelope will be sent to all the other nodes.
18    *
19    * @param blockEnvelope envelope contain {@link EnvelopeType#
       BLOCK_RECEIVE}
20    */
21   Optional<Envelope<BlockWithValidation>> onBlockReceived(Envelope<
       BlockWithValidation> blockEnvelope);
22
23   /**
24    * The callback method that is called when the node was
25    * selected to mine a block.
26    */
27   Optional<Envelope<BlockWithValidation>> onPrimaryNodeSelected(Envelope<
       ConnectionDetails> primaryNodeConnectionDetails);
28 }
```

Listing 4.5: Callback interface that defines the methods that are called based on the message type (*EnvelopeType*)

Proposed block request

When a proposed block is sent to all validation nodes over the P2P network, the *onBlockReceied* method is called to validate the block. First, it has to be checked if a block is expected at all, because other validation nodes may send their block validation after the block has already been marked as accepted. To avoid false block proposals it is also checked if the initially proposed block comes from the current primary node. So when a validation node sees a block for the first time, it must come from the current primary node. If all conditions are met then the block is validated and the validation result is

saved. After that there are three possibilities:

- *N/2 + 1* have confirmed the block: Then the block will be added to the ledger and deleted from the pending block list. Tweets that are in this block are also deleted from the current list of tweets.
- *N/2 + 1* did not confirm the block: Then the current block will be deleted from the list of pending blocks along with the tweets that should be included in this block.
- Not enough replies: This simply means that there are not enough validation nodes that have validated the block to make a decision.

```
public Optional<Envelope<BlockWithValidation>> onBlockReceived
  (Envelope<BlockWithValidation> blockEnvelope) {
    assert (blockEnvelope.getType() == EnvelopeType.BLOCK_RECEIVE);

    Envelope<BlockWithValidation> response = null;

    if (this.pendingBlocks.isEmpty()) {
      return Optional.empty();
    }

    // if we receive a block from a validator that it not the
    // primary node before the primary node has finished we ignore it
    if (isBlockComingFromNonPrimaryNode(
        blockEnvelope.getData().getPublicKeyOfValidator())) {
      return Optional.empty();
    }

    synchronized (this) {
      blockValidationCounter.addBlockValidationResponse(
        blockEnvelope.getData().getHash(),
        new Tuple<>(
          blockEnvelope.getData().getPublicKeyOfValidator(),
          blockEnvelope.getData().isValid()
        )
      );
    }

    // validate block if not yet validated by this node
    if (!blockValidationCounter.alreadyValidated(
        blockEnvelope.getData().getHash(),
        this.nodeConnectionDetails.getPublicKey())) {

      BlockWithValidation blockWithValidation =
        validateBlock(blockEnvelope);
      response = new Envelope<>(blockWithValidation,
        EnvelopeType.BLOCK_RECEIVE);

      synchronized (this) {
        blockValidationCounter.addBlockValidationResponse(
            blockEnvelope.getData().getHash(),
```

```
41              new Tuple<>(
42                  this.nodeConnectionDetails.getPublicKey(),
43                  blockWithValidation.isValid()
44              )
45          );
46        }
47      }
48
49      if (blockValidationCounter.isBlockConsideredValid(
50          blockEnvelope.getData().getHash())) {
51        handleBlockIsValid(blockEnvelope);
52      }
53
54      if (blockValidationCounter.isBlockConsideredInvalid(
55          blockEnvelope.getData().getHash())) {
56        handleBLockIsInValid(blockEnvelope);
57      }
58
59      return Optional.ofNullable(response);
60  }
```

Listing 4.6: Implementation of how a validation node reacts to a proposed block.

Primary node selection request

As soon as a message with the type *EnvelopeType.PRIMARY_NODE_SELECTION* is received, the function below is called. It first checks if there are any tweets to be included, then it checks if the current node is selected as the primary node. If it is not selected as the primary node, then the connection details from the primary node are added to the pending block list and no message is sent. However, if it is selected as the primary node it creates the block and sends it to all other validation nodes.

