MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

BLOCKCHAIN PLATFORM FOR THE MANAGEMENT OF TREE CUTTING

LICENSE THESIS

Graduate: **Erik HALMAI**

Supervisor: **S.I. Dr. Ing. Marcel ANTAL**

2022

Table of contents

Chapter 1. Introduction **1**
1.1. Project context 1
1.2. Current solutions and issues 1
1.3. Proposed solution 1
1.4. Thesis structure 2

Chapter 2. Project Objectives **4**
2.1. Functional requirements 4
2.1.1. Prevention of illegal tree cutting 4
2.1.2. Prove transport illegality 4
2.1.3. Data transparency 5
2.2. Non-functional requirements 5
2.2.1. Performance 5
2.2.2. Usability 5
2.2.3. Portability 5
2.2.4. Security 5
2.2.5. Open source 5
2.2.6. Data integrity 6

Chapter 3. Bibliographic Research **7**
3.1. Blockchain 7
3.2. Cryptocurrencies 10
3.2.1. Bitcoin 11
3.2.2. Ethereum 13
3.3. Web application development 16

Chapter 4. Analysis and Theoretical Foundation **17**
4.1. Solution description 17
4.2. Blockchain used 17
4.3. User description 19
4.4. Use cases 20
4.5. Flow of events 22
4.6. System sequence diagram 25
4.7. Domain model 27

Chapter 5. Detailed Design and Implementation **29**

5.1. Scheme 29
5.2. System architecture 30
5.2.1. React website application 30
5.2.2. Smart contracts 33
5.3. Implementation 35
5.3.1. Server application 35
5.3.2. Client application 43
5.4. Deployment 48

Chapter 6. Testing and Validation 49

Chapter 7. User's manual 53
7.1. System requirements 53
7.2. Deployment 53
7.3. Application walkthrough 53

Chapter 8. Conclusions 56

Bibliography 58

Chapter 1. Introduction

This chapter presents the project context. It presents the current way tree cutting is managed in Romania along with its flaws, and the project's motivation, which is to fix these issues.

1.1. Project context

The context of the project is represented by the management of tree cutting and transportation. This thesis presents a web application, that uses blockchain technology to build a platform for this cause.

The motivation behind the thesis is the present situation in Romania. As described in [1], around 78% of the remaining pristine forests existing in Europe are in Romania. This is changing fast however, due to the amount of illegally cut wood. [2] presents that after a monitoring of 10 years, using world-certified scientific method, a research department from Romania, with the help of the European Union, have found that every year around 20 million cubic meters of wood are cut in Romania without legal forms. Considering an average price of 50 Euros per cubic meter of wood, this results in a value of around 1 billion Euros per year of wood, which is illegally exploited from the forests of Romania.

This is largely due to the fact that the firms which are buying wood, have the right to cut the trees from the forest. By bribing the foresters, the cutting firms transport a lot more wood then agreed upon, and because of the centralized and outdated operation model, proving the illegality is hard. Almost all European countries have given up on this mode of operation a long time ago, and the wood is now bought from warehouses, after they have already been cut and prepared by foresters. Romania should have joined these countries in 2022, but it has been postponed until 2025 at the soonest.

1.2. Current solutions and issues

The system used currently, in Romania, for the management of tree cutting implies two contracts between the forest representatives and the cutting firms. Firstly, foresters search for trees that are dying and they create auctions for the firms to bid on this number of trees. The firm which wins the auction, gets offered a cutting contract, which specifies the number of trees they can cut. Once the firm cuts down the trees and wants to transport them, they are offered a transportation contract, which is valid for 8 hours and it specifies the amount of wood they are transporting on the vehicle.

This outdated system creates the opportunity for illegal tree cutting. Due to the centralized nature of it, by bribing the foresters, the cutting firms are allowed to cut more wood than legally agreed upon. There is no system that checks for the actual number of trees that have been cut and incase of a police control, the contracts can be modified internally. Besides the problem with cutting, since the transportation contract is valid for 8 hours, the same contract can be used multiple times for more than one trip. This makes it possible for the firms to transport the fraudulently cut wood to their warehouses.

1.3. Proposed solution

The solution described in this thesis is an online platform that uses blockchain technology for the management of tree cutting and transportation.

Blockchain technology was popularized with the invention of cryptocurrencies. The most popular of these is Bitcoin, which has released in 2009 written by a person or a group of people called Satoshi Nakamoto. In [3], Satoshi describes Bitcoin as a peer-to-peer electronic cash system, who's main purpose is to allow online payments without the need to go through a financial institution, such as a bank. The implementation of the blockchain data structure along a peer-to-peer architecture, within Bitcoin, made it the first digital currency, that solved the double-spending problem without a trusted authority or a central server. After Bitcoin, many other cryptocurrencies have appeared, and still appear to date. Some of them aim to be a new payment method, some aim to be a store-of-value, like gold is, and some create the possibility for people to create applications that run on the blockchain. Applications on the blockchain operate autonomously, through the use of smart contracts, as described in [4].They offer the benefit of not being owned by any one entity, but rather being decentralized. These applications are often referred as decentralized applications.

The blockchain data structure consists of a growing back-linked list of blocks that contain transactions, as described in [5]. Each block is uniquely identified by a cryptographic hash, and they contain new transactions needed to be validated, as well as the hash of the previous block. The hash of each block is calculated using a cryptographic hashing function applied on the contents of the block. Figure 1.1 shows a visualization of the blockchain data structure. This makes blockchains resistant to modification, as if the data inside a block is modified, the previous hash field inside the next block will point to an incorrect location. If a change needs to be done on the blockchain, the hashes of all subsequent blocks need to be recalculated, which is an expensive operation due to the difficulty level of the blockchain mining. These aspects will be presented in more detail in the next chapters.

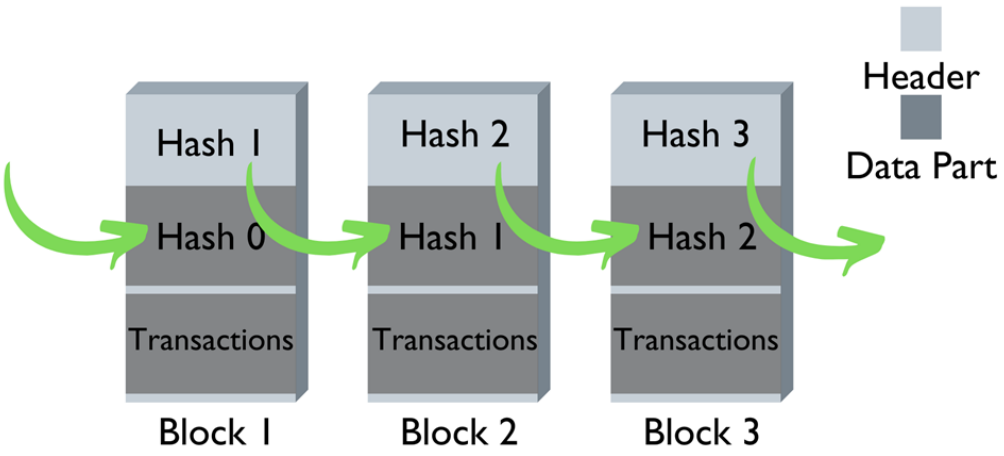


Figure 1.1. Blockchain visualization

This thesis introduces a blockchain-based web platform for the management of tree cutting and transportation. It describes the implementation of a smart contract based decentralized application, whose functionalities can be reached using a website platform. It utilizes both the immutable and decentralized aspects of the blockchain technology.

1.4. Thesis structure

The following paragraph presents the thesis's structure and a brief overview of the contents of each remaining chapter.

- Chapter 1. Introduction: It presents the project's context with the current solutions and issues along the suggested solution that will be described in this thesis.
- Chapter 2. Project Objectives: Presents the objectives this proposed solution aims to solve.
- Chapter 3. Bibliographic Research: It has as an objective the establishment of the references for the project, within the project domain/thematic and the presentation of the knowledge needed to implement this application.
- Chapter 4. Analysis and Theoretical Foundation: Explains the operating principles of the implemented application from a theoretic point of view, without any implementation details.
- Chapter 5. Detailed Design and Implementation: The implementation and the design decisions taken during the development process are presented.
- Chapter 6. Testing and Validation: The obtained results are presented, showing if the project achieved the objectives previously presented.
- Chapter 7. User's Manual: Offers instructions on how to deploy the application and how to use it from the perspective of a user having no technical background.
- Chapter 8. Conclusions: Shows a summary of the achievements and results, and includes the description of the possibilities of improvements/further development.
- Bibliography

Chapter 2. Project Objectives

The main objective of the thesis is the development of a decentralized application for the management of tree cutting and transportation. Such an application would have real life applicability in countries like Romania, where the mode of operation is outdated, and illegal tree cutting is a major problem. In the following sections the functional and non-functional requirements will be discussed.

2.1. Functional requirements

This thesis aims to implement the application using smart contracts, deployed on a distributed ledger, taking advantage of the immutability offered by the blockchain data structure.

The presented application's main objective is to prevent illegal tree cutting in countries like Romania. Both decentralization and immutability play an important role in this aim. Decentralization removes the possibility of a central, controlling entity to do fraudulent activity, while immutability removes the ability of modifying the data already committed on the blockchain. Besides this, the blockchain data structure keeps the log of every transaction ever committed. This benefits the authorities monitoring the tree cutting and transportation activities. To interact with the smart contracts, a web application is used, which invokes the functions defined in the smart contract. The web application

In the following subsections, each functional requirement will be discussed in detail, these being the main objectives the application desires to achieve.

2.1.1. Prevention of illegal tree cutting

Prevention of illegal tree cutting is the most important objective of the application described in this thesis.

It is important to note that the biggest problem of the current mode of operation involving tree cutting in Romania, is bribing. Bribing is difficult to prevent with a database centred, centralized application. The entity in charge has total control of the application and what is or isn't displayed to the users of the application. Fraudulent activities can be hidden from the platform, thus proving it difficult for the authorities to show illegality.

Due to the above reason, a smart contract based decentralized application is suggested in this thesis. Immutability and decentralization make sure that once the data of a cut or a transport is committed, it cannot be changed by any entity.

Furthermore, the application should verify each cut. Before a cut, the cutter should ask for authorization from the application. The application should check whether the cutter has any trees left to cut.

2.1.2. Prove transport illegality

One of the objectives of the applications is to be used by authorities when stopping transport vehicles. The application should provide enough evidence for the authorities to prove if a transport is illegal or not.

It aims to provide every transport, along with the departure date, number of trees transported and company, that has ever happened for the transport vehicle. By using the car's number plate, all the information about the current and past transports should be visible. The

information provided should be enough to convict the cutting company for illegal wood cutting. All this information should also be available to the public.

2.1.3. Data transparency

The application aims to provide a transparent and trustworthy view on all on-chain data available about the cutting firms, transports and cuts. All data should be displayed on the website and publicly available to anyone browsing it. By using a decentralized application as a solution, all transactions done on the application will be publicly available.

This requirement is important, as this way the public can play an important role in the prevention of illegal wood transportation. The public can verify transport vehicles and if illegal activity is detected, the police can be called to investigate further. In the last few years, the problem of illegal tree cutting has gained major significance in Romania. Due to articles like, [6], many civilians have learned about the severity of the problem.

2.2. Non-functional requirements

2.2.1. Performance

This non-functional requirement refers to the overall speed of the application. Due to the fact, that the speed of the web platform is affected by the chosen blockchain's transaction speed, a fast one should be chosen. The application should offer a seamless experience, with load times lower than 2 seconds, even for transactions that save data on the blockchain.

2.2.2. Usability

Usability refers to the quality of a user's experience when interacting with the platform. The web platform should be easy to use, with a short learning curve. The application aims to be open to the public, so any user, with no technical background should be able to check the validity of transports or the data present on or off-chain.

2.2.3. Portability

The application aims to be usable on different operating systems, as well as mobile. When authorities stop a transportation vehicle, they should be able to check the validity of the transport using their mobile phones.

2.2.4. Security

To save new data on the blockchain, a wallet is needed, that is connected to the website. The application should perform the necessary checks if the wallet's address is registered on the application, and if the roles it has is necessary to perform the action.

2.2.5. Open source

Majority of decentralized applications are open source, which means that the source code of the application is freely available to the public for possible modification and redistribution. This non-functional requirement further increases the decentralization aspect of the application and it has many benefits. It creates a strong community around the application, all members bound to improve the application. It also greatly increases security, since more people can detect security flaws.

2.2.6. Data integrity

Data integrity refers to the assurance of data accuracy and consistency. Any unintended changes as result of storage, retrieval or processing operation could mean malicious or fraudulent intent.

This application aims to have trustworthy, accurate and consistent data. This refers especially to the data saved on-chain. Information about the cuts and transports need to be unmodifiable and accurate. This is vital for the success of the application. The authorities need to be able to trust the data and convict felonies based on it.

Chapter 3. Bibliographic Research

In order to develop this application, bibliographic research needed to be done regarding multiple topics. Different technologies and architectures are presented in order to choose the most appropriate one for the implementation. Furthermore, similar projects were studied, to properly understand the domain, difficulties that might come up and design decisions to fix these. All references will be included in the Bibliography chapter of this thesis.

The first discussed topic will be the blockchain data structure, along with the structure and benefits this technology brings. The second topic is cryptocurrencies, which are built upon blockchains, using a peer-to-peer architecture. This thesis will present the two biggest cryptocurrencies, as of the time of writing the thesis, Bitcoin and Ethereum. The next topic discussed will concentrate on web development.

3.1. Blockchain

Being used by cryptocurrencies, blockchain technology has gained a lot of attention in the last 10 years. Even though the technology is still relatively new, and tech-giants, such as Google, IBM or Amazon have only started a few years ago to dedicate departments for studying and developing applications around blockchain, the data structure was first mentioned back in 1982. [7] is the first known proposal for a blockchain protocol. In his dissertation thesis, David Lee Chaum describes the design of a distributed system that can establish trust between mutually suspicious groups. It proposed a vault system, that chained the history of consensus in blocks and immutably time-stamped the chained data. The paper also laid out the details of the code implementation for such a system.

Book [8] discusses the introduction of a Merkle tree into the design of blockchains. A Merkle trees, or binary hash tree is a data structure used for efficiently summarizing and verifying the integrity of large sets of data. In case of blockchains, Merkle trees are used to summarize all the transactions in the block, producing a fingerprint of the entire set of transaction, which provides a very efficient process to verify whether a transaction is included in a block. A Merkle tree is constructed by recursively hashing pairs of nodes until there is only one hash remaining, called the Merkle root. Figure 3.1 displays a Merkle tree with 8 leaves.

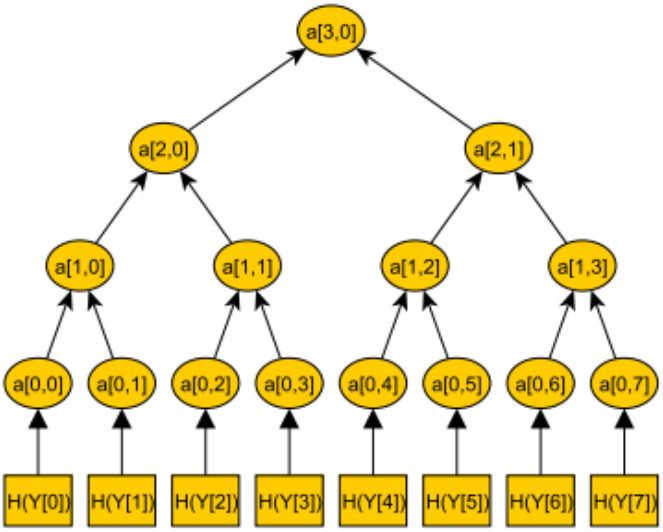


Figure 3.1 Merkle tree with 8 leaves [9]

In case N elements are summarized using a Merkle tree, at most 2 * $\log_2$ (N) calculations are needed at most. This makes this data structure very efficient.

The structure of a block in today's blockchain design differs based on the area it is used in. Even the different cryptocurrencies, that exist, have slight changes between the block structure they use. Most blockchains separate the block into two parts: the block header and the block contents. The block header will be the part that is used to calculate the block's hash. In case of cryptocurrencies, the block contents' part holds the information about each new transaction present in the block. Besides this, there usually is some other metadata, such as the size of the block and a transaction counter, which expresses how many transactions are present inside the block. Table 3.1 shows the block structure of the Bitcoin blockchain.

Table 3.1 Bitcoin block structure [10]

Size	**Field**	**Description**
4 bytes	Block size	Size of the block in bytes
80 bytes	Block header	Fields from block header
1-9 bytes	Transaction counter	Number of transactions
Variable	Transactions	Transactions of the block

The next important aspect to be discussed is the block header. The block header must contain the previous block hash, as this preserves the immutability of the blockchain. Having the previous hash inside the block header assures that it affects the hash of the current block. This way, if any change occurs in the previous node and its hash changes, the hash of the current node will need to be recalculated, and thus all the hashes of the subsequent blocks will need to be recalculated. The Merkle root is also present in the block header, which acts as the digital fingerprint of the transactions inside the block. It is used to efficiently summarize all the transactions inside the block, as described before. Next, the timestamp is specified. This is the approximate creation time of the block and its is specified in seconds from Unix Epoch. The remaining data depends on the algorithm used by the blockchain. If it is a proof-of-work algorithm, the nonce and difficulty target are present. These are used during the mining process, which will be described in more detail in the next section. In Table 3.2 the structure of the Bitcoin block header is described.

Table 3.2 Bitcoin's block header structure [10]

Size	**Field**	**Description**
4 bytes	Version	Version number of software
32 bytes	Previous block hash	Reference to the previous block in the chain
32 bytes	Merkle root	Root hash of the Merkle tree of block's transactions
4 bytes	Timestamp	Approximate block creation time (epoch)
4 bytes	Difficulty target	The proof-of-work algorithm difficulty target for the block
4 bytes	Nonce	Counter used for the proof-of-work algorithm

Image 3.1 shows a representation of the Bitcoin blockchain and the data of stored inside each block.

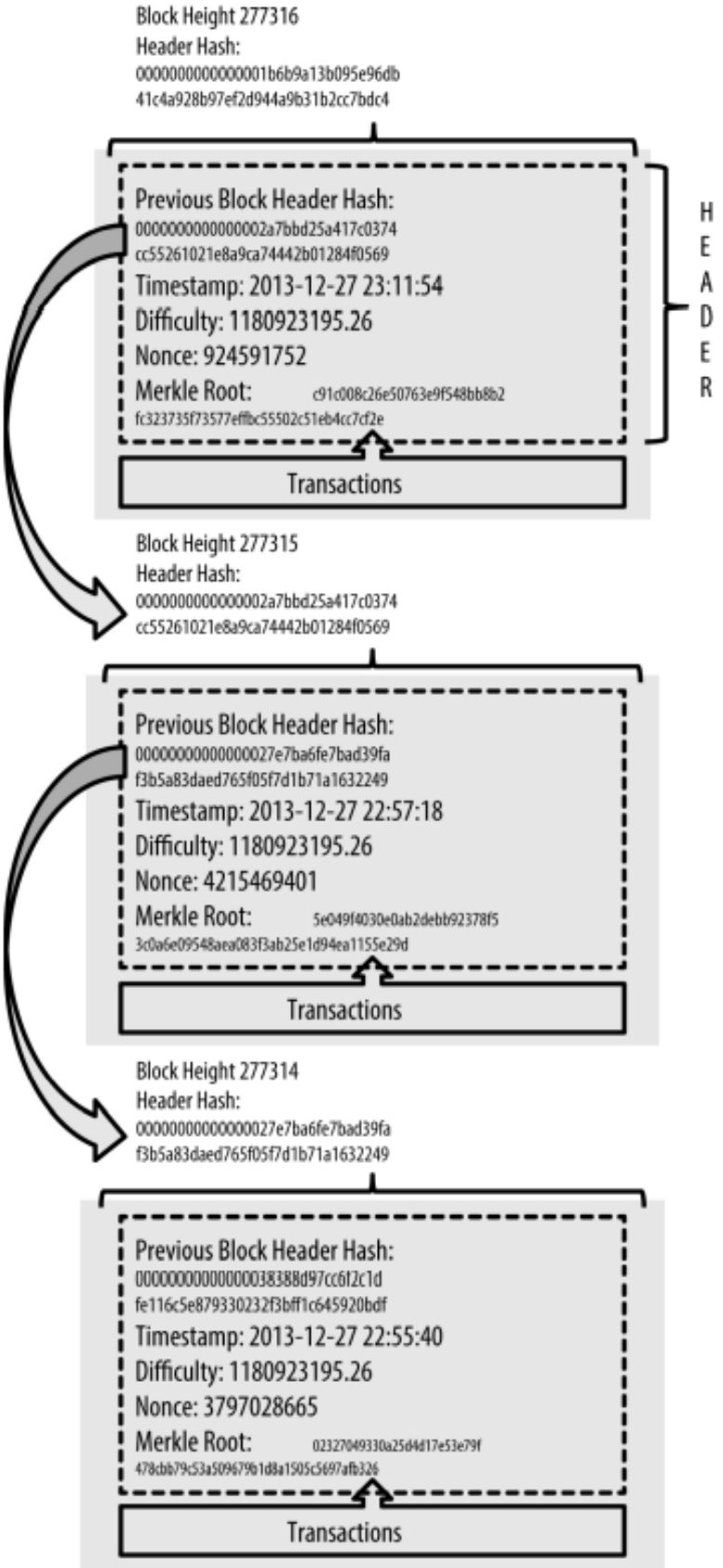


Figure 3.2 Bitcoin blockchain representation [10]

3.2. Cryptocurrencies

Cryptocurrencies have massively gained in popularity in the last years. As described in [11], cryptocurrency is a digital currency designed to work as a medium of exchange though a computer network, that is not reliant on any central authority to validate, uphold and maintain it. The fact that the network can come to a consensus without the need of a central controlling entity makes the transaction speeds much faster than standard bank transfers. Cryptocurrencies can be transacted during the weekend or holidays, as there is no need for a bank to validate the transactions. Due to the rise in popularity, the price of the cryptocurrency market has been growing exponentially in the last years. As of May 2022, there are over 20 thousand cryptocurrencies on the market. Figure 3.3 shows the evolution of the overall cryptocurrency market capitalization per week from July 2010 to May 2022. It can be seen, that at the end of 2021 the market capitalization was worth more than 3 trillion U.S. dollars.