```
1  public Optional<Envelope<BlockWithValidation>> onPrimaryNodeSelected(
2      Envelope<ConnectionDetails> payload) {
3      assert (payload.getType() == EnvelopeType.PRIMARY_NODE_SELECTION);
4
5      List<Tweet> tweets = getTweetsForBlockValidation(
6          payload.getTimestamp());
7
8      if (tweets.size() == 0) {
9        return Optional.empty();
10     }
11
12     // if current node was not selected as primary node return and wait
13     // for the primary node to send a block
14     if (!isCurrentNodePrimary(payload.getData().getPublicKey())) {
15       synchronized (this) {
16         // add the primary node to the list of nodes that have not
17         // yet mined their assigned block
18         pendingBlocks.add(new Tuple<>(payload.getData(), null));
19         // remove all old tweets from the queue
20         tweetQueue.removeIf(tweetEnvelope ->
```

```
21              tweets.contains(tweetEnvelope.getData()));
22        }
23        return Optional.empty();
24      }
25
26      BlockWithValidation blockWithValidation = buildBlock(tweets);
27
28      synchronized (this) {
29        tweetQueue.removeIf(tweetEnvelope ->
30          tweets.contains(tweetEnvelope.getData()));
31        pendingBlocks.add(
32          new Tuple<>(this.nodeConnectionDetails, blockWithValidation));
33      }
34
35      return Optional.of(
36          new Envelope<>(blockWithValidation, EnvelopeType.BLOCK_RECEIVE));
37  }
```

Listing 4.7: Implementation of how a validation node reacts when selected as the primary node.

Primary Node selection

The selection of the primary-node works in such a way that from the list of the validation nodes the first one is taken, which is selected as primary, then the second, third, ... etc. As soon as all nodes were selected once, it starts again from the beginning. This process happens in a fixed interval. Once it is determined who is the primary node for the next block, it is sent to all validation nodes in the P2P network and they take care of the rest.

```
1  private void selectPrimaryNode() {
2      int currentPrimaryNode = currentCount %
3          connectionProvider.getConnections().size();
4      ConnectionDetails primaryConnectionDetails = connectionProvider
5          .getConnections()
6          .get(currentPrimaryNode);
7
8      connectionProvider
9        .getConnections()
10       .parallelStream()
11       .forEach(connection -> {
12         try {
13           Socket socket = new Socket(connection.getIp(),
14             connection.getPort());
15           SendRequestHandler.sendData(socket,
16             new Envelope<>(primaryConnectionDetails,
17                 EnvelopeType.PRIMARY_NODE_SELECTION));
18           socket.close();
19         } catch (Exception e) {
20           e.printStackTrace();
21         }
22       });
```

```
23 }
```

Listing 4.8: Implementation of primary node selection

4.3. Demo Application

The demo app is kept quite simple: Three validation nodes and one authority node are started. In that way the core of the blockchain is initialized and therefore any number of client nodes can be started, which then interact with the blockchain.

4.3.1. Prerequisites

In order to start the app, the following things must be installed:

- Docker [1]
- Maven [6]
- Java >= 11

4.3.2. How to run it

In the first step, all dependencies of prerequisites have to be installed. Once this is done, the necessary frameworks and libraries must be installed. This is done with the following command:

```
# command has to be executed in the root directory
# in order to build and install all dependencies
mvn install
# or this command to skip tests
mvn install -DskipTests
```

The tests can take a bit longer to run the first time, as certain Docker images need to be downloaded. Once all dependencies are installed and the jar-files are generated the Demo-Application can be started by executing the following commands:

```
# change to BlockchainDemo directory
cd BlockchainDemo/target
# execute jar file
java -jar BLOCKCHAIN_DEMO.jar
```

Once the jar-file is running a Client-node to interact with the blockchain can be started. This is done with the following commands:

```
# change to ClientNode directory
cd ClientNode/target
# execute jar file
java -jar ClientNode-0.0.1-SNAPSHOT.jar
```

At this point, everything is up and running so that it is possible to use the CLI provided by the client node. In order to be able to post a tweet, the following commands have to be executed first (using the CLI):