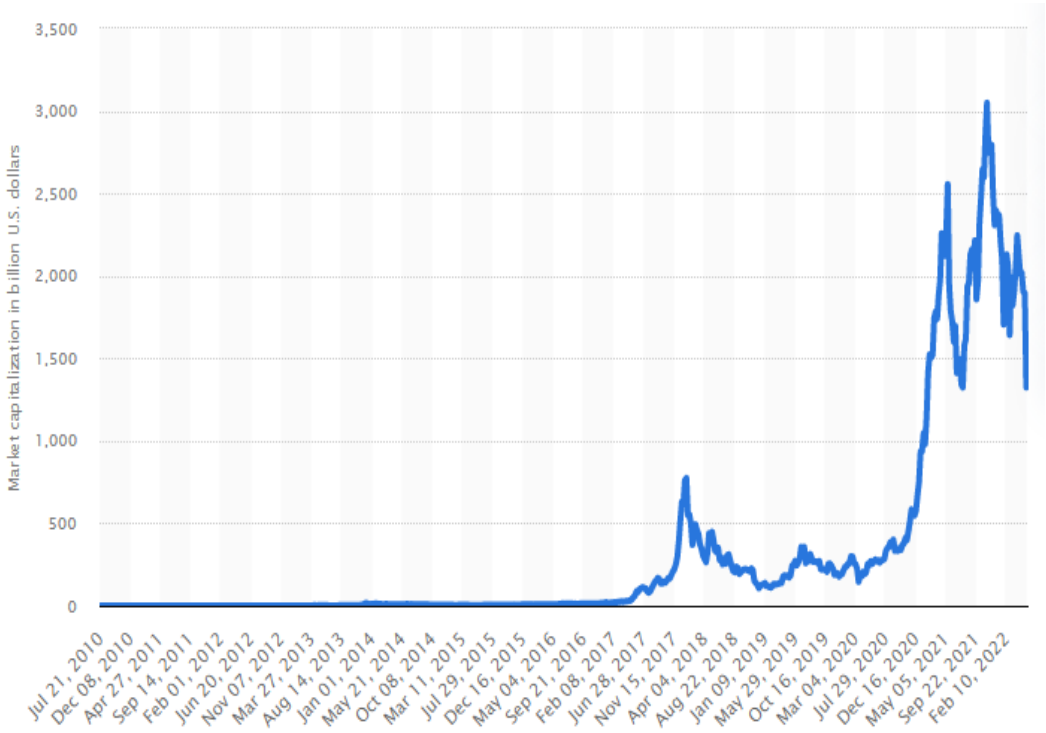


Figure 3.3 Cryptocurrency market capitalization from July 2010 to May 2022 [12]

The issue with digital currencies in the past was the double spending problem. The double spending problem is the issue when a coin can be sent multiple times by a user, without being able to prove that it has been spent already. The aim of digital currencies was to eliminate the controlling entity, and this was first successfully done by Bitcoin. After Bitcoin, multiple new cryptocurrencies appeared and are still being developed to this day. These are called altcoins and they are based on the architecture of Bitcoin, but having different purposes and using different technologies to achieve them.

In the next subsection the technology and architecture of Bitcoin will be discussed, after which, in the following subsection, the second largest cryptocurrency, Ethereum, will be discussed, along with the new use cases it introduces.

3.2.1. Bitcoin

Bitcoin introduced a solution using the above discussed blockchain data structure. It uses the blockchain data structure as a public ledger that records transactions. This ledger contains all transactions that have ever happened.

Bitcoin is structured as a peer-to-peer network architecture. This means that computers, or peers, participate in the network, all of them being equal. Peers, or nodes, in a peer-to-peer network both provide and consumes services at the same time from and to other nodes in the network. This architecture provides the benefit of decentralization, resilience, and openness, this being the reason it is an important choice in the design decisions of Bitcoin. The peer-to-peer architecture, along with all nodes from the network, is often referred to as the Bitcoin Network. This network of communicating nodes, running the bitcoin software, maintains the blockchain, as described in [13]. Almost all nodes hold a copy of the blockchain, based on the roles that will be discussed in the next paragraph.

Nodes in the Bitcoin Network are all equal, but they do differ in the roles they have. This specifies the functionalities they provide to the network. There are four types of functionalities that a node can provide. Furthermore, a node can provide multiple functionalities. The Figure 3.4 shows the structure of a full node, having all four functionalities.

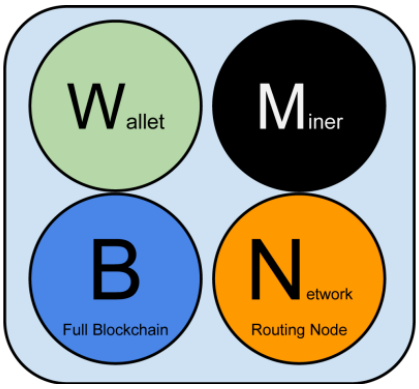


Figure 3.4 Bitcoin network full node [10]

Wallets are containers or simple databases that contain a user's public and private key pairs. These keys are used to sign transactions, thereby proving they own the coins. When a bitcoin wallet is created, its private key is also created. This is a number, which is picked at random. From this private key, by using a one-way cryptographic hash function, a public key is generated. Furthermore, sallets are represented by an address, which is a string of digits and characters that can be shared with anyone trying to send you money. This address is derived from a public key, by using a one-way cryptographic hashing algorithm on the public key. The Figure 3.5 shows the process of deriving each key and the wallet address.

Full blockchain nodes are nodes having the full blockchain. Some nodes can have part of the blockchain downloaded, thus not being fully trustable. In that case, routing nodes can route them to a full blockchain node.

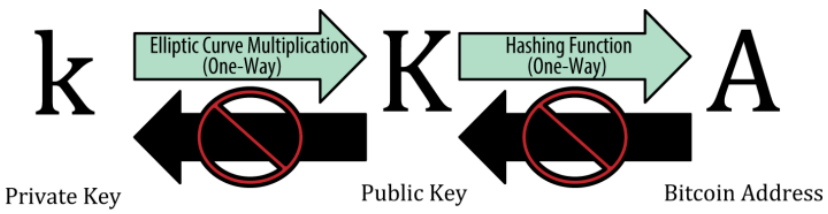


Figure 3.5 Private key, public key, and bitcoin address [10]

Transactions, simply put, are value transfers from one entity to another. They are at the most important part of the Bitcoin system. Everything else is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transaction, or the blockchain. At a high level, the most important two properties of a transaction are the inputs and the outputs. The fundamental building blocks for these two are Unspent Transaction Outputs, or UTXOs. In simple terms, you need to have received bitcoins first to be able to spend any. In effect, there is no such thing as a stored balance of a bitcoin address, there are only scattered UTXOs, recorded on the blockchain, that are locked to a specific owner. The concept of a user's bitcoin balance is derived by the wallet application described above. It calculates the user's balance by scanning the blockchain and aggregating all UTXOs belonging to that user. Thus, as described in [14], the inputs of a transaction are unspent UTXOs belonging to a user, and outputs are newly created UTXOs, that have a destination address, and that will be available for the user of that address to spend in a future transaction. Transactions also result in a fee, which is referred to as the transaction fee or miner fee. It is calculated as the sum of all outputs subtracted from the sum of all inputs. This works by also taking into account the change outputs, that put the remaining bitcoin from a not fully spent UTXO, back to the wallet of the user. Choosing higher transaction fees also makes the processing of the transaction faster. Figure 3.6 shows the effects of a transaction on the UTXOs of a wallet address.

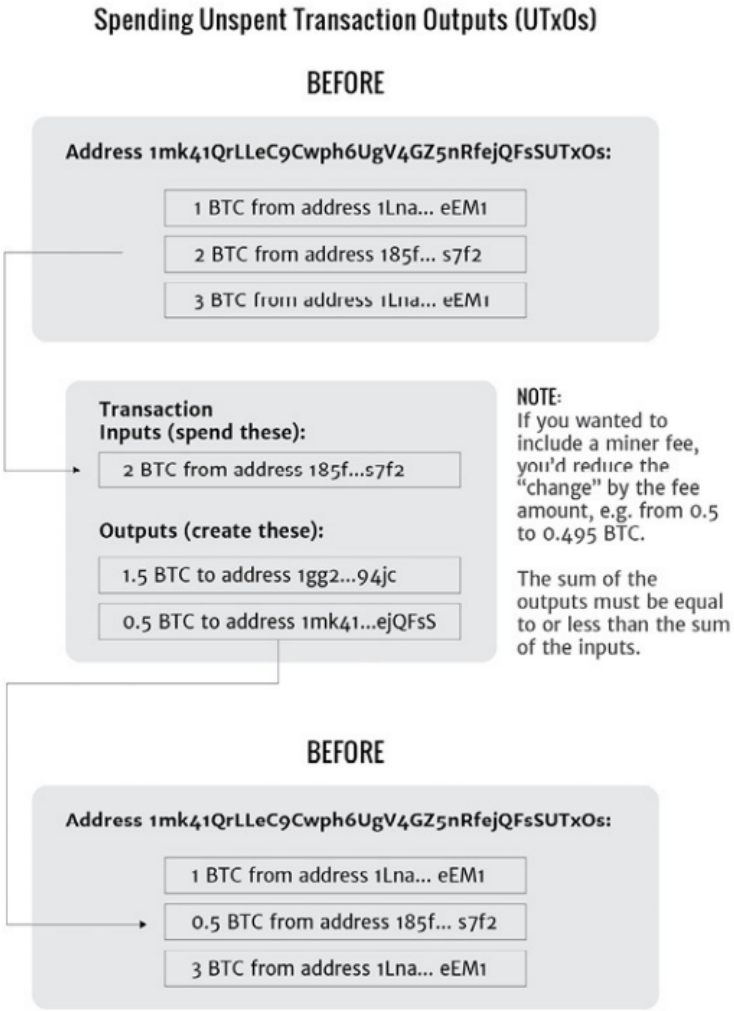


Figure 3.6 Spending UTXOs [15]

Miner nodes validate new transactions and record them on the global ledger. They check the validity of each transaction and add them to a block. A new block, containing transactions that occurred after the last block, is mined every 10 minutes, thereby adding those transactions to the blockchain. All of this is not done for free, however. Miners receive two kinds of rewards for the work they do. Firstly, for the creation of a new block, miners receive an amount of bitcoin. As of the time of writing, this amount is 6.25 bitcoin. This amount decreases approximately every four years. Secondly, miners receive the transactions fees from each transaction in the newly mined block.

The mining process is actually a race between all the mining nodes. Most mining nodes have a full copy of the blockchain, thus being able to validate transactions. When a new transaction comes to a mining node, it validates it, adds it to a new block and propagates it to other nodes. Once there are enough transactions, the block transforms into a candidate block and the demanding work begins. The blockchain has a defined difficulty at each time. This difficulty is a number, which tells how many zeros need to be at the beginning of the new block's hash. This is the problem that the miners try to solve, by having a nonce, or a counter, in the block's header, they can modify its hash by incrementing this. Miners repeatedly increment the nonce and calculate the hash of the block, until one of the miners finds a hash that respects the constraint of the difficulty level. The hashing function used for Bitcoin is SHA256. The difficulty is chosen as to make the mining process last around 10 minutes. After a solution is found, the winning miner gets the rewards and propagates the block to the other nodes, thus adding all its transactions to the distributed ledger. The other nodes verify the newly created block and propagate it further.

This processes, of finding a solution to the SHA256 hash in order to prevent false or malicious transactions from being added to the blockchain, is called proof-of-work. Proof-of-work is the way in which Satoshi Nakamoto has invented a solution to a previously unsolved distributed computing problem, called Byzantine Generals' Problem. As [16] describes, the problem consists of trying to agree on a course of action by exchanging information over an unreliable and potentially compromised network. This solution represents a breakthrough in the distributed computing science and has a wide applicability beyond the Bitcoin currency.

3.2.2. Ethereum

Ethereum was initially conceived in 2014 by Vitalik Buterin in its introductory whitepaper, [17]. Following this, it was released in 2015 by Vitalik and a group of founders consisting of Charles Hoskinson, Mihai Alisie, Amir Chetrit and Gavin Wood. Some of these founders have since moved to create their own blockchain projects, such as Charles Hoskinson, who founded Cardano and Gavin Wood, who founded Polkadot.

As described in [18], Ethereum is a general purpose blockchain, which has a peer-to-peer network connecting participants, uses proof-of-work to achieve Byzantine fault-tolerance and cryptographic primitives such as digital signatures and hashes and has a currency, which is called ether. The architecture is similar to Bitcoin's and many common components can be observed. Yet, the two blockchains are strikingly different in their purpose. While Bitcoin's aim is to be a digital currency, Ethereum is designed to be a programmable blockchain. Bitcoin's programmability is limited to its scripting language, which only allows true/false evaluation of spending conditions. This is not enough for most programs, and thus Ethereum came into play, offering its blockchain technology as a tool of distributed consensus for building new application on top of it, as described in the whitepaper [17]. Ethereum offers a built-in fully fledged Turing-complete programming language, that can be used to create "contracts", allowing users to create applications, simply by writing the logic in a few lines of code. As described in [19], Turing-complete programming languages have the capacity to

describe unbounded computations over unbounded values. In simple terms, this means that in principle, the programming language can be used to solve any computational problem.

The way in which Ethereum makes this possible is by having two types of accounts: externally owned accounts (EOAs) and contract accounts. EOAs are similar to the accounts in Bitcoin, having public and private key pairs and being controlled by users through a wallet application. Contract accounts are controlled by program code, or commonly referred to as smart contracts. These are executed by the Ethereum Virtual Machine and control themselves in the predetermined way prescribed by their smart contract code. The code inside the smart contract is executed whenever a transaction initiates them. Smart contracts are immutable computer programs, that run deterministically in the context of an Ethereum Virtual Machine. In the following, these terms will be explained:

- Immutable: once deployed, the code of a smart contract can only be changed by deploying a new instance
- Deterministic: the outcome of the execution of a smart contract is the same for everyone who runs it
- EVM context: smart contracts have a limited context, being able to only access their own state, the context of the transaction that called them and information about the most recent block

Multiple programming languages have been developed for writing smart contracts on Ethereum, but all of which need to be compiled to low-level byte code that can run on the EVM. Two of the most popular ones are Solidity and Vyper. Solidity is similar to programming languages such as Java and C, while Vyper was built to be as similar to Python as possible.

As previously mentioned, Ethereum also has its own currency, called ether. As Ethereum's purpose is not primarily to be a digital currency payment network, its currency's purpose is not similar to Bitcoin's. Ether can be transacted freely, just like bitcoin, but it is intended as a utility currency that is used to pay for use of the Ethereum platform. As described in [20], currently, after a network upgrade entitled London upgrade, each block has a base fee, which is the minimum price per unit of gas for inclusion in the block. This fee will be burned or destroyed after the transaction ends. Whenever a transaction comes, this base fee must be paid along with a tip, or a priority fee, which compensates the miners. This priority fee is somewhat similar to the transaction fee described in Bitcoin's architecture. The transaction fee also has a gas unit, or gas limit, that the user can choose, depending on how much they are willing to spend on the transaction.

One of the most important uses of smart contracts are decentralized applications. As described before, such applications do not rely on a central server, and they run on the blockchain. There are many advantages to creating a decentralized application, that a typical centralized architecture cannot provide:

- Resiliency: the business logic of the application is controlled by a smart contract, making it fully distributed and available as long as the blockchain is available
- Transparency: everyone can inspect the code of the smart contract and any interaction with it will be stored forever on the blockchain
- Censorship resistance: the code cannot be altered after deployment and any user can interact with it as long as they have access to an Ethereum node

Due to these benefits, many organizations and developers have turned to developing their applications using smart contracts, on the blockchain. The application described in this thesis is in the domain of asset tracking and distributed governance, so similar applications were researched.

In [21], the implementation of an asset tracking application is described, using smart contracts and IoT devices. The benefits of smart contracts in supply chain applications are

described. The application consists of a GPS module on the delivered good, an online platform that receives the GPS module updates and the smart contracts on Ethereum, which have the criteria defined for the delivered good's location. The smart contract checks for any violation between the current location of the good and the destination. The courier also receives an automatic reward, consisting in ether, whenever the delivered good arrives to the destination.

[22] discusses the possible benefits of smart contract implementation into the Indian banking system. It discusses the use of smart contracts to create cryptographically signed immutable loan records. These loan records would be transparent and available across all banks, thus sharing information between them. It would benefit the system by providing faster identification of risky customers having high unpaid debt. Figure 3.7 describes the use of smart contracts for interest rate calculation.

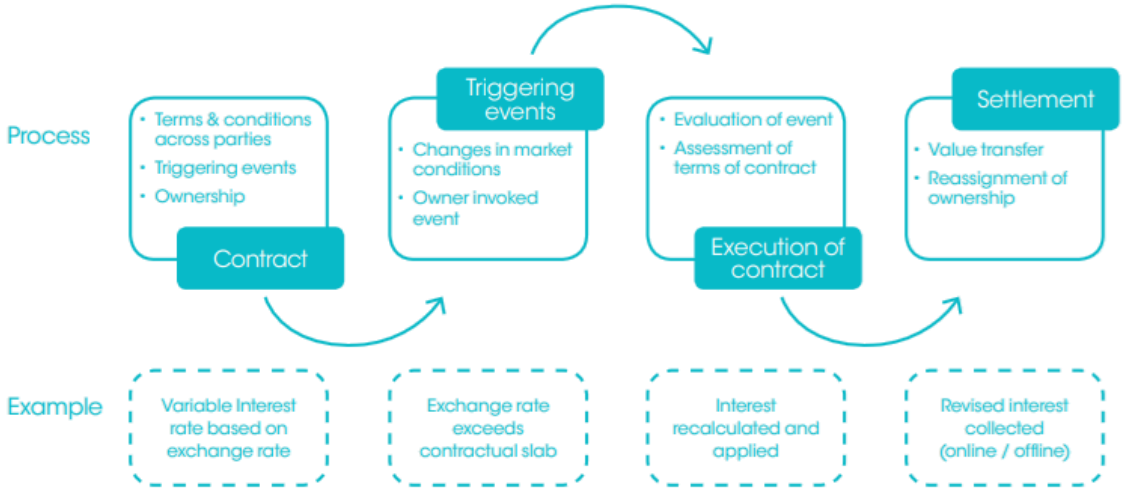


Figure 3.7 Interest rate calculation using smart contracts [22]

Lastly, [23] is a similar project to the one described in this thesis. It describes a blockchain platform for the COVID-19 vaccine management. The solution is implemented using smart contracts for a system that offers transparent tracing of COVID-19 vaccine registration, storage, delivery and side-effects self-reporting, shown in Figure 3.8.

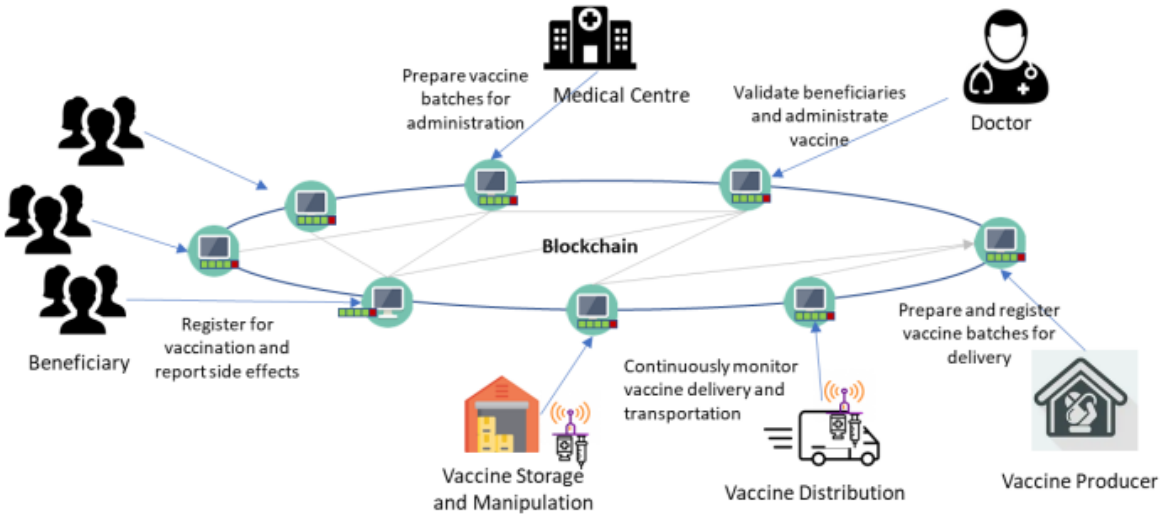


Figure 3.8 Blockchain based application for COVID-19 vaccine management [23]

The benefits of using smart contract based implementation, are described, for an application where transparency and immutability are important. It ensures privacy for beneficiary registration by using Merkle Proof, the root node storing the hash of the secret key generated by the beneficiary and the hash of their Personal Identification Number. It also presents a solution for the registration of actors on a smart contract. It uses Solidity's mapping structure to keep track of registered actors and their types. This leads to smaller execution time and cost of the blockchain transactions mined.