```
# This command will show all commands and how to use them
shell:>help

# Generates a key pair
shell:>generate-keys
# Register with the public key
shell:>register --register ChooseYourPasssword
# Login (password is auto saved by the CLI)
shell:>login

# Now it is possible to use the private blockchain
shell:>post-tweet --tweet "This is a tweet"
shell:>get-my-tweets
shell:>send-invalid-tweet
shell:>get-tweet --hash someHashValue
```

5. Conclusion

The goal of this paper was to develop an implementation of a private blockchain without relying on existing solutions. In order to achieve this, it was first necessary to define a data model as well as to create a peer-to-peer network. In the next step, a consensus algorithm and block mining as well as block validation were implemented. Therefore, this paper can be used as a template to design your own implementation of a private blockchain. From the design to the implementation everything is given that is needed to provide such a complex system. I even included the class diagrams of the different components in the appendix.

5.1. Challenges

Up to the final implementation there were several challenges that had to be overcome in order to provide a finished and working system.

- Consensus Algorithm: This probably the biggest challenge I had to face during the implementation. This is due to the fact that the consensus algorithm has to synchronize the different validation nodes such that they all agree on what the current state of the blockchain is. The issue was solved by using the Aura protocol of the PoA algorithm and introducing the authority node as a load balancer.
- Architecture: Creating a blockchain from scratch and finding the right architecture was not an easy task either. The goal of my architecture was to have very little code duplication as well as a high SoC. This was achieved by using a component based architecture that makes it very easy to reuse components.
- Multithreaded Communication: Another point that had to be solved is multi-threaded communication. Since the Aura protocol requires to exchange more messages than the Clique protocol it was important that those messages can be processed fast. Here I made use of the broker pattern, which allows me to receive and process multiple messages at the same time.