3.3. Web application development

In order to interact with the smart contracts deployed on the blockchain, a user-friendly platform is needed. Most decentralized applications use website platforms, in the aim to have it available to as many users as possible. As this application should be available and usable by the public, such a web platform is chosen.

Decentralized applications are based on the client-server architecture, as described in [18], which is often used in web application. The components of the client-server architecture are presented in [24]. Clients provide an interface to allow users to request services from the server and to display the returned results. Servers wait for requests to arrive from clients and then respond to them. In case of decentralized applications, the smart contracts deployed to the distributed ledger, contain the business logic of the application, thus acting as the server for the application.

For the client side, multiple technologies have been researched. A JavaScript based technology was chosen for the building of the user interface, due to the sheer amount of packages available by using the Node package manager. Using the npm CLI makes it easy to install and organize the packages used by your application. To easily interact with the smart contracts, web3.js was used, which is a collection of libraries that allow developers to interact with remote Ethereum nodes using HTTP, IPC or WebSocket, as described on their official documentation, [25]. React was researched, which is a JavaScript framework offering the possibility to create UI components having their own state and lifecycle methods. Components are visual elements on the screen, each component being able to contain multiple components, and by using states, they can be separately refreshed based on the state they are in.

Chapter 4. Analysis and Theoretical Foundation

This chapter explains the operating principles of the implemented application. It will discuss the solution from a theory standpoint, presenting logical explanations and arguments concerning the chosen solution, logic and functional structure of the application, used algorithms and the abstract model.

4.1. Solution description

This thesis proposes a solution to the current problem of illegal wood cutting in countries like Romania. The solution is composed of a decentralized application, using a web platform for accessibility.

The application aims to offer the ability for foresters to create cutting and transportation contracts. The cutting contracts need to be associated to a registered cutter firm, which can be uniquely identified by using the firm's Taxpayer Identification Number (TIN). Cutting contracts have a number of trees associated with them, which determines how many trees is the firm allowed to cut using the cutting contract. On the other hand, transportation contracts are associated to cutting contracts, each cutting contract having multiple transports, in order to move the cut trees to the firm's warehouses. Transport contract also contain the number of trees that are being transported, the number plate of the transportation vehicle and the departure time. This way, vehicles can be checked, when seen in public, if they have any recent transportation contracts associated with them. All of this information needs to be publicly available for anyone, and viewable on the application's website.

In order to identify the cutting and transportation contracts effectively and to decrease gas costs, the contracts are uniquely identified by a hash. This hash is calculated when saving the contract, thus creating a digital fingerprint for the contract. This also decreases gas cost, as there is no need to save all the information about the contract at each place where it is used, but rather this hash, or id can be used, which can be also be used to access all the information about the contracts. To make sure these hashes are unique, timestamps are used in the contracts. For the cutting contract, the start time is saved, while for the transportation contract, the departure time. This makes sure the calculated hash will always be different.

Using this hash for the contracts, makes it possible to create QR Codes that can encapsulate the contracts. This QR Code can be used to access the website which contains the information about the contract.

The registration of cutting firms and the creation of cutting and transportation contracts can only be done by foresters. The application, being a decentralized application, does not have any username and password login, but rather wallet connection, using cryptocurrency wallet software, such as MetaMask. Thus, the checks for foresters and cutters are done using the addresses of the wallets. The application also needs to make sure cutters are not allowed to cut more trees than agreed upon in the contract, or transport more trees than have been cut for a cutting contract. To make this possible, the cutters need to make a request to the application, before cutting the tree, to check if they are allowed to cut anymore.

4.2. Blockchain used

For the development of a decentralized application, a blockchain is needed, which has smart contract capability, to run the business logic of the application. In the context of this application, a public blockchain is preferred, to make the transaction made to the smart

contracts publicly available to anyone who wants to check them. The Ethereum blockchain was chosen for this project, which is the second biggest cryptocurrency as of the time of writing. Ethereum is also one of the oldest cryptocurrencies, being released in 2015, and the most used cryptocurrency in terms of development purposes. This makes it the most viable option for the blockchain.

Ethereum offers fast transaction speeds with affordable transaction fees. These are described in regard to Bitcoin's stats in Table 4.1. However, Ethereum's stats will be upgraded following the release of Ethereum 2.0. This upgrade is planned to release in the near future, some test networks already having parts of it deployed, and it aims to move the network from the previously discussed proof-of-work model to proof-of-stake. In proof-of-stake, the consensus is not dependent on miners, but rather owners of the cryptocurrency can stake their coins, which than gives them the right to check new blocks of transactions and add them to the blockchain in exchange for some rewards. This would greatly decrease the transaction costs of the network and introduce the possibility of sharding, which splits the blockchain into smaller chains, known as 'shards', thereby allowing distribution of data across the network and a great increase in transaction speeds.

Table 4.1 Bitcoin vs Ethereum transaction speeds and fees [26]

	Bitcoin	**Ethereum**
Transactions per second (TPS)	3+ TPS	12+ TPS
Average transaction fee	$2.99	$2.89
Transaction confirmation	10-60 minutes	10-20 seconds

Besides this, Ethereum being the most developed on blockchain, it has the largest collection of libraries and smart contract programming languages available of all other blockchains. Development on Ethereum is easy to get started with and the documentation from previous applications come in handy when implementing the decentralized application from this thesis. In Figure 4.1, which shows the monthly active developers for some of the biggest blockchains, it can be seen that Ethereum has the largest number of active developers, around 4000 monthly, and it is ahead by a big margin from the second blockchain, Solana.

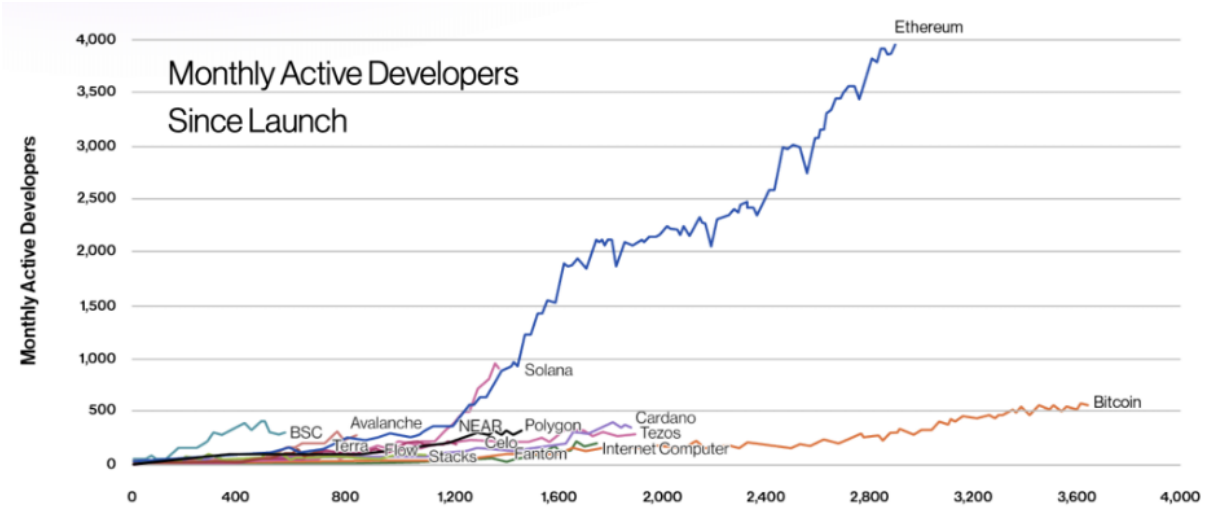


Figure 4.1 Monthly active developer since launch on blockchains [27]

4.3. User description

This application aims to be available to anyone who accesses the web platform. One of the most important aims of this application are to be transparent and trustworthy. This is important, as it aims to eliminate the possibility of illegally cutting trees by bribing.

The platform aims to have two major types of users. Firstly, users having their MetaMask wallet connected to the platform. Anyone can connect their wallet, but the application checks if the address of the account is a registered forester or a cutter address. If they are, these types of addresses can add data onto the blockchain. Users can also interact and save data by interacting directly with the smart contract, without using the website platform, thus the address needs to be checked on smart contract level as well. Forester are able to create new cutting contracts, transport contracts, register new foresters or new cutter firms. They have the most privileges to the application and they are responsible to maintain the platform. On the other hand, users being connected to a wallet that has a register cutter address, can request to cut a tree for contracts that they own. As discussed before, the application checks if the cutting contract still has trees that can be cut. Secondly, users who do not have their wallets connected, or who have a unregistered wallet address. These users are only able to view the data of the application. All data is public and displayed on the website. This data is also available on the Ethereum blockchain, thus users are able to view it without the website platform. These users can be authorities, environmental organizations or anyone who wants to check the validity of transport or cuts. By implicating the public and having the data available to everyone, illegal activity can be observed more easily and signaled to the responsible foresters or police. Table 4.2 presents the types of users and their responsibilities.

Table 4.2 Users description

Name	Description	Responsibilities
Registered wallet connected user	User having their Ethereum wallet connected to the platform, which has its address registered as a forester or a cutter.	Update the data on the blockchain.
Forester	User with highest rights. Manages the platform.	Organizes auctions and prepares the contracts with the cutting firms. Adds the contract to the blockchain. Monitors the cutting process. Creates transport contracts for the cut trees of a cutting contract.
Cutter firm	Performs the wood cutting. Has limited rights.	Request the cutting of a tree for a cutting contract. If there are still trees left to be cut, the transaction will succeed and the cutter can proceed with the cutting the tree.

Supervisor	User not having their Ethereum wallet connected, or having a wallet with an unregistered address connected.	Checks the data on the blockchain to see if there was any illegal wood cutting or transportation.
Policeman	User trying to catch illegal wood cutting.	Checks the contract status and if the wood on a transport vehicle is justified or not.
Environmental organization	User interested in preventing excess and illegal tree cutting.	Monitors and signals any irregular cutting or transportation activities.
General public	User checking the transport on a vehicle or the tree cutting in their country.	Any user can visit the platform and have the data on the blockchain available.

Figure 4.2 present a UML diagram showing the grouping of these users based on having a registered wallet connected or not.

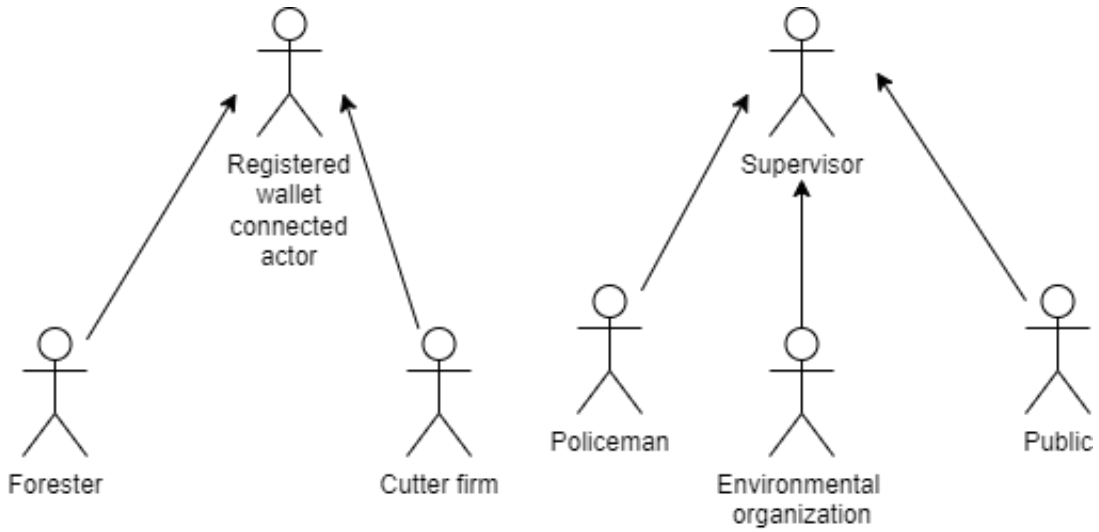


Figure 4.2 User diagram

4.4. Use cases

To satisfy the application's requirements, the use cases need to be established. These are used in the planning and development and testing phases to ensure the application functions as needed. Figure 4.3 present the use cases diagram of the application.

The use cases are divided based on the roles each user has. Foresters have the highest privilege, being a trusted authority, that is supposed to manage the forest and the application's data. Even if this trust is broken and the forester tries to cover illegal cutting by creating cutting contracts, by having the application's data open to the public, and all data available, this illegal activity can be discovered by supervisors.

As a precondition, any user trying to add data onto the blockchain, need to have an Ethereum wallet connected to the platform. After this, the application does some tests in order to check if the user is authorized to perform the operations. Foresters are allowed to register

new foresters and register new cutter firms, by specifying their wallet address, and the Taxpayer Identification Number (TIN), name and phone for the cutter firms. They are also allowed to create cutting and transportation contracts. Before creating the cutting contract, the system checks if it is created for a registered cutter and saves it to the blockchain. In addition, for the creation of the transportation contract, the system checks if it is being created for an active cutting contract and makes sure a valid cutting contract is used, since transportation contracts are linked to cutting contracts.

Cutters are allowed to request cuts, but just for cutting contracts that are registered to their cutting firm. Thus, the system checks if the used address is the one registered for the cutting firm linked to the contract. The cutter gets the feedback if the tree can be cut or if there are no more trees left for the contract.

Supervisors represent any user that accesses the platform. They are allowed to monitor the data of the application in case any illegality is observed.

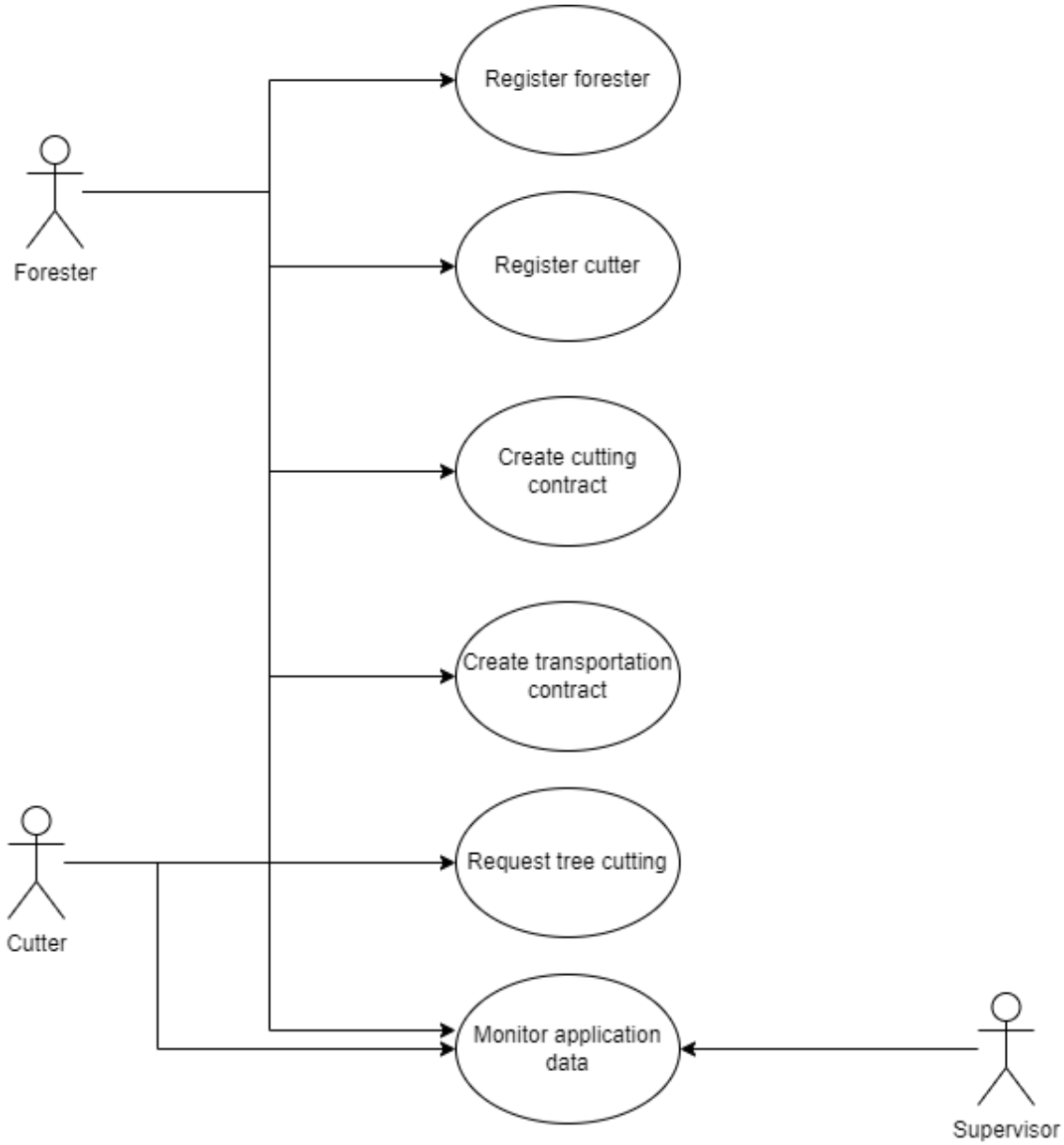


Figure 4.3 Use case diagram

4.5. Flow of events

The flow of events showcases the events that the system goes through in order to complete the use cases mentioned above. These events represent steps taken and verifications that it does to preserve the integrity of the application. This results in basic flow for each diagram and alternative flows. The basic flow represents the steps for the successfully completed use case, while the alternative flows represent deviations from this flow, in this application's case.

The system always checks if the transaction sender's address is of a registered entity and if it has the necessary role, for any operation that saves data on the blockchain. After this, as presented in Figure 4.4, in case of the cutting company registration, the application also checks to make sure the company is not registered already. If all these checks succeed, the transaction succeeds and the registration is complete, being saved on the blockchain. Else, alternative flows will trigger, which all result in failed transactions. When a transaction fails for a smart contract transaction, all changes done will be reverted, thus not changing the state of the blockchain data.

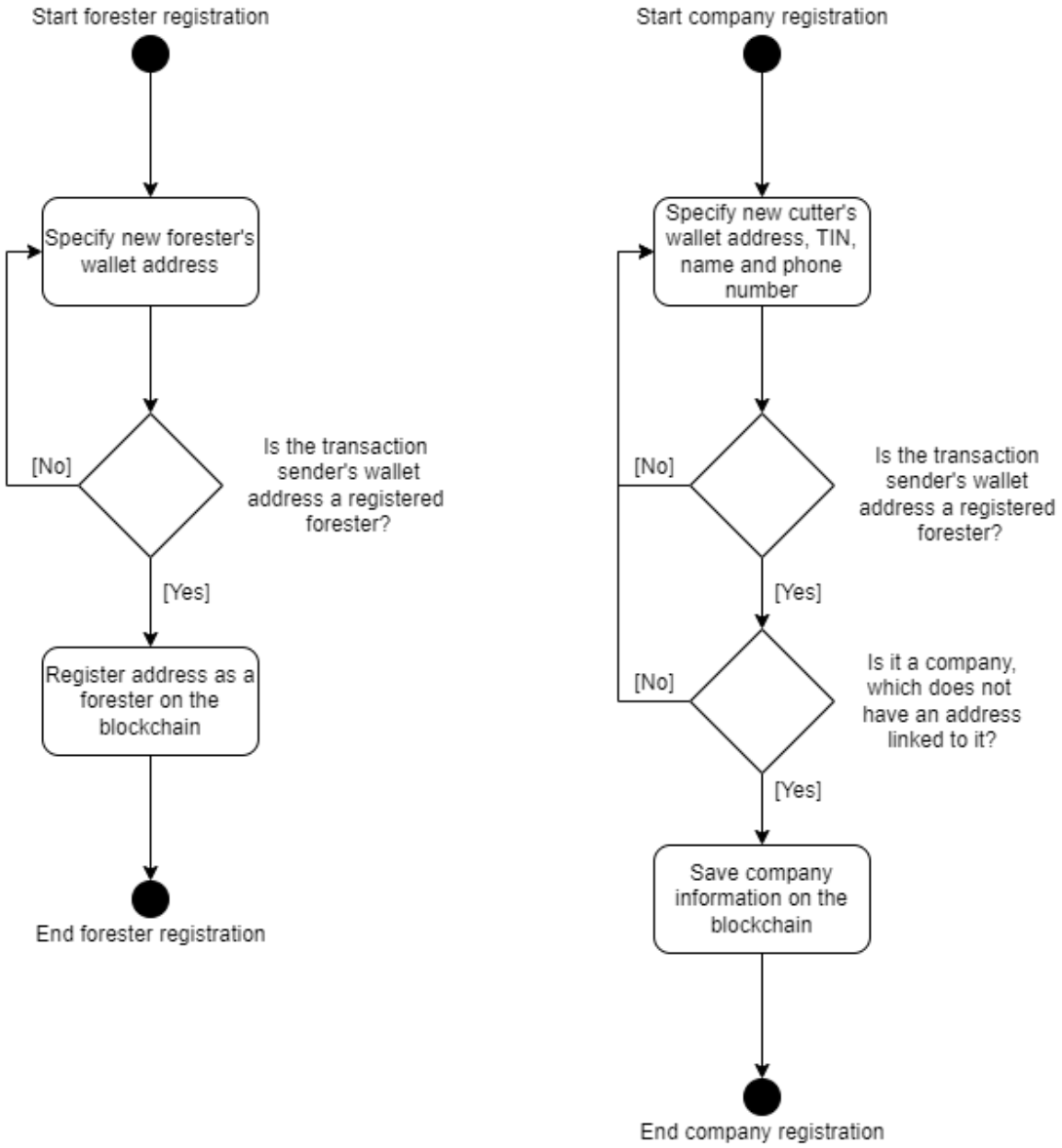


Figure 4.4 Flow of events for registration use cases

Additionally, in the case of the contract creation use cases, the system does some additional checks before the creation of the contract, and finally returns the new contract's hash code, along with the generated QR Code. Figure 4.5 showcases the flow chart for the cutting and transportation contract creation use cases.