5.2. Limitations

This thesis describes the design and implementation of a private blockchain, but it does not specify under which conditions validators are "excluded" from the blockchain if they misbehave. Misbehaving means that they try to create non-valid blocks or blocks not at

all. Therefore, no process defines how to ignore or exclude a validator.
Another limitation of my implementation is that it is not possible to add more validators once the blockchain is running. This is due to the fact that all validation nodes need to know all connections at build time. The same is true for the authority node.

Bibliography

[1] Docker. https://www.docker.com. Accessed on 01.07.2022.

[2] Geth. https://geth.ethereum.org. Accessed on 30.05.2022.

[3] Java jwt. https://github.com/auth0/java-jwt. Accessed on 28.05.2022.

[4] Jaxb, the java™ architecture for xml binding (jaxb) provides an api and tools that automate the mapping between xml documents and java objects. https://javaee.github.io/jaxb-v2/. Accessed on 28.05.2022.

[5] Lombok. https://projectlombok.org/. Accessed on 28.05.2022.

[6] Maven. https://maven.apache.org/. Accessed on 20.05.2022.

[7] Mongodb. https://www.mongodb.com/.

[8] Parity. https://www.parity.io. Accessed on 30.05.2022.

[9] Spring boot. https://spring.io/projects/spring-boot. Accessed on 28.05.2022.

[10] Testcontainers. https://www.testcontainers.org/. Accessed on 28.05.2022.

[11] Public versus private blockchains part 1 : Permissioned blockchains.

[12] Development plan and whitepaper vechain, May 2018.

[13] Hash algorithm comparison: Md5, sha-1, sha-2 sha-3.

[14] Angelis, S. D., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A., and Sassone, V. Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain. In *Italian Conference on Cyber Security (06/02/18)* (January 2018).

[15] Caldarelli, G., and Ellul, J. The blockchain oracle problem in decentralized finance—a multivocal approach. *Applied sciences 11*, 16 (2021), 7572.

[16] Jezek, K. Ethereum data structures. *CoRR abs/2108.05513* (2021).

[17] Merkle, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87* (Berlin, Heidelberg, 1988), C. Pomerance, Ed., Springer Berlin Heidelberg, pp. 369–378.

[18] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system.