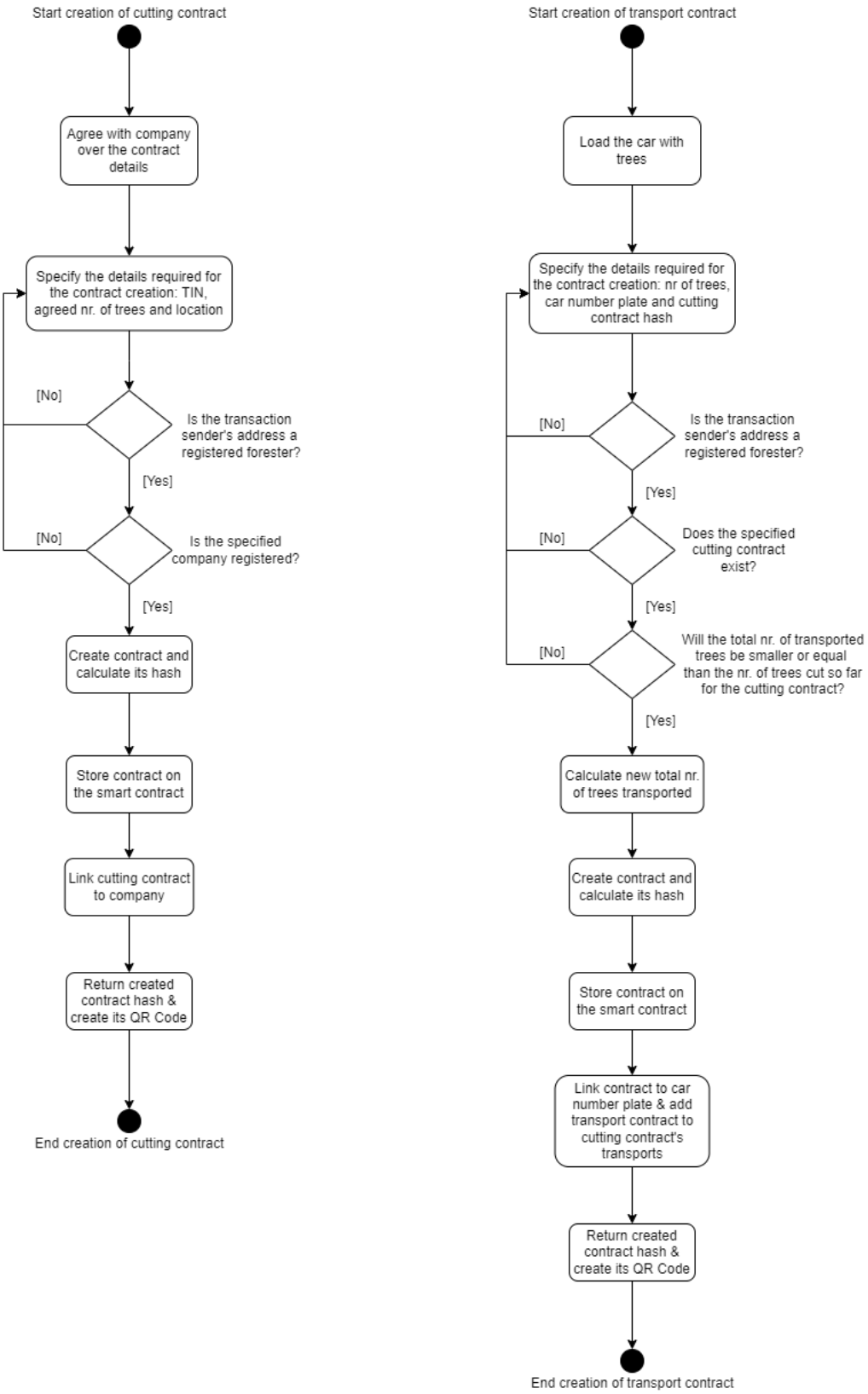


Figure 4.5 Flow of events for contract creation

The basic flow for both contract creation use cases are composed of specifying the needed data and information about the contract, doing the necessary checks to make sure the entered data is correct, saving the data on the blockchain and creating the necessary links between the data and finally returning the new contract's hash and the generated QR Code. In case of the cutting contract creation, the TIN of the company must be specified, the agreed number of trees that can be cut and the location where the cut can happen. After specifying the data, the system checks if the transaction sender's address is a registered forester or not, and if the company is already registered or not. In case any of these checks are invalid, the transaction will revert and an alternative flow will trigger, bringing the user back to correct the entered data. If the entered data is correct, the new contract is created and its hash is calculated, the cutting contract is linked to the company and its hash is returned, along with the generated QR code containing a link to the webpage where the contract details can be checked. In case of the transportation contract creation, it is initiated by first loading the transportation vehicle with the trees and specifying the number of trees that have been loaded, the vehicle number plate and the hash of the associated cutting contract. The system checks if the transaction sender's address is a registered forester, as before, if the specified cutting contract exists and if the total number of trees transported is smaller or equal to the number of trees that have been cut for the specified cutting contract. The last check makes sure that the cutters cannot create transportation contracts without having registered cut trees, thus preventing illegal tree transportation. As before, in case any of the checks fail, the transaction will be reverted and the user will need to correct the data entered. If this is not the case, and the data is valid, the system calculates the new number of transported trees for the cutting contract, creates the new transport contract and its hash and stores it onto the blockchain. The necessary links are created between the new transport contract and the transportation vehicle's number plate and between the cutting contract and the new transportation contract. Finally, the hash of the newly created transportation contract is returned along with the generated QR code.

Lastly, the last flow chart that will be discussed is the one related to the realization of the cut request use case. This use case is done by the cutting firm, thus needing to be connected with the registered cutting firm wallet address. The cutting firm first chooses a cutting contract and sends a transaction to the smart contract. Since each cutting contract is linked to a cutting firm, the chosen cutting contract needs to belong to the cutting firm which has the wallet address that has sent the transaction. This way, other cutting firms do not have the ability to cut trees using a cutting contract belonging to another firm. This is also checked by the system when initializing the cut request use case. When creating a cutting contract, a number of trees need to be specified, which represents the number of trees that the cutting firm is allowed to cut using the contract. The number of cut trees cannot exceed this agreed upon number. In the present system, the number of actually cut trees is not reported anywhere, and no check is being done when cutting the trees. By implementing such a system, it can check when requesting a cut, if there are anymore trees left that can be cut. This use case is needs to be realized before actually cutting the tree, since the system needs to check if the cut is allowed or not. To be able to access the number of cut trees for each contract, the contract hash needs to be linked to the number of cut trees so far. This number is updated each time a cut request arrives and the checks are valid. Figure 4.6 presents the flow chart for the cut request use case. As before, if any of the checks, related to the transaction sender's address or to the number of cut trees, fails, than the transaction will revert and the user is prompted with an error describing the problem.

The use case of monitoring the data is a simple one, thus no flow of events will be presented for it. Users can view data about cutting firms, cutting contracts, transportation contracts and the links between these. Each cutting firm have its own cutting and transportation contracts and each cutting contract can have multiple transportations. Transportation contracts

are also linked to vehicles, thus making it possible to see all transportations that have happened on the vehicle.

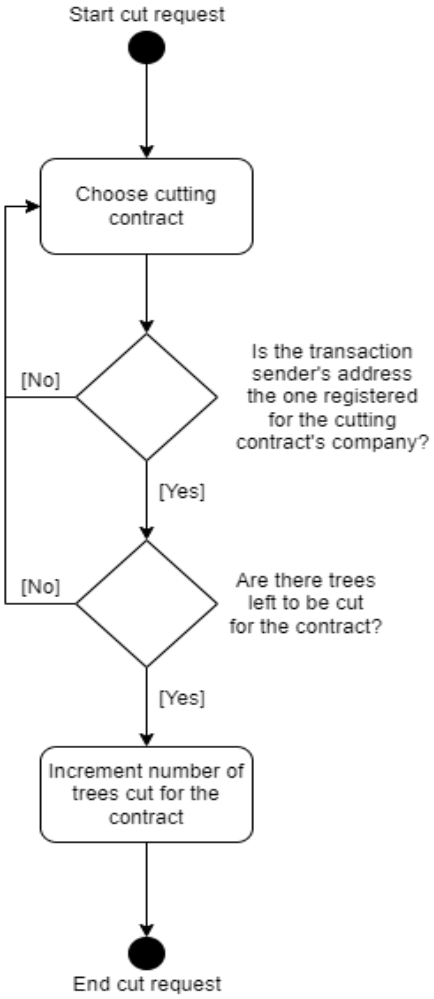


Figure 4.6 Flow of events for cut request

4.6. System sequence diagram

The System Sequence Diagram (SSD) shows, for a particular use case scenario, the events generated by external actors, their order and the inter-system events. These present the basic flow of the application from the perspective of events that come externally, to initiate the use case and internal events that are between the modules of the application.

The events of this application are initiated by the users of application. In case of the more complex use cases, that save data on the blockchain, these actors are the foresters and the cutters. To interact with the system, the actors use the website GUI, which initiates the transaction and communicates it to the smart contract deployed on the Ethereum blockchain. Thus, any use case will have an external event coming from the actor having the necessary roles to perform the use case, which is directed to the GUI of the application, or the website platform. The GUI than creates a internal event which is sent to the smart contract, which executes the business logic of the use case. This is presented in the following paragraphs that show the sequence diagrams for the registration, contract creation and cut request use cases.

The two registration processes, of the cutter and the forester, have similar sequences of events. Firstly, an external event is initiated by the Forester to the GUI. The Forester actor

than needs to enter the details mandatory for the registration. In case of a forester, this will be an Ethereum wallet address, while for the cutter it will be the TIN, the company name, the company's phone number and the Ethereum wallet address of the new cutter. Next, the website creates and initiates a transaction to the address of the smart contract on the Ethereum blockchain. The smart contract responds with the error or the success event of the transaction. This is than translated to a simple error or success message that is shown to the user. Since the registration sequence of events is the same for both the cutter and forester registration, a general sequence diagram can be built, in order to showcase these. Figure 4.7 shows this sequence diagram for the registration use cases.

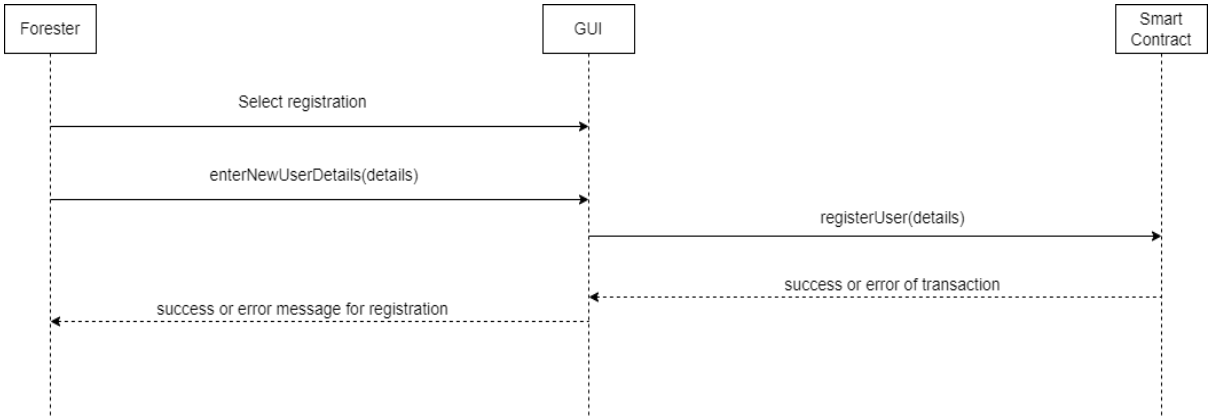


Figure 4.7 System sequence diagram for registration use cases

Similarly, the two use cases for the creation of cutting and transportation contracts are similar to one another. A general sequence can be described for these two, the diagram of which is shown in Figure 4.8. An external event comes from the Forester, who enters the details needed for the contract. The GUI than sends the information through a transaction to the deployed smart contract, which verifies the data and gives the verification results. In case this verification succeeds and no problem was found during the transaction sending, a success message is shown to the user. In case any error happened, or the specified details contained bad data, the problem is described to the user using an error message. In case the contract was created successfully, the calculated contract hash is sent from the smart contract to the GUI, which returns this to the user, along with the QR code it generates based on the contract hash.

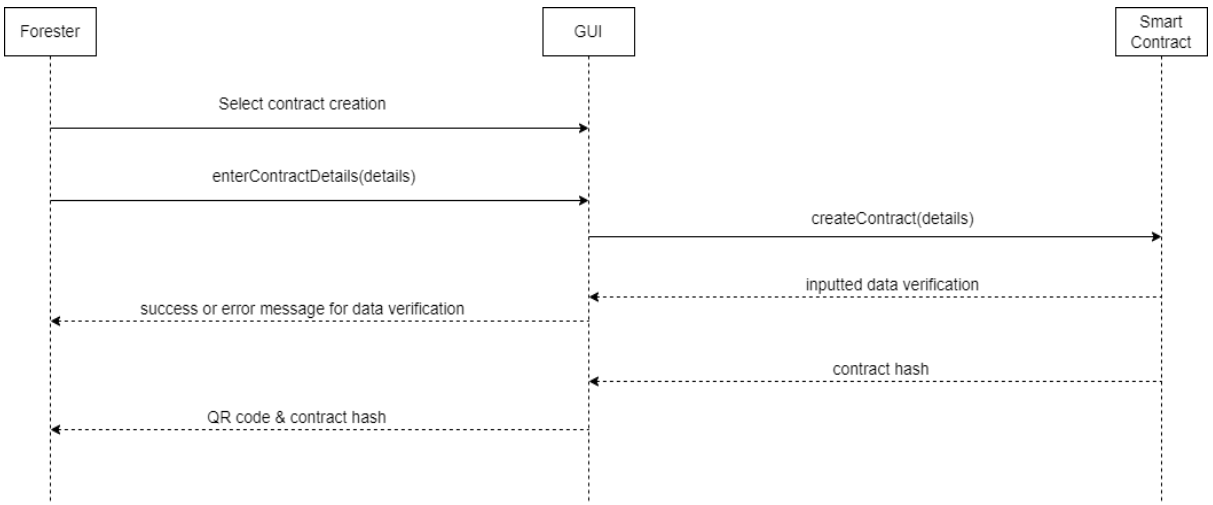


Figure 4.8 System sequence diagram for contract creation use cases

Finally, the sequence diagram of the cut request use case is discussed. The diagram can be seen in Figure 4.9. The use case starts with the Cutter selecting a cutting contract that they own. After this, they can request a cut, by using the hash of the selected cutting contract. This interaction can be done by using a simple button on the cutting contract's webpage. The GUI than sends the transaction for the cut request to the smart contract, that responds with a success or failure message. The smart contract performs the checks discussed in the flow of events chapter in order to conclude if the transaction succeeds or not. If the request is accepted, a success message is displayed to the cutter, on the GUI. Else, in case the checks or the transaction has failed, an error is displayed describing the reason.

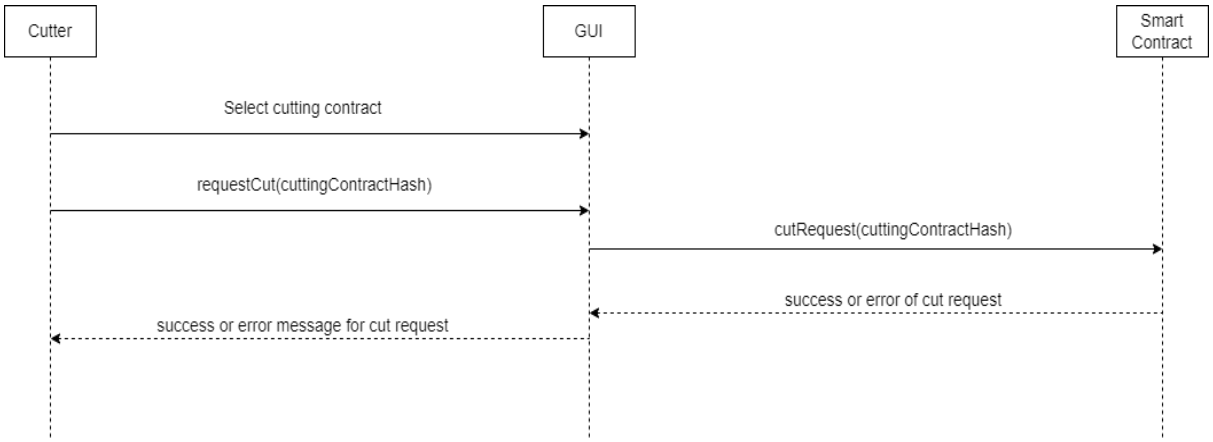


Figure 4.9 System sequence diagram for cut request use case

4.7. Domain model

The domain model is a visual representation of conceptual classes in the domain of the application. It depicts real world object that are candidates to be classes in the application and it is regarded as one of the most important artifacts during Object-Oriented Analysis. The conceptual classes or domain objects have associations between one another, which is a relationship between the two instances, that indicate some meaningful connection. Domain objects also have attributes associated to them, which is a logical data value of the object.

For the application described in this thesis, six conceptual classes have been identified. Firstly, the Wallet domain object, which has an address attribute, which represents the address for the Ethereum wallet. Both the Forester and Cutter objects need to have such a wallet in order to interact with the system. A Forester and a Cutter object can only have a single address registered, thus this creates a one-to-one association for these objects to the Wallet object. The Cutter object also has some attributes that represent general information about the cutting firm. These are the Taxpayer Identification Number (TIN), the name of the company and a phone number. The Forester object is associated with the CuttingContract object, as it is the creator of such objects. Same with the Cutter, which is associated with CuttingContracts, since it owns them. Both the Forester and Cutter objects have a one-to-many association with the CuttingContract class, because a Forester can create multiple CuttingContract, and a Cutter can have multiple CuttingContract associated with it. CuttingContract objects include the information about the contract, the agreed number of trees, that the contract can be used for cutting, the creation time of the contract and the number of cut trees so far for the contract. CuttingContract objects are associated with TransportContract objects in a one-to-many association, since one CuttingContract can have multiple transports. Such TransportContract

classes contain the number of trees that have been transported using it and the departure time of the vehicle.

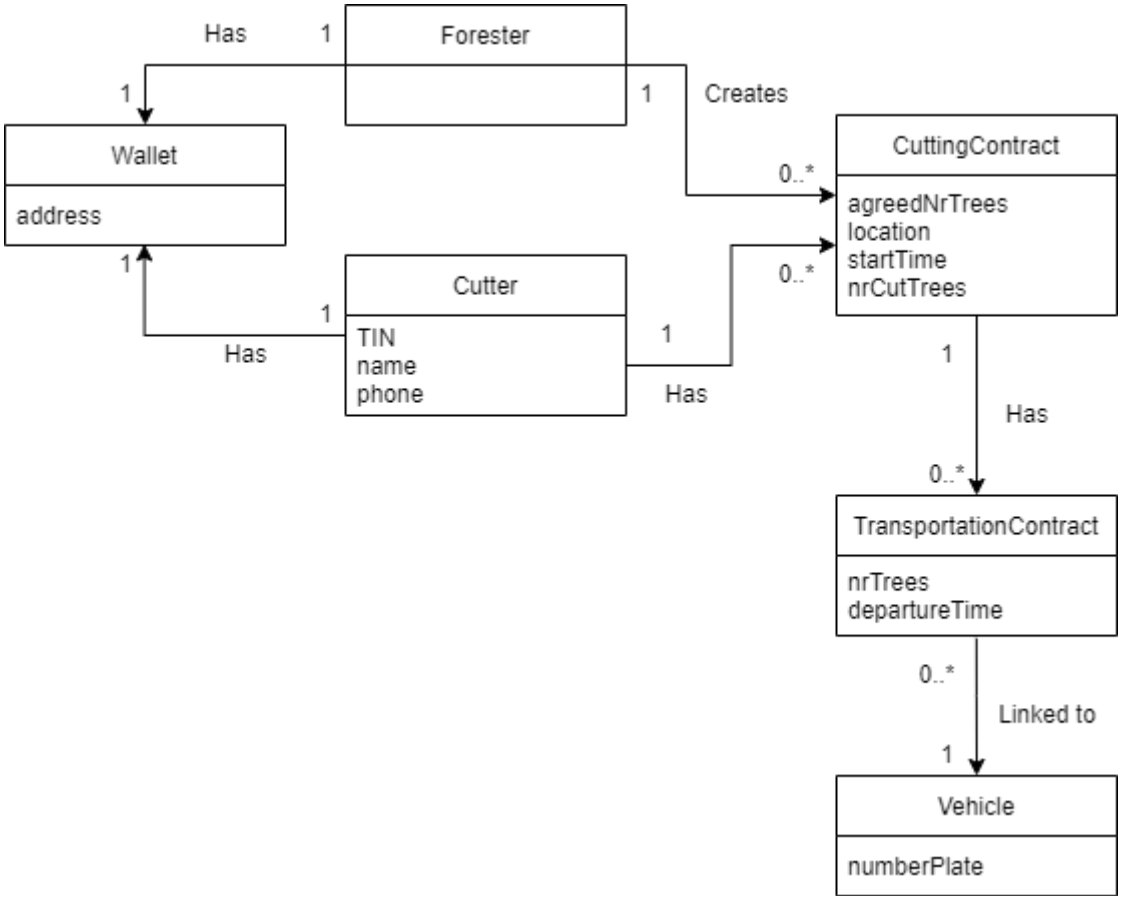


Figure 4.10 Domain model

Chapter 5. Detailed Design and Implementation

This chapter dives into the design and implementation details of the project. It starts by building a general schema for the application, showing at high-level how the different users and interact with the application's components. Then it describes the overall architecture of the program, along with the description of every implemented component at module level. It details the classes used and the important methods for the key classes, concentrating on the design decisions that were taken in order to create an application that achieves the previously mentioned goals.

5.1. Scheme

This thesis describes a decentralized application that is accessible on a website and its smart contracts are deployed on the Ethereum blockchain. The users can connect to the website using Ethereum wallets and by using the website, create transactions that interact with the services offered by the smart contracts. The deployed smart contracts can be interacted with by the website, by connecting to an Ethereum node, running the Ethereum software and having access to the smart contracts.

Figure 5.1 presents a schema of the application. The foresters are cutters are connected to an Ethereum wallet that is connected to the website platform. The website interacts with the Ethereum nodes, that interact with the deployed smart contracts, providing the required services. Users such as authorities and general users can also interact with the website, but only view the data present in the smart contracts.

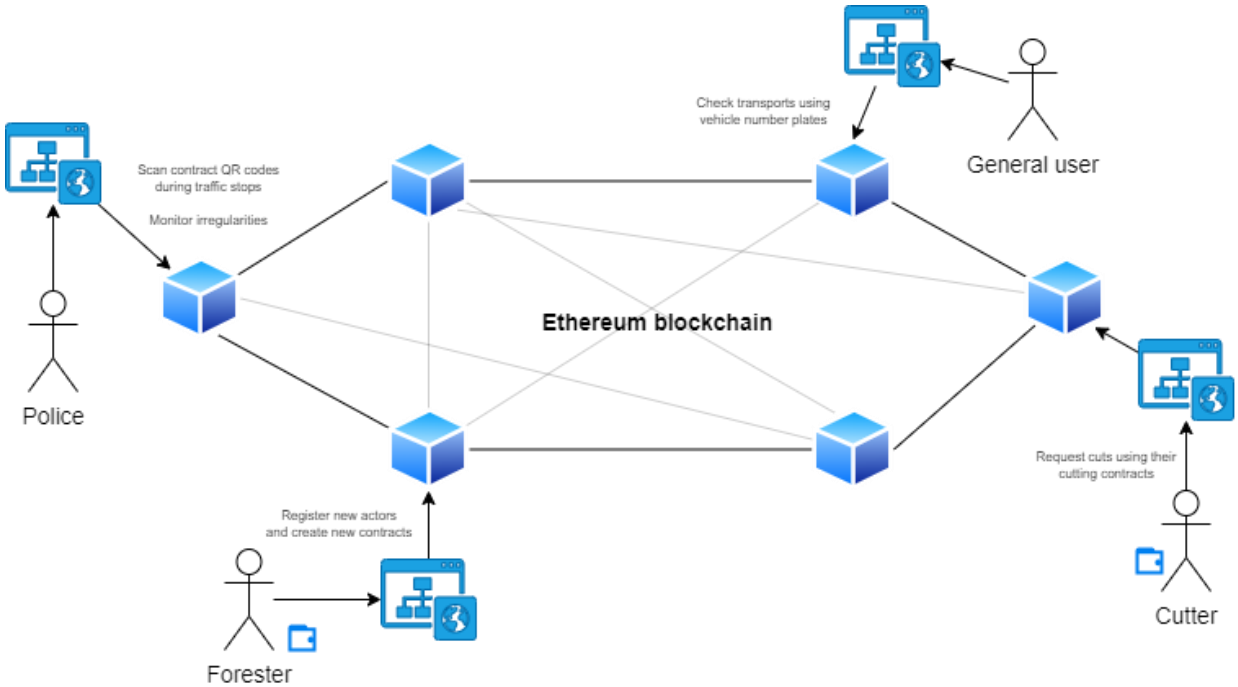


Figure 5.1 Application scheme

5.2. System architecture

The system is composed of two modules: the web application and the smart contracts deployed on the Ethereum blockchain. The web application acts as the client application, which provides an interface to allow users to request services and use the application, while the smart contracts are the server applications, which receive the requests coming from users on the client application and provide their implemented services. In order to establish an effect communication method between the two modules, the web3.js package was used, which is an Ethereum JavaScript API. The packages and the modules will be discussed, in detail, in the following paragraphs. Figure 5.2 presents the architecture diagram of the implemented application, showing its decomposition into modules and the communication between them.

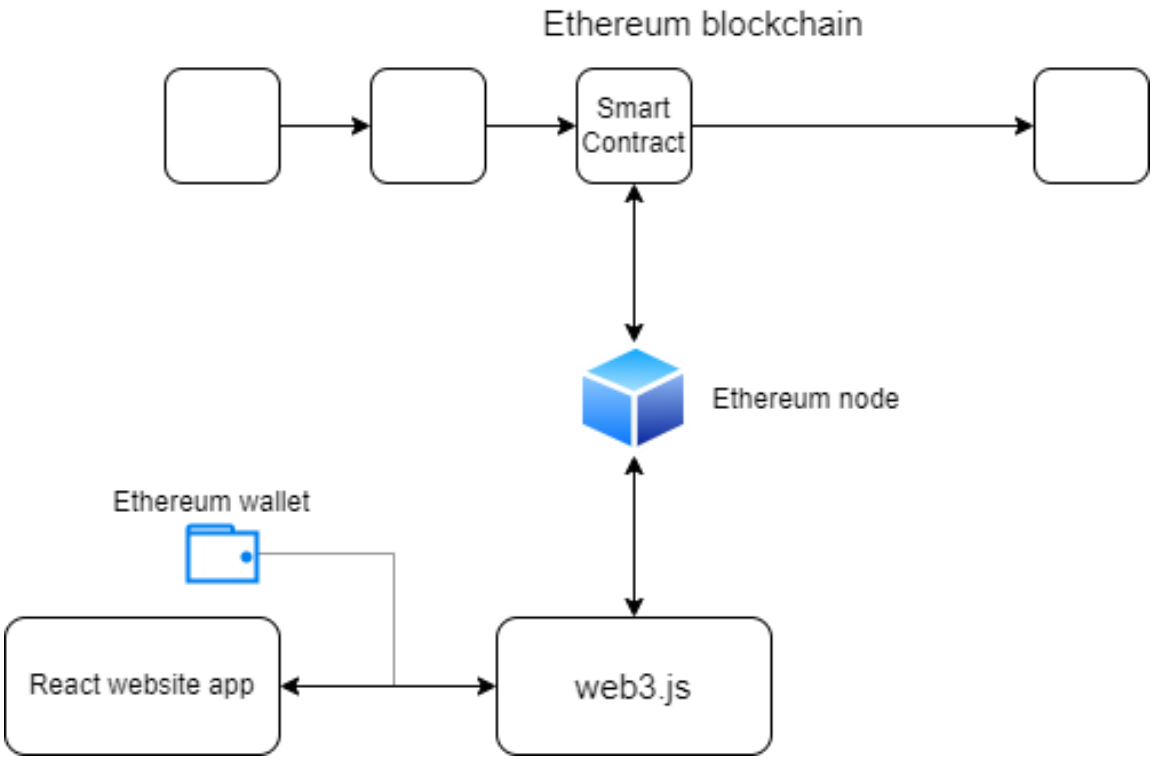


Figure 5.2 System architecture

5.2.1. React website application

The frontend or client application was implemented using React. It was the preferred JavaScript library, as it boosts the speedup of development. React allows the development of reusable components, which saves a lot of time during development, as there is no need to write various codes for the same feature. Furthermore, if any change is made in any particular part, it will not affect other parts of the application. It also increases flexibility, by making the code easier to maintain due to its modular structure. This flexibility, in turn, saves huge amount of time and cost for the application. Additionally, it is designed to provide high performance. The core of the framework offers a virtual DOM program, which makes complex application run extremely fast. This is an important aspect in case of the application described in this thesis, since it also depends on the speed of the blockchain. It is not desired to lose additional loading speed due to the website's performance. Websites built using React are also compatible with mobile browsers, thus making them usable on the mobile. React can also be used directly to build mobile applications. This is also important for our system, as it aims to be usable on

mobile devices too, to offer more flexibility to the user, who needs to be able to access the platform without a desktop device. The police need to be able to check transports when stopping transport vehicles on the road, and not having access to their laptops. As React uses JavaScript for building web applications, it can be used along with Node Package Manager (npm), which is the world's largest Software Register. It contains over 800,000 code packages from open-source developers sharing their software. It consists of a command line interface (CLI), which makes it easy to install new packages and also helps with managing the React application's package.json file, which declares all the used packages. In the context of the application described in this thesis, multiple packages are needed. For the development of a user-friendly user interface, the React Suite library is used. This is a React component library, that offers a friendly development experience for designing interactive website platforms. Furthermore, to easily connect with the deployed smart contracts, the web3.js package is used, which will be presented in more detail in the next paragraphs. Truffle packages are used for the deployment of the smart contracts. This will be presented in the deployment section of this chapter. QRCode-React package is used as well, which generates the QR codes for the of the created cutting and transportation contracts. Due to these reasons, React was the favorable framework to build the client application with.

In order for the website application to communicate with the deployed smart contracts, located on the Ethereum blockchain, the web3.js Ethereum JavaScript API package is used. Web3.js is a collection of libraries that allow web applications to interact with a local or remote Ethereum node using HTTP, IPC or WebSocket, as described in [25], which is the documentation of the current web3.js package, v1.7.3. Web3.js communicates with the Ethereum blockchain with JSON RPC, which stand for remote procedure call protocol. It is used to make requests to an individual Ethereum node with JSON RPC in order to read and write data to the blockchain's smart contract. It is similar to how usual centralized client-server applications communicate using Rest API, or jQuery. Figure 5.3 presents the way in which decentralized application are built and how they offer a web application, that can communicate with the Ethereum blockchain by using web3.js. The web application uses web3.js, which sends JSON RPC requests to a Web3 provider, which is connected to an Ethereum node. The Ethereum node, which has connection to the blockchain, or has a copy of the blockchain can than provide access to the decentralized application's smart contracts. In the following paragraphs the concept Web3 provider will be presented, along with its types and the providers used by this application.

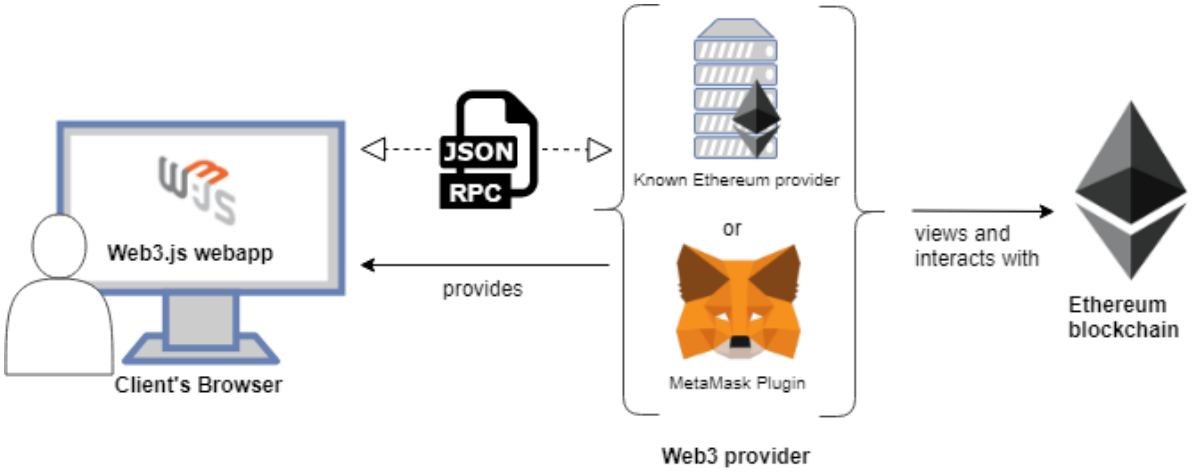


Figure 5.3 Decentralized application communication between components

In order to able to communicate with a smart contract deployed on the Ethereum blockchain, it is needed to use a Web3 provider. A web3 provider can be a wallet application, such as MetaMask or other known Ethereum providers, such as Infura. A web3 provider is an abstraction of a connection to the Ethereum network, providing a consistent interface to standard Ethereum node functionality. They are essentially blockchain data keepers. These projects run networks of hundreds of blockchain nodes and are tasked with providing applications with the latest and historical blockchain data. Nodes can be of multiple types, but typically they contain a copy of the full or part of the blockchain, as described in the Bibliographic Research chapter. They can validate transactions and thus act as a bridge between the blockchain and the user.

MetaMask is a software cryptocurrency walled that is used to interact with different blockchains, such as Ethereum or Binance Smart Chain. It allows users to access their wallet through a browser extension or mobile application, which can be used to interact with decentralized applications. Users can manage their keys in a variety of ways, including hardware wallets, isolating them from the website. Developers can simply interact with globally available ethereum API that identifies the users of web3-compatible browsers and whenever a transaction is initiated or a signature is needed from the user, MetaMask will prompt the user with a comprehensive MetaMask window. This keeps the user informed and makes mass hacks from attackers less likely. As described in [29], MetaMask injects a global API into websites, visited by the user, at window.ethereum. This API allows websites to request users' Ethereum accounts, read data from blockchains the user is connected to and suggest that the user sign messages and transactions. Thus, for development, window.ethereum can be used as the API that connects the user's wallet to the decentralized application. MetaMask pre-loaded with fast connections to the Ethereum blockchain and several test networks via Infura. This allows to get started without synchronizing a full node, while still providing the option to upgrade your security and use the blockchain provider of your choice.

Infura is a blockchain development suite that provides application programming interfaces (APIs) and developer tools. Moreover, Infura provides fast and reliable access to the Ethereum network to enable developers build next generation decentralized and Web3 applications that meet user demand. Infura is an Infrastructure-as-a-Service (IaaS) and Web3 backend infrastructure provider that speeds up the development process of a Web3 application. This is achieved by reducing the time spend building infrastructure from scratch. Infura offers enterprise-ready infrastructure using a distributed cloud-hosted network of nodes. For the development of this project, one important aspect of Infura is that it is a blockchain node provider, which makes accessing the Ethereum blockchain a lot faster. Rather than having to sync a node, Infura can communicate with one from its cloud network. Furthermore, it provides developers with the Infura Ethereum API, which makes it possible to connect applications in just a single line of code. This is a microservice-driven architecture, that dynamically scales to the user's demands and makes it possible to connect with the Ethereum blockchain using WebSockets and HTTPS. To sum up, Infura greatly increases development speed and allows users to allocate more time and resources to product development, instead of infrastructure building. In the application presented in this thesis, Infura is used by MetaMask, to connect to its cloud-hosted network of Ethereum nodes, but it is also used directly by the application. In case a user connects to the website, without having the MetaMask plugin installed on their browser, or by phone, the website needs to be able to connect to the deployed smart contracts and read its data. This is possible by directly using the Infura Ethereum API, which consists of an endpoint link for the application, using which users can read the data from the smart contracts without using a wallet address. Of course, this cannot be used to send transactions, or create new data on the blockchain. For that purpose, the user needs to have the MetaMask extension installed on their browser and connect its Ethereum wallet to the website platform.

5.2.2. Smart contracts

The smart contracts contain the business logic of the application. They act as the backend, or server part of the application and offer services to the users that call for them using the client web application. This backend is not as the common Spring backend application, connecting to a centralized database, but rather it is decentralized, and all data is publicly available and immutable. Smart contracts contain the functions needed for the purpose of implementing all use cases of the applications and the data about the actors, contracts and information about these. They represent programs that are executed inside a peer-to-peer network where nobody has special authority over the execution.

The smart contracts are implemented using Solidity. Solidity is an object oriented, high-level language that is targeted for the Ethereum Virtual Machine (EVM). Its documentation paper, [30], describes it as a statically typed curly-braces programming language designed for developing smart contracts that run on the EVM. It was influenced by C++, Python and Java, supporting inheritance, libraries and complex user-defined types among other features. It also comes with a compiler, which encodes the smart contracts and outputs byte code and other artifacts needed for deployment and interaction with the smart contract. One of these artifacts is the Contract Application Binary Interface (ABI), which is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. The other artifact is the contract bytecode, which is then deployed on the blockchain. This will be discussed in the deployment section of this chapter. Figure 5.4 shows the outputs of the Solidity compiler, after a smart contract compilation.

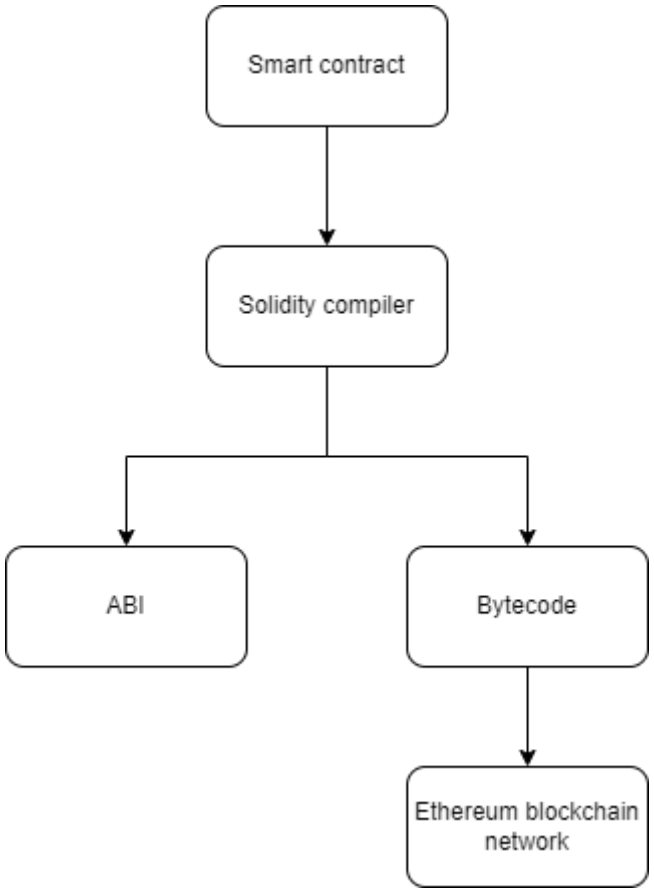


Figure 5.4 Solidity compilation process

Using solidity, programmer can create structs, similar like in C/C++, which can be used to save data. Structs have an identifier name, used to create new instances and attributes, which are defined for each instance of the struct. Since there is no database, that the program can connect to, it needs to save all data inside the smart contract itself. Structs are specifically useful for this task. Solidity also provides arrays, which are the same as in other programming languages. Each element of the array can be accessed using an index, indicating its position in the array. Mappings are also an option, which are implemented using hash tables, which are virtually initialised such that every possible key exists and is mapped to a value, which is the mapping's value type's default value. Mapping are of the form mapping(keyType => valueType) variableName. The key type is actually not stored in the mapping itself, only its keccak256 hash is used to look up the value when needed, in O(1) time complexity. Structs, arrays and mappings represent the reference types of Solidity. They can be modified through multiple different names. In case of reference types, programmers need to explicitly provide the data area where the type is stored. These can be the following:

- Memory: whose lifetime is limited to an external function call.
- Storage: the location where the state variables are stored, where lifetime is limited to the lifetime of the smart contract.
- Calldata: special non-modifiable, non-persistent data location that contains the function arguments. It behaves mostly like memory.

Solidity contracts are similar to Java classes, containing persistent data in state variables and functions that can modify these variables. Calling a function on a different contract instance will perform an EVM function call and thus switch the context such that state variables in the calling contract are inaccessible. Due to this reason, getter or setter functions need to be created in order to access or modify the state variables of other contracts. For state variables, setters are automatically created by Solidity. Contracts can be created programmatically using the previously discussed web3.js Ethereum JavaScript API. It has a function called web3.eth.Contract to facilitate contract creation. When a contract is created, its constructor is executed. This constructor is optionally implemented by the programmer, mostly in order to initialize the contract's state variables, similarly to constructors in object-oriented programming languages, such as Java. After the constructor has executed, the final bytecode of the contract is stored on the blockchain, having all public and external functions reachable through function calls. Functions in Solidity can have their visibility types, specifying from where they can be called. The functions of a Solidity contract can have four visibility types:

- External: part of the contract interface, which means that they can only be called from other contracts or via transactions. They cannot be called internally, from the smart contract itself.
- Public: can be called internally or externally.
- Internal: can only be accessed from within the current contract or contracts deriving from it.
- Private: similar to internal ones, but they are not visible in derived contracts.

Furthermore, functions need to specify their state mutability. This specifies how the function will change the state of the contract. These are the following:

- View: functions declared as view can only read the state, but not modify it.
- Pure: functions declared as pure can neither read nor modify the state of the smart contract.
- Payable: functions declared with payable can accept Ether sent to the contract, if it is not specified, the function will automatically reject all Ether sent to it. Especially useful for smart contracts implementing finance applications.

5.3. Implementation

The application uses the client-server architectural pattern. The web application acts as the client, requesting services from the server. The deployed smart contracts act as the server application, implementing the business logic of the system and responding to the client with these implemented services. In the following, the implementations of these applications will be presented, along with class diagrams of the important classes.

5.3.1. Server application

The server application serves as a decentralized service provider to the whole system. Rather than having a centralized backend application with a private database holding its data, this application has all its data and services public, making it more trustworthy and reliable. This is important for this system, as its main goals. The server application is implemented into three separate smart contracts deployed on the Ethereum blockchain. This way the registration, cutting contract creation and transport contract creation can be separated into different smart contracts. The name of these contracts are: ActorsRegistration, Transportation and TreeCutting.