[19] Nisha, S., and Farik, M. Rsa public key cryptography algorithm – a review. *International Journal of Scientific Technology Research 6* (07 2017), 187–191.

[20] Rasolroveicy, M., Haouari, W., and Fokaefs, M. *Public or Private? A Techno-Economic Analysis of Blockchain.* IBM Corp., USA, 2021, p. 83–92.

[21] Sompolinsky, Y., and Zohar, A. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. Cryptology ePrint Archive, Paper 2013/881, 2013. `https://eprint.iacr.org/2013/881` Accessed on 20.05.2022.

[22] Sondhi, S. *Empirical Performance Evaluation of Consensus Algorithms in Permissioned Blockchain Platforms.* PhD thesis, 2021. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2021-10-20.

[23] Spasovski, J., and Eklund, P. Proof of stake blockchain: Performance and scalability for groupware communications. In *Proceedings of the 9th International Conference on Management of Digital EcoSystems* (New York, NY, USA, 2017), MEDES '17, Association for Computing Machinery, p. 251–258.

[24] Szefer, J., and Biedermann, S. Towards fast hardware memory integrity checking with skewed merkle trees. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2014), HASP '14, Association for Computing Machinery.

[25] Wood, G. Ethereum: A secure decentralised generalised transaction ledger.

[26] Zdun, U., Kircher, M., and Voelter, M. Remoting patterns. 353–380.

Acronyms

AI Artificial intelligence. 6

DApp Decentralised application. 1

DPRP deterministic pseudo-random process. 6

IoT Internet of Things. 6

JWT JSON web token. 12, 15

NFT Non fungible token. 1

PoA Proof of Authority. i, 1, 3–5, 12, 13, 27

RDBMS relational database management system. 10

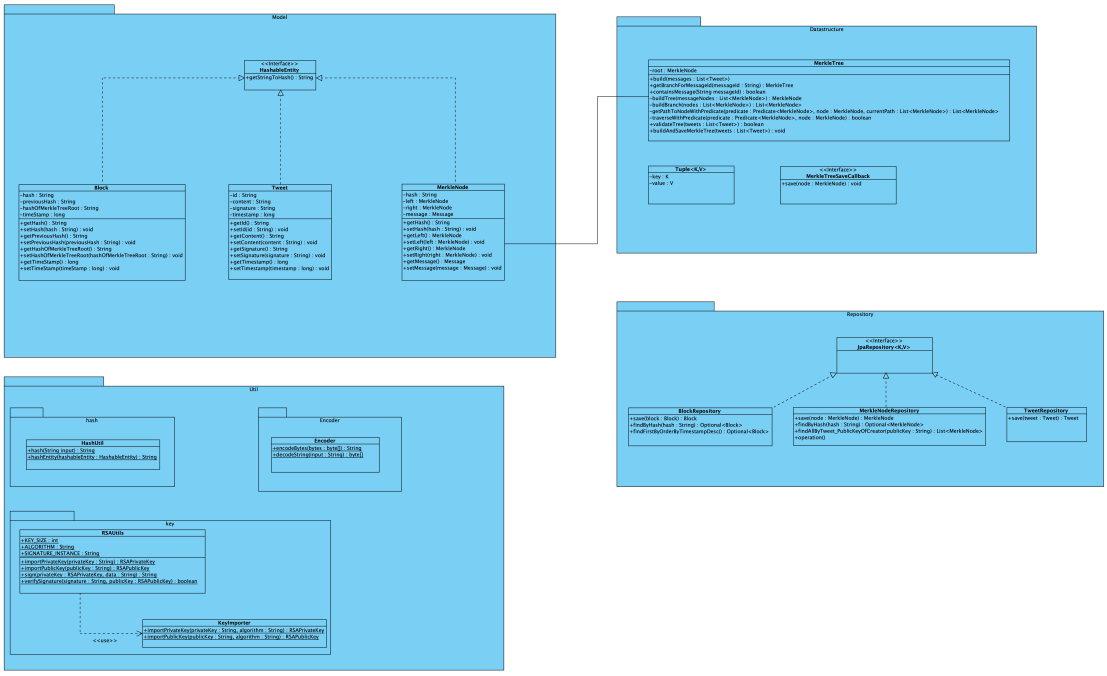
SHA-256 Secure hash algorithm 256. 16, 18

SoC Separation of Concerns. 8, 27

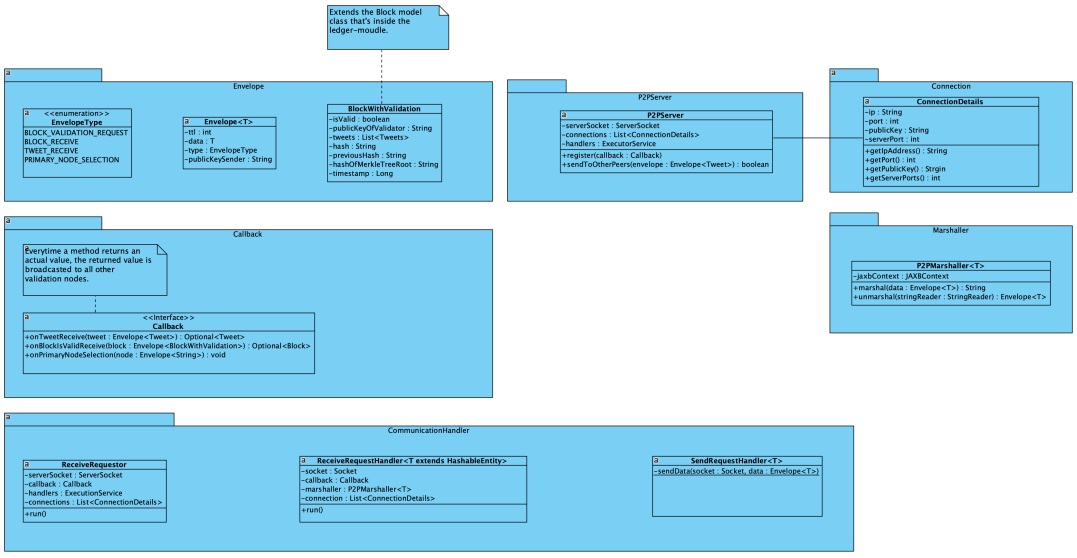
A. Appendix

A.1. Class Diagrams

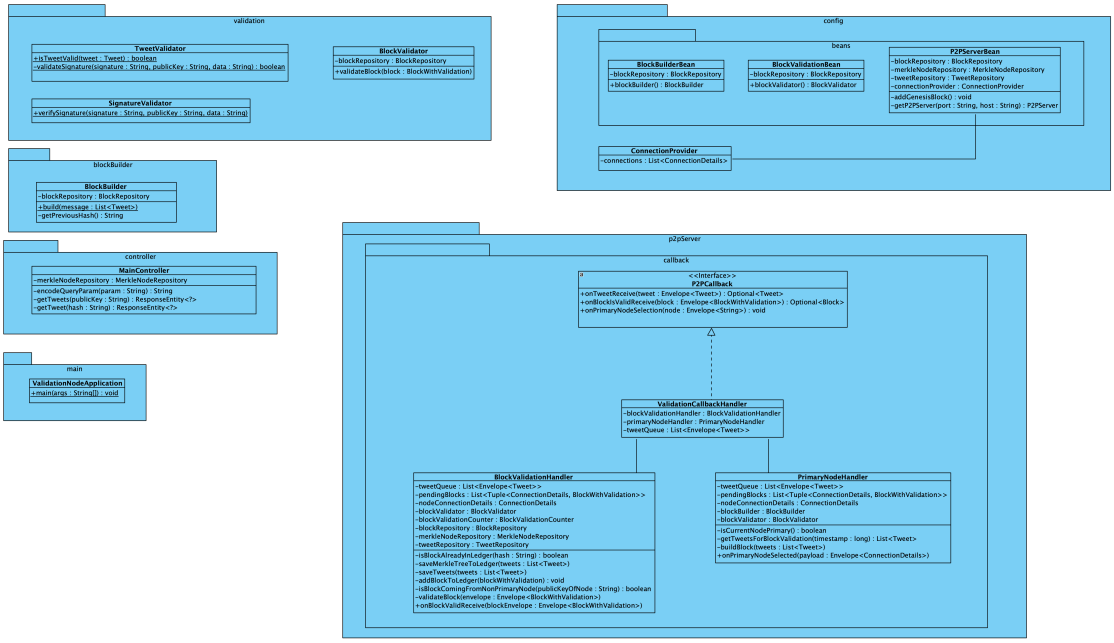
A.1.1. Ledger



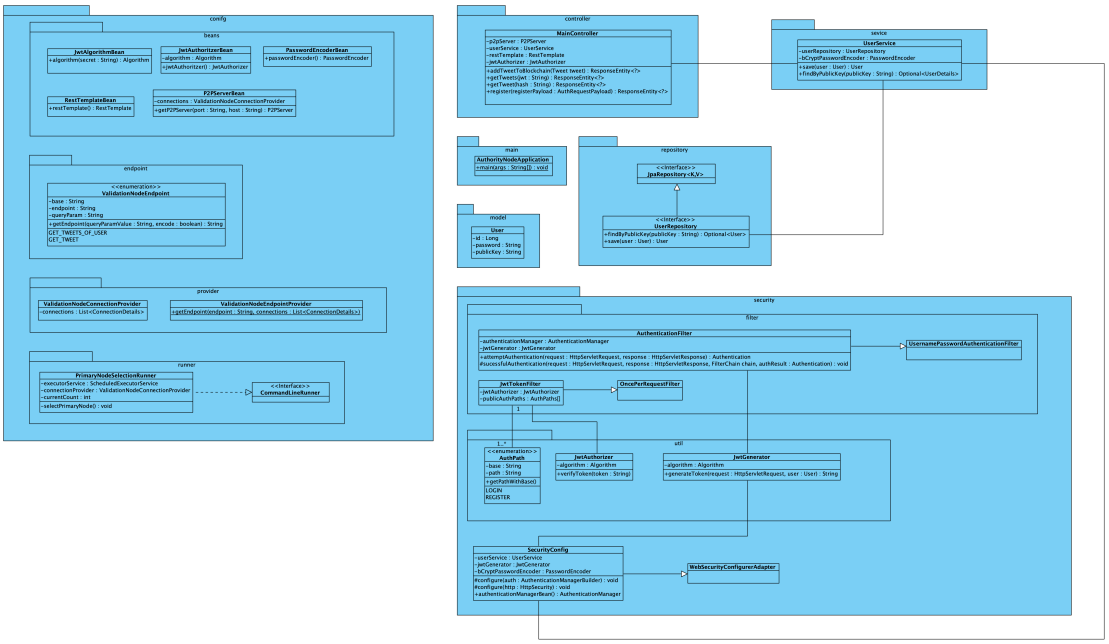
A.1.2. P2PServer



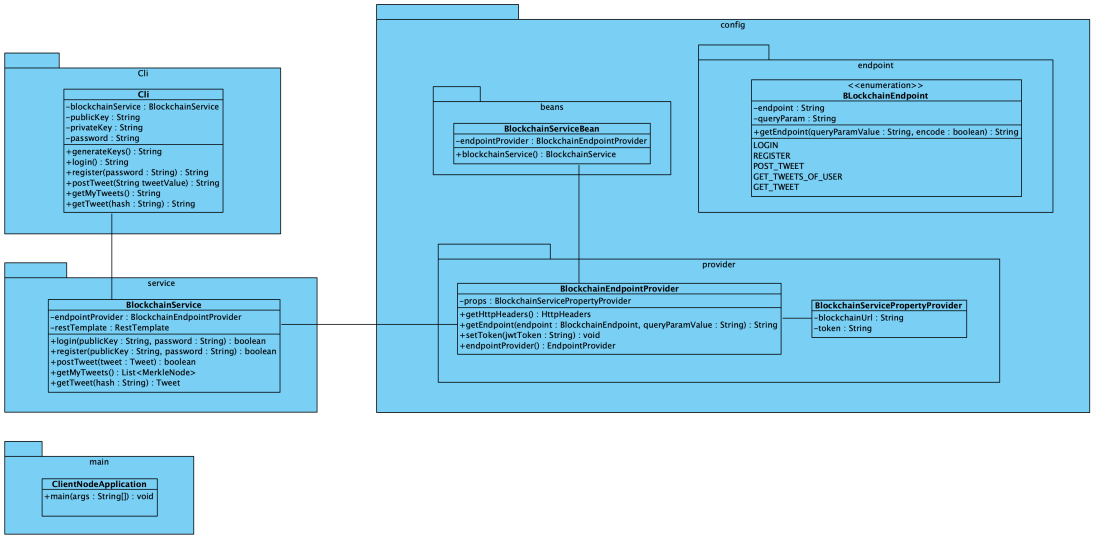
A.1.3. Validation Node



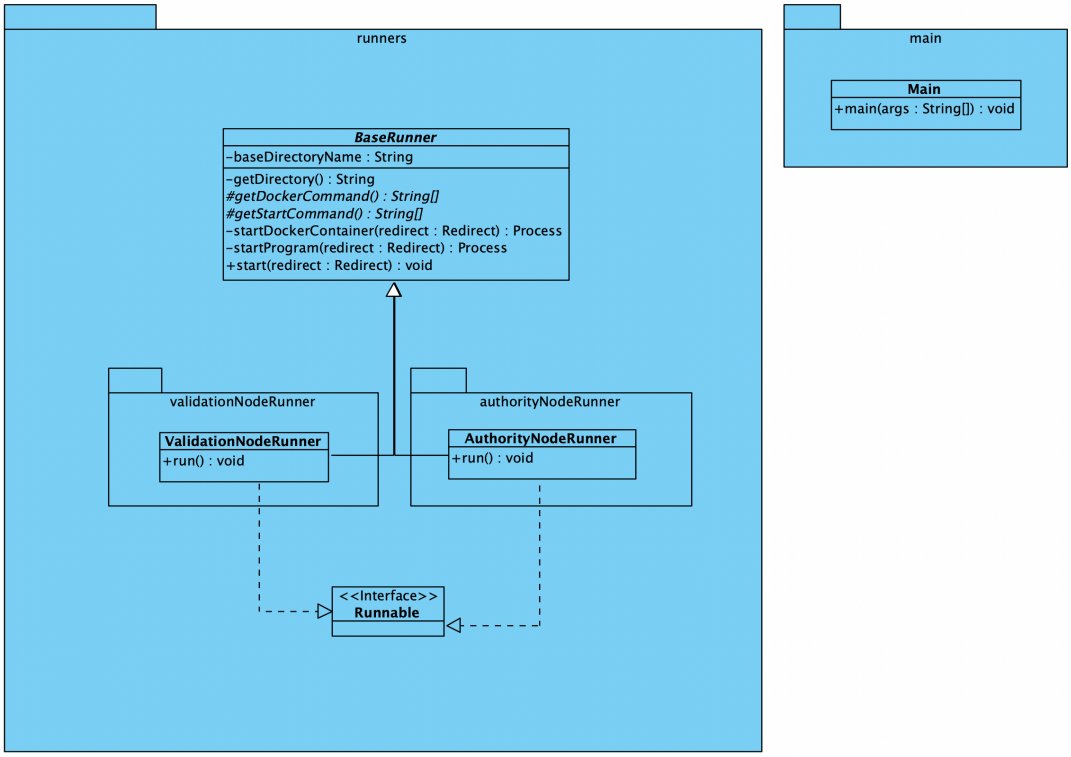
A.1.4. Authority Node



A.1.5. Client Node

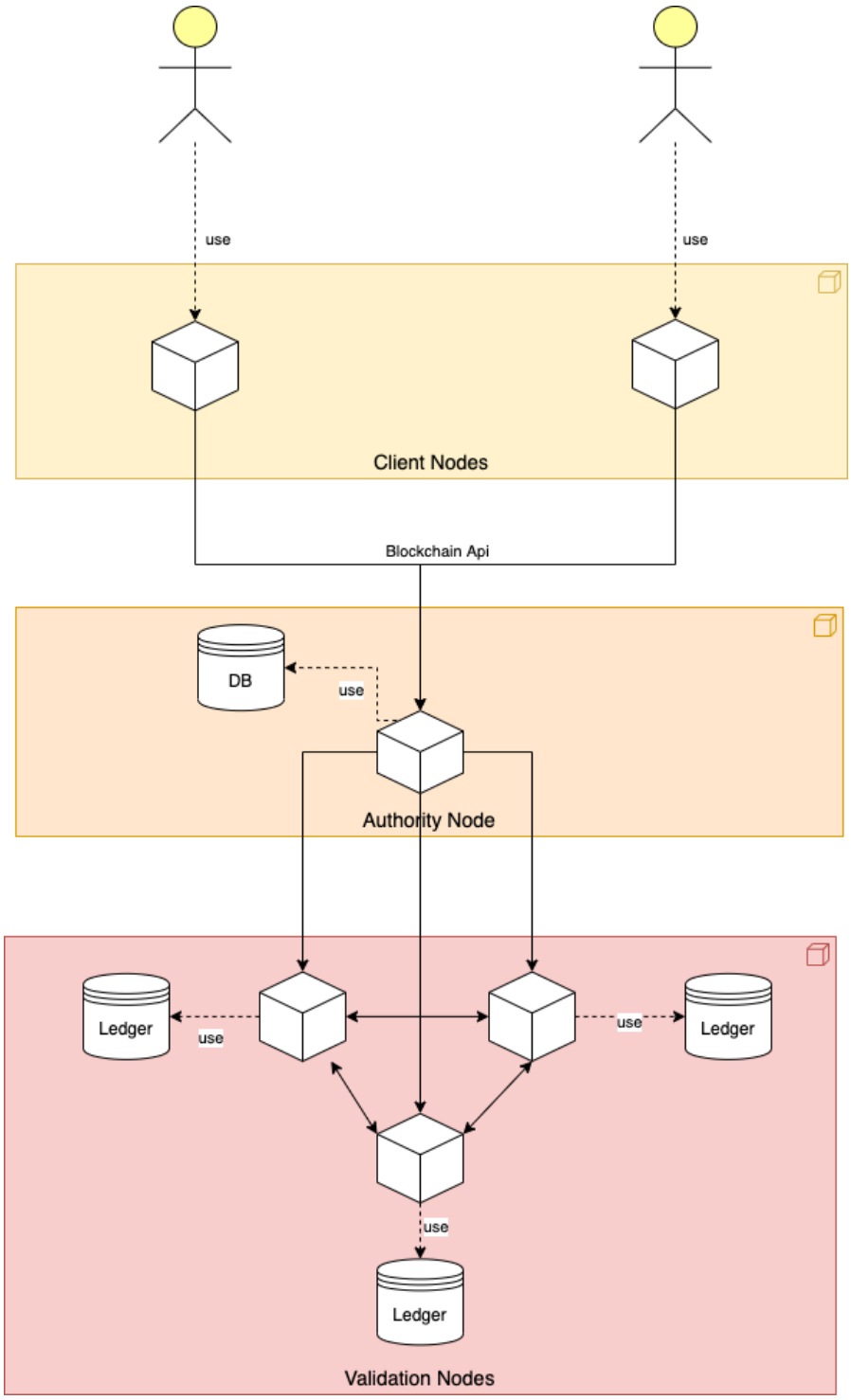


A.1.6. Blockchain Demo



A.2. Demo Application

A.2.1. Architecture



A.2.2. Visual illustration of block mining