The actor registration smart contract implements the logic needed for the registration of new cutting firms and foresters. In order to save the information of each cutting company, a struct is implemented, holding the name, phone and wallet address of each company, as shown in Figure 5.5. The data type for phone numbers is bytes10, which allows the saving of 10 bytes, or 10 length strings. The wallet address of the company is saved in Solidity's special address data type which is specialized in holding address located on the blockchain.

```
struct Company {
    string name;
    bytes10 phone;
    address walletAddress;
}
```

Figure 5.5 Company struct

In case of smart contracts, storing data inside of arrays is not always a good action. This is because of gas fees. When using arrays to store data, for loops need to be used in order to find and operate on the elements of the array. Due to this reason, when the array is large, the for loop will need to do large amounts of work. Because in Ethereum every operation has its gas fee, the bigger this for loop is, the larger the gas fee will be. Furthermore, if for example one of the last elements in a large array is needed, the for loop will need to iterate through each element of the array, but because the gas limit is set by the user, and there is a limit for each block in Ethereum, it is possible that the gas cost to reach the desired element will surpass these limits. This will result in a failed transaction every time the function is ran. Thus, it is important to reconsider the implementation of programs that need to use arrays with an undefined number of elements. Effectively, operations that act on every element of an array, or ones that need looping through a variable number of array elements, should be done on the client application, or the design should be revaluated. In addition, when needing to operate on a single element of an array, that can be at any unknown location, it is indicated to implement the application in such a way that the finding of this element is done in O(1) and doesn't need a for loop. This is where mappings come into play. Mappings offer O(1) lookup time for any element, by storing each element in a hash table, at the position of the key value's keccak256 hash. Thus, mappings are an excellent choice when operations are needed on single elements of data. But, what

mappings do not offer is that they cannot be used in order to get all the companies that are saved, for example. It is not possible to return a mapping to the client application. Because of this, arrays need to be used in order to return all of the saved data. They cannot be returned as a whole to the client application, but it is possible to return the length of the array and by using a for loop in the client application, getting each element of the array. Therefore, I came up with a solution where I save the TINs of the companies in an array and use the TINs as key values in a mapping, which maps a company TIN to a Company struct. TINs are saved as bytes8 data types, which make it possible to save 8 bytes, which is the length of TIN numbers in Romania. The TIN of a company is constant, it is a number issued by the Internal Revenue Service when the company is created and registered. The company uses this TIN as an identifier for tax paying purposes. Thus, there will never be a case in which the TINs saved in the array will need to be updated. They will keep getting pushed on top of the array and being immutable. There is no function that lets users modify or delete the contents of this array. The mapping will point from a registered TIN of a company to its information, that is saved into a struct having the form presented in Figure 5.5. Additionally, in order to have a O(1) lookup of a wallet address to a company, I have created a mapping from an address key value to a TIN number, which than in turn can be used to get the information about the company using the previously mentioned mapping. In the case of foresters, I used a simple mapping from an address data type, representing the forester's Ethereum wallet address, to a bool variable. In case of foresters, there is no need to get all forester wallet addresses in the client application. The only use for this mapping is to check if a wallet address that is sending a transaction to the smart contract, is a registered forester address or not. This is the way that application checks for the forester roles, when registering new actors or creating new contracts. In order to be able to register foresters and cutting companies, and to essentially start using the application, I created a constructor for this smart contract, that sets the message sender's address as a forester. The constructor function of a smart contract is executed only once, when it is deployed to the Ethereum blockchain. Due to this reason, the address deploying the smart contract will be automatically set as a forester. This address acts as an initial admin of the whole system. Figure 5.6 presents the state variables of the actor registration smart contract.

```
bytes8[] public companies;
mapping(bytes8 => Company) public companyInfo;
mapping(address => bytes8) public addressCompany;
mapping(address => bool) public foresters;
```

Figure 5.6 ActorsRegistration state variables

To implement the registering functionalities, I have created a modifier, which is used to change the behaviour in a declarative way. In this case, it was used to create a prerequisite to a function, and to make a condition be checked automatically when a function is executed. Modifiers are declared in the signature of a function. This modifier is used to check if the transaction sender's wallet address is a registered forester. In order to get the transaction sender's address, Solidity's msg.sender can be used, which will return the wallet address in Solidity's address data type. To check if this address is a registered forester, the previously defined foresters mapping can be used, if the bool is true, the address is a forester. By defining this modifier, the code for checking for the forester role does not need to be repeated for both forester and cutter registration functions. The modifier can be specified on both function signatures. To make sure that the transaction will fail if the message sender's address is not a registered forester, Solidity's require function can be used. In a require function, a condition is checked. In case this condition fails, the transaction is aborted, reverting all the remaining gas fees to the

transaction sender. In case the condition succeeds, the code after the require statement is executed. In case of the forester registration, the operation is simple. The modifier's check is executed and if the wallet address is a registered forester and then the foresters mapping's value at the position of the new forester address, received as a parameter, is set to true. Thus, effectively registering a new forester address. The registration of cutting companies is more complex. It requires four arguments, the TIN of the company, the name of the company, a phone number, where the company can be reached and a wallet address, that the cutting company will use. It is similar to the forester registration, using the modifier, but it also has other requirements checked by two other require statements. First it checks if the provided TIN is already registered or not. Thus, it will not let the registration of a single company multiple times. As discussed before, in case of mappings, each possible key value is virtually loaded and as default, at each key value position in the mapping, the default value of the value object will be placed. Thus, making it possible to create check for a TIN to be registered or not. The companyInfo mapping at position TIN will return a Company struct that will have default values for all its fields, in case the cutter is not registered, or specific non-default values in case it is registered. I choose to check if the wallet address field of this struct is equal to the default value for an address data type, that can be denoted by address(0), and has the value 0x00…0. Secondly, it checks if the wallet address has already been used by a cutter company, making it impossible to register two different companies using the same wallet address. For this, simply the addressCompany mapping can be used and checked if it has the default value of bytes8, which is denoted by bytes8(0), similarly to address. If all these checks succeed, the registration process can begin. A new Company struct is created, holding the name, phone number and wallet address provided in the function arguments. This struct is placed in the companyInfo mapping at position TIN. The provided TIN is also pushed on top of the companies array, and lastly the addressCompany mapping is updated, as to contain the TIN in the position of the provided cutter firm's wallet address. Figure 5.7 shows the previously described code for the two registration functions.

```
function registerForester(address foresterAddress) onlyForester external {
    foresters[foresterAddress] = true;
}

function registerCutter(bytes8 tin, string memory name, bytes10 phone, address walletAddress) onlyForester external {
    require(getCompanyAddress(tin) == address(0), "Cutter company already registered");
    require(addressCompany[walletAddress] == bytes8(0), "Ethereum address already used by another company");

    Company memory company = Company(name, phone, walletAddress);
    companyInfo[tin] = company;
    companies.push(tin);
    addressCompany[walletAddress] = tin;
}
```

Figure 5.7 Forester and cutter registration functions

The tree cutting smart contract implements the functionalities of creating cutting contracts and requesting cuts. They contain all of the created cutting contracts and information about them. In order to save the information about the cutting contracts, a struct is used, which contains the TIN of the involved cutting company, the agreed number of trees that can be cut using the contract, the location where the contract can be used in, the contract's creation time and the number of trees cut so far. This last number can be incremented when a new cut is requested and accepted by the system. The struct containing the information about cutting contracts can be seen in Figure 5.8.

```
struct CuttingContract {
    bytes8 tin;
    uint16 agreedNrTrees;
    string location;
    uint startTime;
    uint16 nrCutTrees;
}
```

Figure 5.8 Cutting contract struct

In order to optimally store the data about cutting contracts, I used the same strategy as for cutting companies in the actor registration smart contract. But, due to the reason that a cutting contract does not have any constant unique identifier that can be stored inside an array, it needed some adaptation. Solidity offers the possibility to use the keccak256 cryptographic hash function, that takes in an input and converts it to a unique 32-byte hash. I used this hash function to create a unique identifier, or fingerprint, for each cutting contract. When a forester creates a new cutting contract, this hash will be calculated and used to store the cutting contract in the smart contract. To represent this hash, the bytes32 data type is used, which can hold the 32-byte hash generated by the keccak256 function. This way, the hash of the contract can be used to by other contracts and by the client application to identify and use any cutting contract registered in the system. Thus, similarly to the actor registration contract, I created an array containing the cutting contract hash functions and two other mappings. The first mapping uses the bytes32 hash of a contract as key, to point to a CuttingContract struct, that contains the information about the contract. The second one is a mapping from a TIN, which is saved as a bytes8 data type, to an array of bytes32 elements, which denote the cutting contract hashes of a company. This way, one can get all the cutting contracts of a company by simply using the company's TIN, making it possible to display the cutting contracts of each individual firm on the client application's end. Besides these mappings, there is also a state variable containing a actors registration smart contract instance. This is used to verify the roles of each sender for additional security checks. To use an instance of another smart contract, it is needed to have its address where it is deployed on the blockchain. By using the address, one can initialize the instance of the smart contract and use its external and public functions. Due to security reasons, the address of the actors registration smart contract is passed through the constructor of the tree cutting smart contract, and the instance is initialized there. Thus, the state variables and constructor will look like in Figure 5.9.

```
bytes32[] public contractHashes;
mapping(bytes32 => CuttingContract) public contractInfo;
mapping(bytes8 => bytes32[]) public companyContractHashes;

ActorsRegistration private actors;

constructor(address actorsContractAddress) {
    actors = ActorsRegistration(actorsContractAddress);
}
```

Figure 5.9 Tree cutting contract state variables and constructor

The function implementing the cutting contract creation needs 3 arguments, the TIN of the cutting company associated with the contract, the agreed number of trees that can be cut using the contract and the location where it can be used. Before the contract creation, there are 2 require statements. The first one checks that the transaction sender's wallet has a registered forester address. The second one checks that the TIN provided as argument is of a registered cutter company. This way, it makes sure that only foresters can create cutting contracts and that these need to belong to already registered cutting companies. After these checks, a new instance of a CuttingContract struct is created. This uses the TIN, agreed number of trees and location arguments provided as parameter, but it also uses Solidity's block.timestamp value, which will give the current block's timestamp in the Unix timestamp and it will also set the number of trees cut so far to 0. The Unix timestamp shows the number of seconds that have elapsed since January 1st, 1970. It can be easily converted in the client application to a more user-friendly date format. After the struct instance was created, its hash is calculated using the keccak256 function. In order to use the keccak256 function on a struct, it is needed to first ABI encode the struct, after which the hash function can be applied on the encoding. After these are created and calculated, it is time to save them to the smart contract's state variables. First the in the contractInfo mapping it will place the create struct to the calculated hash's location. Next the contractHashes array will have the new hash pushed on its top. After, the companyContractHashes's array at position TIN will have the new cutting contract hash pushed on top of it. Lastly, an event is emitted. Events in Solidity lets programmers pass arguments to it, which will be stored in the transaction logs. These events will be stored on the blockchain and remain accessible using the smart contract's address until it is present on the blockchain. This makes it possible to have a clearer view on all the transactions made with the smart contract using the Ethereum explorer and also, by using web3.js in the client application, the event's arguments can be retrieved and displayed on the client side. This event's name is CutCreated. Figure 5.10 shows the implementation of the function responsible with the creation of a new cutting contract.

```
function createCuttingContract(bytes8 tin, uint16 agreedNrTrees, string memory location) external {
    require(actors.foresters(msg.sender), "Not using a registered forester address");
    require(actors.getCompanyAddress(tin) != address(0), "Not using a registered cutter");

    CuttingContract memory cuttingContract = CuttingContract(tin, agreedNrTrees, location, block.timestamp, 0);
    bytes32 cutHash = keccak256(abi.encode(cuttingContract));
    contractInfo[cutHash] = cuttingContract;
    contractHashes.push(cutHash);
    companyContractHashes[tin].push(cutHash);
    emit CutCreated(cutHash);
}
```

Figure 5.10 Cutting contract creation function

The next function in the tree cutting contract is the one responsible with cut requests coming from the companies. This function takes a single argument, which is the hash of the cutting contract, that the cut is requested from. The function first gets the cutting contract's information, using the contractInfo mapping and the hash provided as argument. It stores this struct as a storage variable, meaning that changes done to this variable will be reflected in the struct stored in the contractInfo mapping. This is done to save gas, since multiple mapping lookups and saves to mappings cost a lot of gas. This is avoided by creating a storage variable from the beginning, thus no more lookups in the contractInfo mapping are needed and changes to the storage variable will also be saved and reflected. After the variable creation, two require statements are executed. The first one checks that the TIN message sender's wallet address is the address linked to the company associated with the cutting contract. This require also makes

sure that a valid cutting contract hash was given as argument, since this address will be address(0) in case the contract hash is not registered. This is because the when getting the contract struct using the given hash, if a non-registered hash is given as argument, it will get a struct that has the default value for each of each attributes. Thus, the TIN in the contract struct will have the default value of bytes8, which is bytes8(0). The second require checks if there are any more trees left to cut for the contract. It does this by checking if the agreed number of trees from the struct is bigger than the number of cut trees that is also saved in the struct. If both of these checks succeed, the number of cut trees for the cutting contract struct, saved as a storage variable, is incremented. This also changes the struct saved inside the contractInfo mapping, as discussed previously. The implementation for the cut request function can be seen in Figure 5.11.

```
function cut(bytes32 cutHash) external {
    CuttingContract storage cuttingContract = contractInfo[cutHash];
    require(actors.getCompanyAddress(cuttingContract.tin) == msg.sender, "Not using the contract's company address");
    require(cuttingContract.agreedNrTrees > cuttingContract.nrCutTrees, "No trees left for this contract");

    cuttingContract.nrCutTrees++;
}
```

Figure 5.11 Cut request function

The transportation smart contract is responsible for the creation of transportation contracts and its information as efficiently as possible to implement all use cases. The information about transports is saved inside a struct, which will hold the number of trees transported, the car's number plate, the cutting contract's hash, for which the transport is done, and the departure time. The car number plate is saved as a bytes7 data type, since vehicle number plates consist of 7 characters in Romania. The departure time is saved as Unix timestamp, similarly to the creation time for cutting contracts. The struct used to save transport information is presented in Figure 5.12.

```
struct Transport {
    uint8 nrTrees;
    bytes7 car;
    bytes32 cutHash;
    uint departureTime;
}
```

Figure 5.12 Transportation contract struct

The strategy for transportation contracts is the same as for cutting contracts, a hash is calculated using the struct mentioned above. There is a contractHashes array, holding all hashes and a contractInfo mapping, that maps a hash to the contract's struct. In order to create a link between cutting contracts and transportation contracts, a mapping is defined that maps a cutting contract hash to an array of transportation contract hashes. Thus, in the client application, it is possible to get all the transportations associated to a cutting contract without doing any filtering. This increases the gas fees by a little, but in case of thousands of transportation, this gas fee will be worth it to trade for the speed up. Besides this, I used a mapping from the transport contract hash to an unsigned int with 16-bytes. This has the purpose to keep the number of transported trees for each transportation contract. It was created in order to make sure that there aren't transportation contract created, for a cutting contract, that add up to more trees transported than have been cut for the contract. Thus making sure that no more trees are transported than the

number of cut trees that have not been transported yet. The last state variable is another mapping that maps the number plate of a vehicle, to an array of transportation contract hashes. It offers the possibility to show the transports of a transportation vehicle, thus making it possible to monitor and validate trees that are on a transportation vehicle. It offers a speedup, by providing the contracts hashes without the need to filter the array of all contracts. Besides this, both the actor registration and the tree cutting smart contracts are used by this smart contract. The registration contract is used to check the role of the message sender wallet address, while the cutting contract is used to get the number of trees that have been cut for a cutting contract. These are both initialized in the constructor of the transportation smart contract. The state variables of the Transportation smart contract can be seen in Figure 5.13.

```
bytes32[] public contractHashes;
mapping(bytes32 => Transport) public contractInfo;
mapping(bytes32 => bytes32[]) public cuttingContractTransportHashes;
mapping(bytes32 => uint16) public treesTransported;
mapping(bytes7 => bytes32[]) public carTransports;

ActorsRegistration private actors;
TreeCutting private cutting;

event TransportCreated(bytes32 transportHash);

constructor(address actorsContractAddress, address cuttingContractAddress) {
    actors = ActorsRegistration(actorsContractAddress);
    cutting = TreeCutting(cuttingContractAddress);
}
```

Figure 5.13 Transportation smart contract state variables and constructor

The function responsible with the transportation contract creation requires 3 arguments. The first one is the number of trees that can be transported using it, the second one is the transportation vehicle's number plate, represented as a bytes7 data type, and the third one is the cutting contract hash for which the transport is created. It begins with a require statement that checks whether the message sender is using a forester address or not. After this it calculates the new number of trees transported, by getting the old value and adding to it the number of trees provided as argument. Next, it checks if this newly calculated number of trees is smaller or equal to the number of trees cut for the cutting contract, whose hash is provided as argument. This makes sure that there cutters cannot request the creation of transportation contracts without a valid cutting contract, that has trees cut, and not transported. This eliminates the possibility of illegal tree transportation, since transportation contracts cannot be created without going through the process of having an active cutting contract and requesting cuts for it. If these checks succeed, the function assigns the new value for the number of trees transported and creates the Transport struct instance using the number of trees, the car number plate, the cutting contract hash that were provided as argument, and the block.timestamp value of Solidity for the departure time field. The hash of the newly created transportation contract is calculated, after which the information is saved in the smart contract. First, the struct is saved in the cutting contract info mapping's transportHash position. After which the transportHash is pushed to the array of hashes found in the cuttingContractTransportHashes mapping's cutHash position, which will link the newly created transportation contract with the cutting contract in the function arguments. Next, the transport hash is also pushed to the array of hashes found in the carTransports mapping, at position of the provided transport vehicle's number plate. This links the transport contract to the car provided as argument, thus being able to search

for transportations done by a vehicle. After this, the transport hash is pushed to the top of the contractHashes array, containing all of the transportation contracts. Lastly, an event is emitted, called TransportCreated, which contains the transport hash as an argument. Figure 5.14 shows the function responsible for transportation contract creation in the Transportation smart contract.

```
function createTransportContract(uint8 nrTrees, bytes7 car, bytes32 cutHash) external {
    require(actors.foresters(msg.sender), "Not using a registered forester address");

    uint16 newNrTreesTransported = treesTransported[cutHash] + nrTrees;
    require(newNrTreesTransported <= cutting.getContractNrCutTrees(cutHash),
            "Cannot transport more trees than what have been cut for cutting contract");
    treesTransported[cutHash] = newNrTreesTransported;

    Transport memory transport = Transport(nrTrees, car, cutHash, block.timestamp);
    bytes32 transportHash = keccak256(abi.encode(transport));
    contractInfo[transportHash] = transport;
    cuttingContractTransportHashes[cutHash].push(transportHash);
    carTransports[car].push(transportHash);
    contractHashes.push(transportHash);
    emit TransportCreated(transportHash);
}
```

Figure 5.14 Transportation contract creation function

To three contracts represent the service provider server side of the implemented system. They are deployed on the Ethereum blockchain, making it possible to request their services using RPC calls. The class diagram of the server application is presented in Figure 5.15

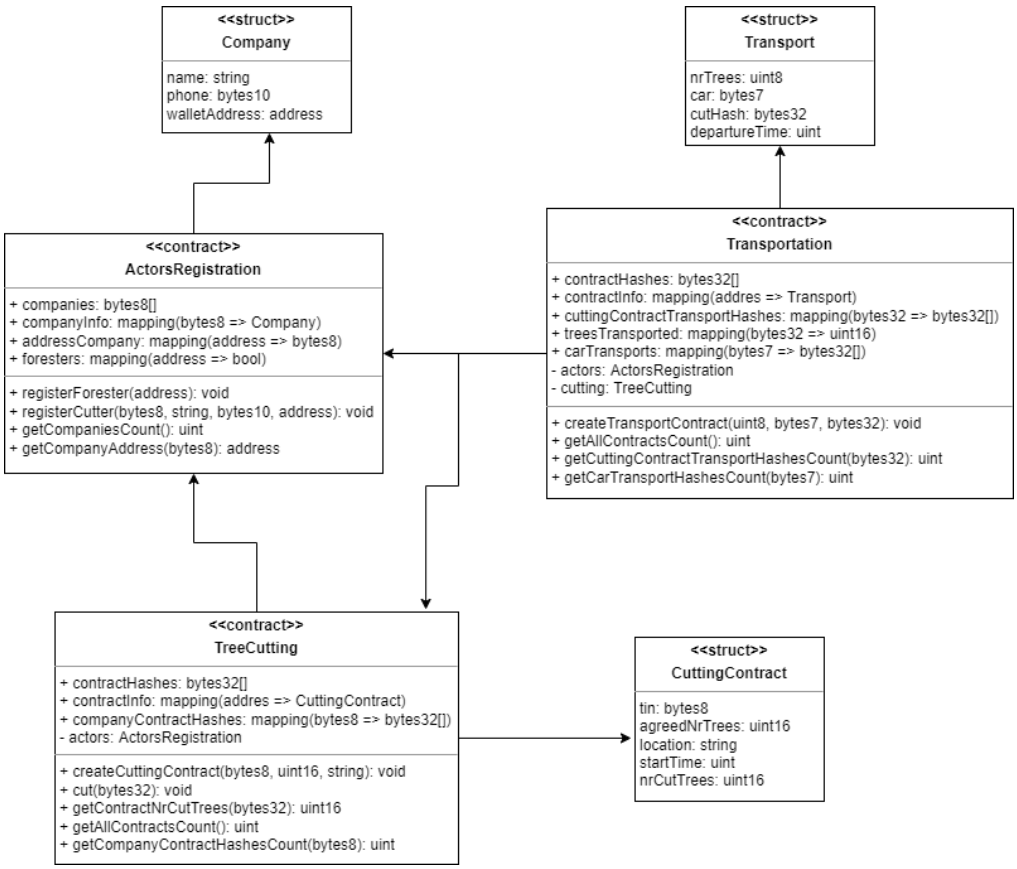


Figure 5.15 Class diagram server application

5.3.2. Client application

In order to implement the client application, we take advantage of the React components, thus being able to separate parts of the application having more complicated logic. React makes it possible to implement components into arrow functions, thus being able to separate them into files. Thus, the implementation of the client application can be done by separating the pages of the website into separate arrow functions and also some components of the page, that contain complex logic. Using Node Package Manager (npm), the use of packages and libraries needed to communicate with the Ethereum blockchain is made easier.

For the communication with the Ethereum blockchain, the previously described web3.js package is used. It was installed using npm, which saved it as a dependency, whose package will be installed automatically when running the npm install command. In order to use it, a web3.js file was created, which initializes a Web3 instance. The Web3 instances takes, as a constructor parameter, a Web3 provider. This is set as window.ethereum, in case MetaMask is installed. MetaMask will use the built in Infura Web3 provider by default. In case there is no MetaMask plugin installed on the accessing browser, I set the provider as an Infura endpoint. Infura lets developers create free accounts and access Ethereum using its provider. To access it, I created an account and a new application, which creates this endpoint for the provider. Next, the contract addresses are defined, which I got after the deployment phase. Deployment will be discussed in more detail in the next section. After this, I got the JSON interfaces of each smart contract, which is needed to create an instance of a contract and call its methods in the client application. These interfaces are obtained when compiling the smart contracts using the Solidity compiler. Lastly, the instances of the smart contracts are created. In order to achieve this, the web3.eth.Contract object is used from the web3.js library. It makes it easy to interact with smart contracts on the Ethereum blockchain. When creating such a contract object, the JSON interface of the respective smart contract is needed and web3 will auto convert all calls into low level ABI calls over RPC, using the provider of the previously created Web3 object instance. Figure 5.15 shows the setup done in the web3.js file, which is needed to communicate with the smart contracts.

```
import Web3 from "web3";
let web3;
if (window.ethereum && window.ethereum.isMetaMask) {
    web3 = new Web3(window.ethereum);
}
else {
    web3 = new Web3('https://ropsten.infura.io/v3/872068728e024a3485c78b9fc66020ca');
}

export default web3;

const actorContractAddress = "0xA0aE44CAE30482eD33C3C36651c0eB3C5991Ce91";
const cuttingContractAddress = "0x9b8a11063aAe16EEC72687DfFfa29bD6BE8b45F2";
const transportationContractAddress = "0xa6434512e68B993E20ACfd9479F7f49539E25115";

const abiActorContract = require('./abi/ActorsRegistration.json').abi;
const abiCuttingContract = require('./abi/TreeCutting.json').abi;
const abiTransportContract = require('./abi/Transportation.json').abi;

export const actorContract = new web3.eth.Contract(abiActorContract, actorContractAddress);
export const cuttingContract = new web3.eth.Contract(abiCuttingContract, cuttingContractAddress);
export const transportContract = new web3.eth.Contract(abiTransportContract, transportationContractAddress);
```

Figure 5.16 Setup for communication with smart contracts

To implement the routing between pages, I have used the react-router-dom package. React-router-dom lets developers define the component that will be loaded for each URL path. It also makes it possible to add some default Components that will be present for each of these pages. Each page for this application will have a header section, containing a navigation bar and a button acting as the connection to MetaMask. This MetaMask connection component will be discussed in more detail later. Besides this, each page will also have a footer component, containing the copyright and the build version of the website. The App.js file is shown in Figure 5.17, showing the routing and the header and footer components of each page.

```
const App = () => {
    return (
        <CustomProvider theme="dark">
            <BrowserRouter>
                <Header />

                <Routes>
                    <Route path='/' element={<Home />} />
                    <Route path='/transports' element={<Transports />} />
                    <Route path='/cuts' element={<Cuts />} />
                    <Route path='/cutters' element={<Cutters />} />

                    <Route path='/transport/:hash' element={<TransportDescription />} />
                    <Route path='/cut/:hash' element={<CutDescription />} />
                    <Route path='/cutter/:cif' element={<CutterDescription />} />

                    <Route path='/car/:car' element={<Car />} />

                    <Route path='/dashboard/:address' element={<CutterDashboard />} />

                    <Route path="/notfound" element={<NotFound />} />
                </Routes>

                <Footer style={{ 'backgroundColor': '#1a1d24' }}>
                    <p style={{ 'textAlign': 'center', 'paddingTop': '10px' }}>© Erik Halmai 2022</p>
                    <p style={{ 'textAlign': 'center', 'paddingBottom': '10px', 'margin': '0px' }}>Build v1.0</p>
                </Footer>
            </BrowserRouter>
        </CustomProvider>
    )
}
```

Figure 5.17 App.js page routing

The MetaMask connection component is part of the Header component. It contains the logic for the connection with a MetaMask wallet. This connection can only be done by having the MetaMask plugin installed on the browser accessing the website. To check whether MetaMask is installed, the window.ethereum and window.etherem.isMetaMask values can be used. The installed plugin injects a global API into websites and these values can be used in order to check if the plugin is installed. In order to get the current account of MetaMask, a request can be made to this API, for the method eth_requestAccounts. Such a request will return all MetaMask connected accounts. Our component contain this logic inside the getAccount asynchronous function, as depicted in Figure 5.18.

```
const getAccount = async () => {
    if (window.ethereum && window.ethereum.isMetaMask) {
        const accounts = await window.ethereum.request({
            method: "eth_requestAccounts",
        });
        const account = accounts[0];
        return account;
    }
    else return '0x0000000000000000000000000000000000000000';
}
```

Figure 5.18 Function for getting the connected MetaMask account

In order to separate complicated logic, I have created separate components for forms and for lists. Forms are used to input new contracts or actors during registration. Lists are used to display the data present in the smart contracts. I have created separate forms and lists for cutting companies, cutting contracts and transportation contracts. Forms contain the logic to send a transaction to the smart contract in order to save the new data to the blockchain, while list components get the array of data from the page component and displays them with a user-friendly design.

The cutting contract form components contains the logic of building a UI friendly form for the creation of a new cutting contract. It provides a model using rsuite's Schema.Model object, where and constraints can be specified for each form group input. You can specify the input type, as string or number types for example, if the input is required and additionally add some patterns using regexes. For the purpose of creating the transaction that saves the new cutting contract to the smart contract, I created the function createCuttingContract asynchronous function. First, the connected wallet is obtained using the getAccounts function of web3.eth. After this a try-catch block is executed. In web3.js there are two functions to interact with a smart contract. The first one is call, which will execute the smart contract method without sending a transaction, while the second one is send, which will send a transaction to the smart contract on the blockchain and execute the function. Inside the try-catch block, in order to get the error message from the require statements, when they fail, I first executed the function with the call function. This call is awaited to finish and if it fails, it will enter the catch block and have the correct revert message from the require statement in the smart contract. Next, I used the send function, which initiates a transaction and it will display errors where the user declined the transaction, for example, or any other errors not related to the require statement fails. On the present version of web3.js, the send function cannot be used to get the failure messages from the require statements. Figure 5.19 shows the createCuttingContract function. The functions for registering new actors, cut request and creation of transportation contracts have a similar implementation.

```
37  const createCuttingContract = async () => {
38      const accounts = await web3.eth.getAccounts();
39
40      try {
41          await cuttingContract.methods.createCuttingContract(web3.utils.asciiToHex(formValue.cif), formValue.agreedNrTrees, formValue.location).call({
42              from: accounts[0]
43          });
44
45          cuttingContract.methods.createCuttingContract(web3.utils.asciiToHex(formValue.cif), formValue.agreedNrTrees, formValue.location).send({
46              from: accounts[0]
47          }).on('error', (e) => {
48              toaster.push(errorNotification(e), { placement: 'bottomEnd' });
49          }).on('transactionHash', (txHash) => {
50              toaster.push(loadingNotification(txHash), { placement: 'bottomEnd' });
51          }).then(result => {
52              toaster.push(successNotification('Cutting contract created'), { placement: 'bottomEnd' });
53              reload();
54              openResultModal(result.events.CutCreated.returnValues.cutHash);
55          });
56      }
57      catch (e) {
58          toaster.push(errorNotification(e), { placement: 'bottomEnd' });
59      }
60
61  };
```

Figure 5.19 Function for the creation of a new cutting contract

This component uses the forms provided by rsuite. When submitted, the form will execute the previously described function for the creation of cutting contracts. It will also check if the values of each form group obey the rules provided in the model object. When a change occurs, the formValue object is updated, which will contain the values inputted into each form group. Figure 5.20 shows the returned html of the CutForm component.

```
63      return (
64          <Form fluid onChange={setFormValue} formValue={formValue} model={model}>
65              <Form.Group controlId='cif-9'>
66                  <Form.ControlLabel>CIF</Form.ControlLabel>
67                  {
68                      !!givenCif ?
69                          <Form.Control readOnly name="cif" />
70                          :
71                          <Form.Control name="cif" />
72                  }
73              </Form.Group>
74              <Form.Group controlId="agreedNrTrees-9">
75                  <Form.ControlLabel>Agreed nr. trees</Form.ControlLabel>
76                  <Form.Control name="agreedNrTrees" />
77                  <Form.HelpText>Required</Form.HelpText>
78              </Form.Group>
79              <Form.Group controlId="location-9">
80                  <Form.ControlLabel>Location</Form.ControlLabel>
81                  <Form.Control name="location" />
82                  <Form.HelpText>Required</Form.HelpText>
83              </Form.Group>
84              <Form.Group>
85                  <ButtonToolbar style={{ 'float': 'right' }}>
86                      <Button appearance="primary" onClick={submitForm}>Submit</Button>
87                      <Button appearance="subtle" onClick={closeFormModal}>Cancel</Button>
88                  </ButtonToolbar>
89              </Form.Group>
90          </Form>
91      )
92  }
```

Figure 5.20 CutForm component html

The function to get all the cutting contracts from the smart contract is implemented in the Cuts.js page. This component implements the whole page listing all the cutting contracts. There are other pages, such as a separate page for each cutter company, where only the cutting and transportation contracts belonging to that company are displayed, but these have a similar logic. They just the other mappings that have been created in the smart contracts. This function is implemented inside the page component, and not in the list component, because by doing so, a refresh function can be implemented, calling the get function again after an insert was performed. This way newly inserted contracts will be available right after an insert was executed. The Figure 5.21 shows the implementation of the function responsible for getting all cutting contracts.

```
64  const getCuts = () => {
65      cuttingContract.methods.getAllContractsCount().call()
66          .then(count => {
67              for (let i = 0; i < count; i++) {
68                  cuttingContract.methods.contractHashes(i).call()
69                      .then(contractHash => {
70                          cuttingContract.methods.contractInfo(contractHash).call()
71                              .then(contract => {
72                                  actorContract.methods.companyInfo(contract[0]).call()
73                                      .then(info => {
74                                          const cut = {
75                                              hash: contractHash,
76                                              company: info.name,
77                                              agreedNrTrees: contract[1],
78                                              location: contract[2],
79                                              startTime: contract[3],
80                                              nrCutTrees: contract[4]
81                                          }
82                                          setCuts(cuts => cuts.concat(cut));
83                                          setSearchedCuts(searchedCuts => searchedCuts.concat(cut));
84                                      });
85                              });
86                      });
87              }
88              setCutsFetched(true);
89          });
90  };
```

Figure 5.21 Function for getting all cutting contracts

To read the data from the smart contract, it is enough to do a call to the functions. First the function gets the count of how many contracts there are. This is because the array of contracts cannot be returned as a whole, so elements need to be read one-by-one. Next, for each element the contract's hash is read and the contract's info after that. To get the cutting company's name, the TIN if used, which is inside the cutting contract struct. This struct is returned as a JSON object. Next, the cut is built, which is a record containing the data that will be displayed about each cut. After which it is placed inside an array containing all the cuts. This array is a useState variable, thus it will refresh the component when its contents change.

The final HTML containing the components for the Cuts.js page is presented in Figure 5.22. The CutForm component is used as a modal, that can be opened by clicking on a button. This button is only shown in case the connected wallet's address is a forester. If the creati on process succeeds, a QR Code is generated, whose value is the URL of the newly created cutting contract. Thus, this code can be screenshotted and later scanned by the authorities to easily check the validity of cuts. The CutsList component gets the searchedCuts array as a prop. These are the array of cutting contracts that contain the search bar's input results. If there is nothing searched for, this array will contain all the cutting contracts.

```
107     return (
108         <>
109             <h2 className={styles.pageTitle}>Cutting contracts</h2>
110 
111             <div className={styles.addButton}>
112                 { isForester && <Button appearance='ghost' onClick={openFormModal}>+ Create cutting contract</Button> }
113                 <Modal overflow={false} size='md' open={isFormModalOpen} onClose={closeFormModal}>
114                     <Modal.Header>
115                         <Modal.Title>Create cutting contract</Modal.Title>
116                     </Modal.Header>
117                     <Modal.Body>
118                         <CutForm closeFormModal={closeFormModal} reload={reload} openResultModal={openResultModal} givenCif='' />
119                     </Modal.Body>
120                     <Modal.Footer />
121                 </Modal>
122 
123                 <Modal overflow={false} size='md' open={isResultModalOpen} onClose={closeResultModal}>
124                     <Modal.Header>
125                         <Modal.Title>Created cutting contract</Modal.Title>
126                     </Modal.Header>
127                     <Modal.Body>
128                         <QRCodeCanvas value={clientUrl + '/cut/' + resultHash} className={styles.resultHash} />
129                         <p>Contract hash: {resultHash}</p>
130                         <p><i>Please make sure to screenshot the QR code!</i></p>
131                     </Modal.Body>
132                     <Modal.Footer />
133                 </Modal>
134             </div>
135 
136             <div className={styles.search}>
137                 <InputGroup inside>
138                     <Input placeholder='Search for cutting contract hash' onChange={searchHandler}/>
139                     <InputGroup.Addon>
140                         <Search />
141                     </InputGroup.Addon>
142                 </InputGroup>
143             </div>
144 
145             <div className={styles.content}>
146                 {
147                     cutsFetched ?
148                         <CutsList cuts={searchedCuts} />
149                         :
150                         <Loader size='lg' backdrop content="loading..." vertical />
151                 }
152             </div>
153         </>
154     )
155 }
```

Figure 5.22 Cuts.js page HTML

5.4. Deployment

Both parts of the application, the smart contracts and the website, are publicly available. It the web application has a public link, while the smart contracts can be accessed via RPC requests. Figure 5.22 shows the deployment diagram of the system.

The web application was deployed on Heroku. Heroku is a cloud platform as a service supporting several programming languages. It is one of the first cloud platforms, being developed in 2007. It supports Node.js application, thus making it perfect for the web application. Using a free account, Heroku can be used to host the website. The steps to deploy the web application will be described in the User Manual chapter.

The smart contracts are deployed on the public Ethereum test network called Ropsten. Ropsten is the oldest test network of Ethereum and its purpose is to deploy and test applications before deploying them to the main Ethereum network. Test network simulate the behavior of the Ethereum blockchain, but do not use real ether. Each test network has its own ether. For Ropsten it is called ropsten ether and it can be gained using faucets. Faucets are platforms which mine and generate new Ropsten ether and give it away to developers for free, in order to test their application. For the deployment, the truffle package was used, which lets developers configure the network and blockchain they want to deploy on.

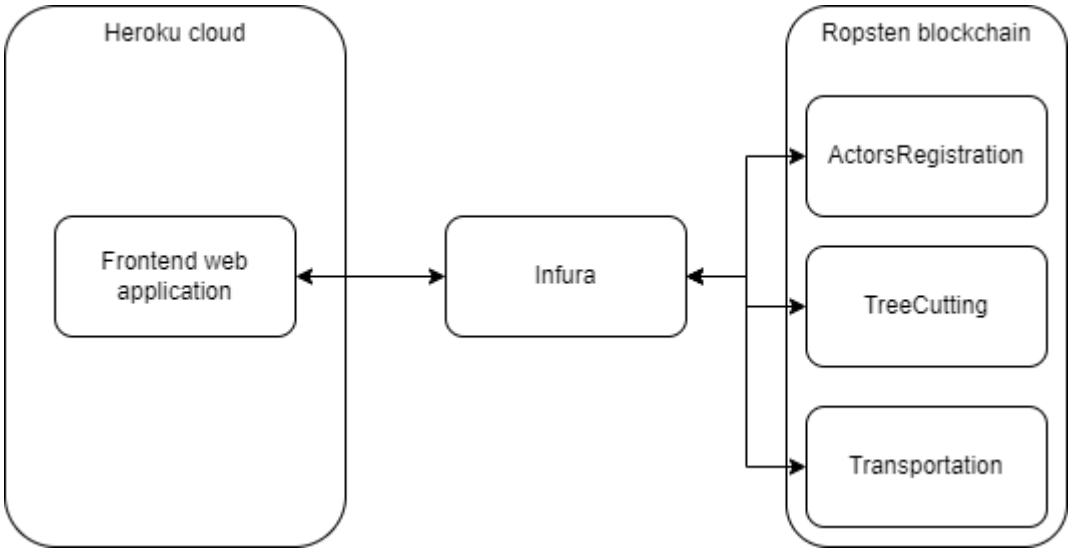


Figure 5.23 Deployment diagram

Chapter 6. Testing and Validation

In order to establish if the solution presented in this thesis accomplishes the objectives presented in the previous chapters, the normal flow of the application was put through tests. Furthermore, some malicious behavior was simulated. This behavior tests if the system can be tricked, in order to be able to continue illegal tree cutting and transportation. These test if there are any edge cases that were missed during development and makes sure that the application is as temper proof as possible.

Firstly, the normal flow of the application was tested. The retrieval of data from the blockchain was successful. All data can be viewed on the website. Additionally, data inside the special mapping, linking data parts together, also functions as expected. Figure 6.1 shows the retrieval of the information and the cutting and transportation contracts of the company Timberwolf Cluj SRL.

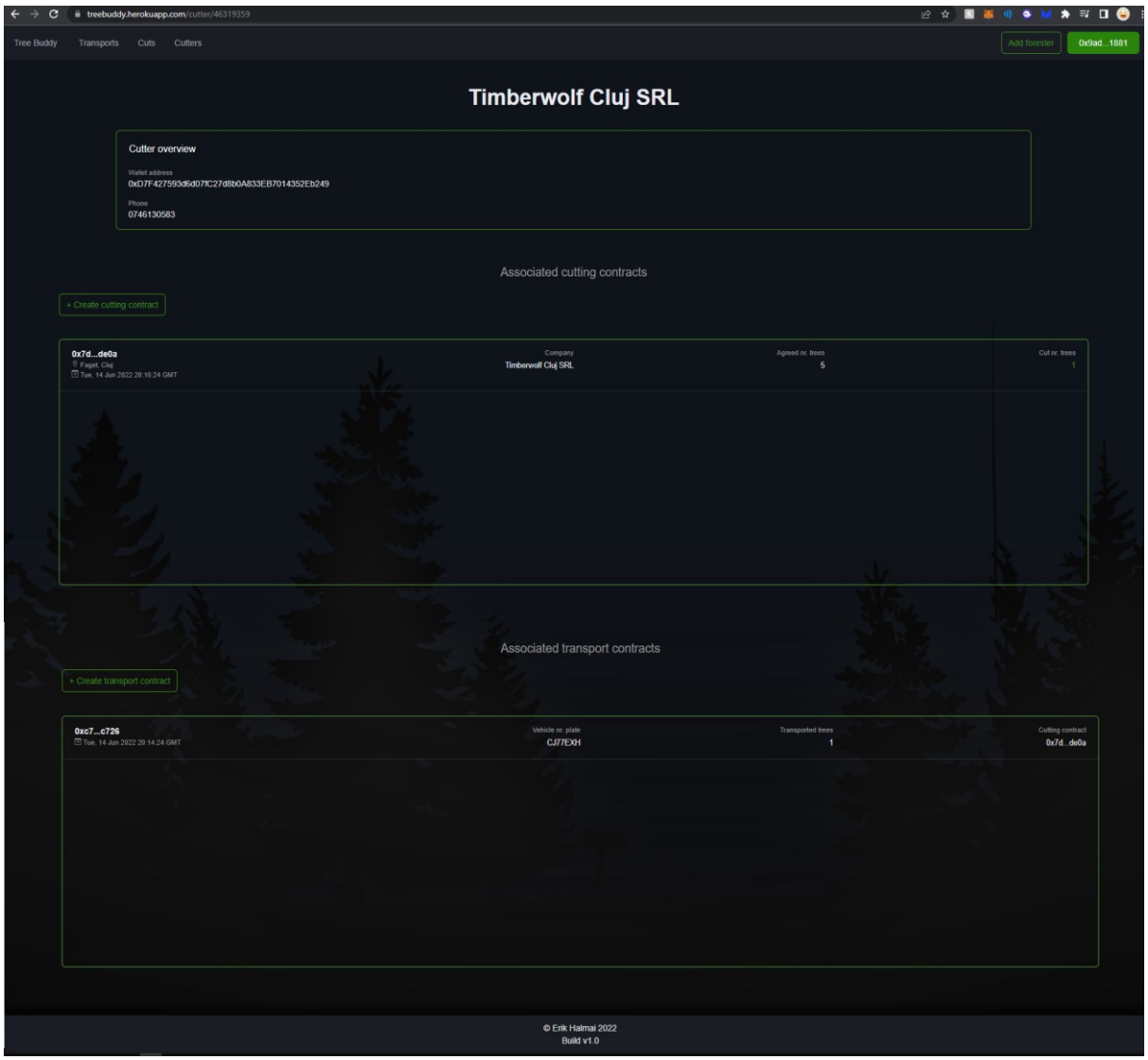
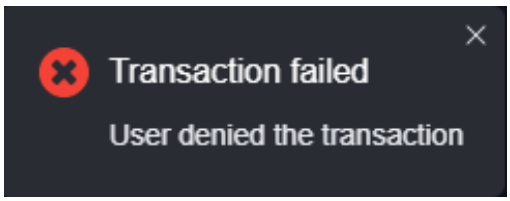


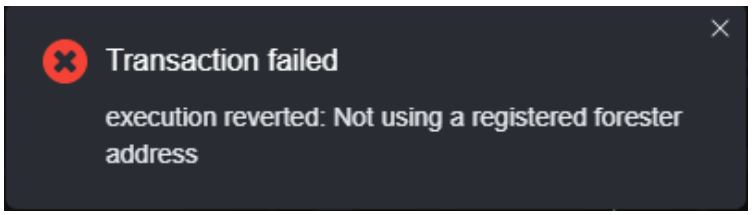
Figure 6.1 Contracts and information retrieval of cutting company

When saving data, if a forester wallet is connected, like in Figure 6.1, the create buttons will appear for the pages. These open the modals containing the previously presented forms. When creating any new input, first the require statements are verified and due to the call function that was implemented before the send one, the revert messages can be returned using user friendly notifications. These revert messages make sure that the rules set by the application are followed, thus making additional or illegal wood cutting and transportation is not possible. The checks done for each transaction are the following:

- Forester registration:
  - The transaction needs to be sent by a registered forester wallet.
- Cutter company registration:
  - The transaction needs to be sent by a registered forester wallet.
  - The provided TIN must belong to an unregistered company.
  - The provided wallet address must not be used by another already registered company
- Cutting contract creation:
  - The transaction needs to be sent by a registered forester wallet.
  - The provided TIN must belong to a registered cutting company.
- Cut request:
  - The transaction sender's wallet address must be the same as the wallet address of the company associated with the cutting contract.
  - The cutting contract must be valid/registered in the system.
  - The number of trees cut so far for the contract must be smaller than the number of trees agreed to be cut for the contract.
- Transportation contract creation:
  - The transaction needs to be sent by a registered forester wallet.
  - The cutting contract must be valid/registered in the system.
  - The new number of trees transported must be smaller than the number of trees cut for the cutting contract.

Figure 6.2 shows some of the error messages that can happen in case the require statements fail when creating a new transportation contract. These notifications appear at the bottom right of the page. These errors show that it is not possible to create transportation contracts without being connected to a registered forester wallet, and using a valid cutting contract, which has trees that have been cut, but not transported yet. Such error messages are displayed in case of all transactions that fail due to a require statement or the user denying the transaction from the MetaMask window.





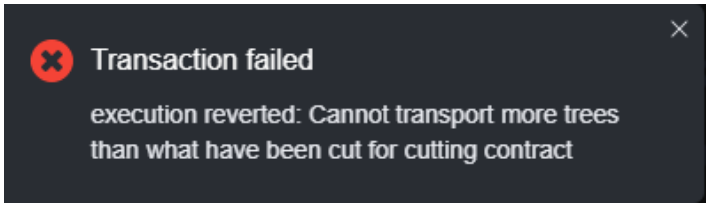


Figure 6.2 Possible error messages when creating new transportation contracts

In case the transaction of creating a new transportation contract succeeds, the QR Code will be generated and the contract will be saved, having the necessary links to the car's number plate and to the cutting contract. Figure 6.3 shows the input for the creation of a transportation contract.

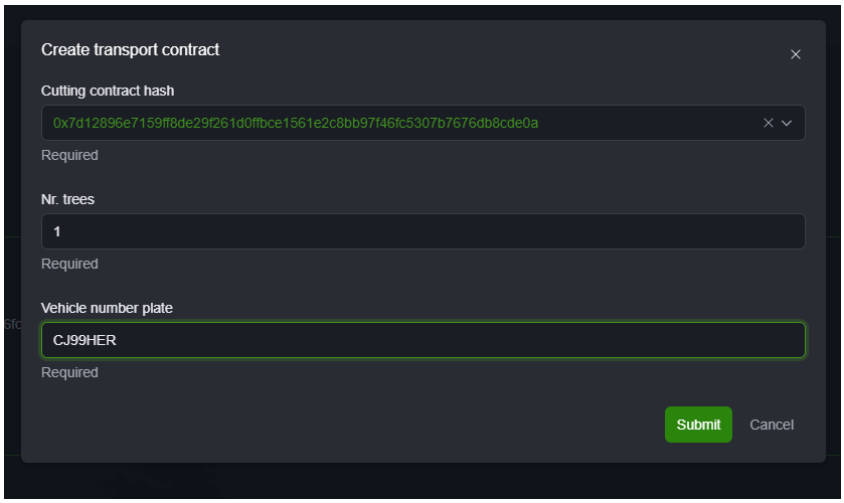


Figure 6.3 Transportation contract creation input

After submitting the input, the MetaMask plugin window is opened, where the user needs to validate the transaction. The transaction is initiated and when it was confirmed, the newly created transportation contract's QR Code is generated and the page is refreshed, showing the new transportation contract in the list.

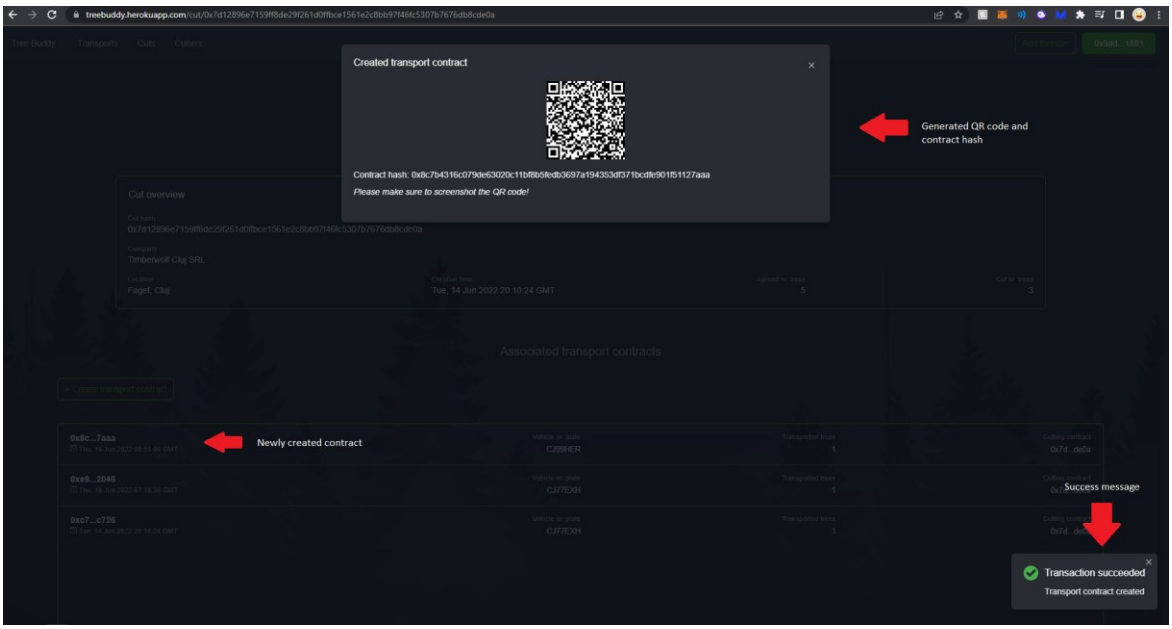


Figure 6.4 Results of successful transportation contract creation

For testing purposes, the Ropsten Ethereum test network was used. Ropsten simulates the Ethereum main network blockchain, thus resulting in similar gas fees for the execution of smart contract functions. The average values for gas used for each of the transactions are the following:

- Forester registration: 46,000
- Cutter registration: 105,000
- Cutter contract creation: 208,000
- Cut request: 39,000
- Transportation contract creation: 319,000

Chapter 7. User's manual

7.1. System requirements

The final system is a website application, that can be run on any web browser. The website can be accessed on computers, independent of the operating system, or mobile phones. In order to view and monitor the data, there are no more requirements than having a web browser on the computer or phone accessing the website. On the other hand, for foresters and cutter companies to be able to connect their wallet to the application and send transactions to the smart contract, a computer must be used. The web browser must have the MetaMask plugin installed with an Ethereum wallet created.

In order to deploy the application, Node.js must be installed on the computer. It is used to install the modules and packages needed to deploy the smart contracts. Git is also needed, in order to clone the application locally.

7.2. Deployment

1) Log into GitHub.
2) Fork the application's repository: https://github.com/HalmaiErik/tree-buddy
3) Git clone the forked repository.
4) Run *npm install* in order to install all the modules of the application.
5) Create an account and a new application on Infura. It will provide a new endpoint, which can be used by the application. Use the Ropsten network endpoint for testing purposes.
6) Install the MetaMask plugin on your browser and create a new wallet. Switch the network to Ropsten Test Network.
7) Get some Ropsten ether on the wallet from faucets online.
8) Edit the project's truffle-config.js file. Modify the pk variable to your wallet's private key. Read it from a file that is added to gitignore if you plan to use the or add real ether to it.
9) Run the truffle *migrate –network ropsten* command to deploy the smart contracts to the Ropsten blockchain
10) After deployment take the contract addresses of each contract and place them in the web3.js file's address variables. Also in the web3.js file change the Infura API link, in the else branch, to your API endpoint.
11) Create an account on Heroku and a new application.
12) Go to the Deploy tab and at Deployment method connect with your GitHub account having the forked repository. Add the repository as the App connected to GitHub.
13) Scroll down to Manual deploy, select the main branch and click on Deploy Branch.
14) After the deployment process completes you can click on Open app at the top of the page to launch the application.

7.3. Application walkthrough

The website's home page consists of a search bar, where the users can search for the number plates, cutting contract hashes, transportation contract hashes or company TINs. Searching for any of these will open the information page for these. At the top of the page is the navbar, having the navigation links on the left and the wallet connection button on the right. The website will try to connect the MetaMask wallet automatically, but if the password is

required, a MetaMask window will open to enter it. When a wallet is connected, the first and last few characters of the wallet's address are displayed on the button, and if it is clicked, a dropdown window will appear with a button to navigate to the wallet's dashboard, showing all the contracts related to the wallet and a button to disconnect the wallet from the website. In case the wallet connected is registered as a forester address, the 'Add forester' button will also appear on the header, which opens a modal to add a new forester wallet address. This header and the footer component, that contain the copyright and build version, are present in all pages. Figure 7.1 shows the website's home page.

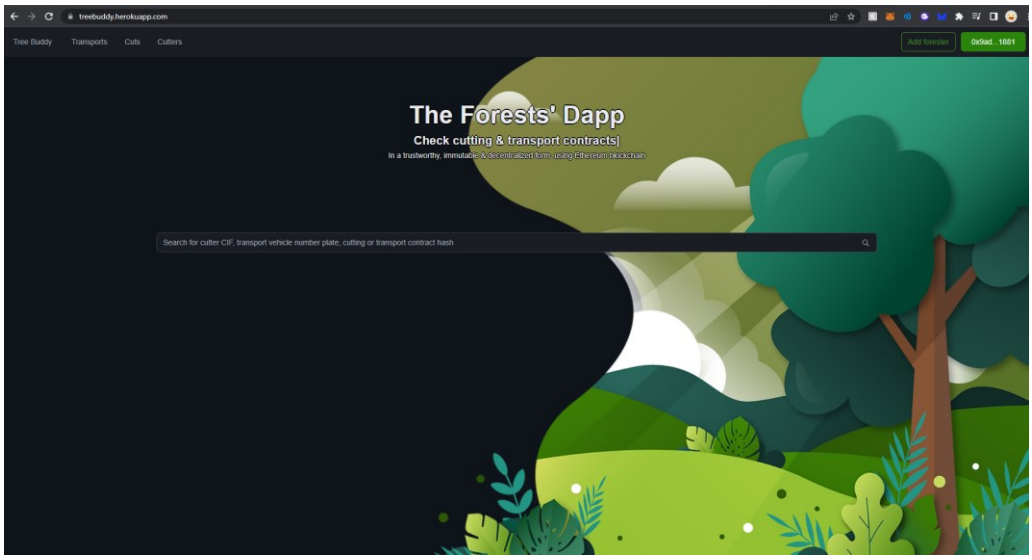


Figure 7.1 Home page

Users can navigate to the Cutters, Cuts or Transports pages using the navbar on the header. The cutters page shows the list of all cutter companies, the cuts list shows all the cutting contracts and the transports page has all the transportation contracts that were ever created. In case the connected wallet is a registered forester address, these pages also have an add button, which open a modal to create new instances of the respective element. In case there's no wallet connected or the address does not belong to a forester, the buttons will not appear. Figure 7.2 shows the cutters page.

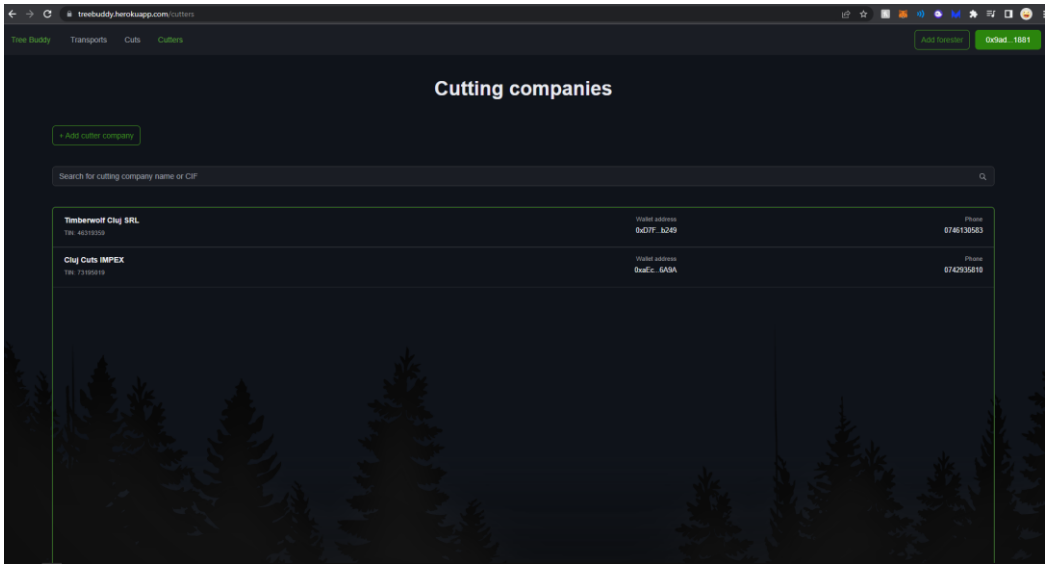


Figure 7.2 Cutters page

In case the application was just deployed, the wallet address that deployed the smart contract will be a forester by default. To add a new cutting company, go to the cutters page and click add cutter company to open a modal containing the form. Input the necessary details and click on submit. The MetaMask window will pop up to accept the transaction. After accepting a notification will appear at the bottom right of the page, showing the transaction was initialized and the having the transaction id. This transaction id can be searched for on the Ropsten explorer, being able to view its status. Once the transaction completes, the website will refresh and the new cutting company will appear in the list. Clicking on any of the companies from the list will navigate to the cutter information page.

Cutting and transportation contracts can be created the same way from the cuts and transports pages. They can also be created from the cutter information page, thus the TIN input for cutting contracts will be filled automatically and the cutting contract hash input, for a new transportation, can be chosen from a dropdown having all the cuts of the company. Contracts also have such information pages, where other data will also be filled to ease creation. For the cutting contract information page, the details of the contract are presented along with all transport contracts linked to it. For the transportation contract the details are presented and the associated cutting contract that was used to create it. When the creation of a contract completes, a modal is opened presenting the hash of it and the generated QR code that can be scanned to get the URL of the new contract. Figure 7.3 shows an example of such a modal. This QR code should be screenshotted and sent to the cutting company associated with the contract. Thus, they can present the code in case they are stopped by authorities. The authorities can scan the code and have all the information about the contract.

Figure 7.3 Contract creation modal

Using the home page search bar, car number plates can be searched, which will show all the transportation contract associated with the car. Thus, making it possible to monitor vehicles seen in public.

In case of the cutting contract information page, if the wallet connected belongs to the cutter associated with the cutting contract, a 'Cut request' button will appear. The address can request a new cut, whose result will be shown by the notification on the button right of the page.

Chapter 8. Conclusions

This thesis describes a solution for the prevention of illegal wood cutting and transportation. The problem of illegal wood cutting has gained major significance in countries like Romania, where billions of euros worth of trees are exploited illegally, each year. The cause of the problem is the bribing of foresters and the opacity of the whole system, which is a result of the centralized nature of it.

The described system is a decentralized application built on the Ethereum blockchain, using smart contracts. Smart contracts are applications deployed on a ledger and can be interacted with using RPC calls. A decentralized solution fixes the problems of opacity and transparency, since all data is saved in the smart contract, which is deployed on the blockchain, thus making it publicly available and monitorable. Furthermore, one of the benefits of blockchain technology is the immutability of data once it is added to the ledger. The implemented smart contracts do not have any functions to modify the data saved, thus making it impossible to change or hide transactions.

The management of cutting is done with the help of two contracts. The cutting contract specifies the number of trees that can be cut using it. It is linked to a cutter company who can use it to request cuts. Before cutting a tree, the cutter needs to request the cut from the system. The second contract is the transportation contract, which is linked to the cutting contract. It specifies the number of trees transported and it is linked to the number plate of the transporting vehicle. As a result, all the transports done by a vehicle can be viewed using the number plate of the car.

Before any data is committed to the smart contract, it is checked to make sure the rules set by the application are met. The application uses two roles, foresters and cutters, these being registered with their wallet addresses. Wallet addresses registered as foresters can create new contracts, while cutters can request cuts for the contract they own. Such checks make sure that no more trees are cut than agreed upon in the contract and no more are transported than the number of trees that have been cut and not transported for the cutting contract. Furthermore, since only trusted actors have the authority to save new data on the blockchain, the integrity of the data is achieved. All data of the application is public, thus authorities can use the platform as a monitoring tool in order to detect anomalies of cuts and transports. The application is open to the public, thus anyone can check and signal any abnormalities seen on the platform. Transports seen in public can be checked, using the car's number plate, thus they can be signaled to the police to do the necessary checks. For every contract, the application generates a unique identifier, which is used to build the URL of the page providing the information about the contract. This URL is encoded into a QR code, which can be screenshotted and later presented to scan using a mobile phone camera, opening the page containing the information.

The decentralized application offers low fees, paid in ether. The functions of the smart contracts are built in such a way as to reduce the costs as much as possible, while still maintaining high speeds and reduce the filtering of data in the client application. This is the price paid for the decentralized nature of the application. Until these fees do not exceed high dollar costs, this price is worth to pay. Soon, Ethereum 2.0 will be deployed, which an upgrade of the existent chain. It will lower gas prices immensely and significantly speed up the transaction speeds, and in the turn the interaction with smart contracts.

To interact with the deployed smart contracts, a user-friendly website application was implemented, which is hosted on Heroku cloud. The website offers an easy-to-use way to interact with the smart contracts, use its services and view the data saved on it. The platform

lets users connect to the platform using the MetaMask web browser plugin. MetaMask makes it easy to create new Ethereum wallets and manage the transactions sent to the smart contracts. The website can be used on any browser and also using a mobile phone, making it possible to enter the website after scanning the QR code of a contract. The mobile phone version does not support the connection with the MetaMask mobile application.

As future improvements and developments, the connection with the mobile MetaMask application could be implemented. This would let foresters and cutters use their mobile phones to input and save new data. Furthermore, a mobile application could be developed, thus making the mobile experience more user-friendly. A second improvement would be the implementation of an overview of all cutting and transport contracts based on a map view. Thus, users could see a map with the contracts and transports from each forest or cutting site.

To conclude, the system described in this thesis offers a solution to the existing problems, found in Romania, caused by illegal tree cutting. The application provides a user-friendly platform for the monitoring and management of tree cutting and transportation, offering transparency and trustable data.

Bibliography

[1] EJOLT: Environmental Justice Organisations, Liabilities and Trade, „Cutting the illegal cutters. Deforestation in Romania," 21 May 2015. [Interactiv]. Available: http://www.ejolt.org/2015/05/cutting-illegal-cutters-deforestation-romania/. [Accesat 19 May 2022]. - Website containing a report presenting the damages of illegal tree cutting in Romania.

[2] Romania Insider, „Romania illegal logging: Authorities censor scientific report that shows volume of wood cut each year," 25 October 2019. [Interactiv]. Available: https://www.romania-insider.com/romania-illegal-logging-report-censored. [Accesat 19 May 2022]. - Website containing a report of a research conducted about illegal tree cutting in Romania.

[3] S. Nakamoto, „Bitcoin: A Peer-to-Peer Electronic Cash System," Bitcoin, 2009.

[4] M. Andoni, V. Robu, D. Flynn, S. Abram, D. Geach, D. Jenkins, P. McCallum și A. Peacock, „Blockchain technology in the energy sector: A systematic review of challenges and opportunities," în *Blockchain technology in the energy sector: A systematic review of challenges and opportunities*, ScienceDirect, 2019, pp. 143-174.

[5] A. Narayanan, J. Bonneau, E. Felten, A. Miller și S. Goldfeder, Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction, Princeton University Press; Illustrated edition, 2016.

[6] A. Nedea și D. Muntean, „Recorder," 17 November 2021. [Interactiv]. Available: https://recorder.ro/singur-impotriva-mafiei-lemnului/. [Accesat 19 May 2022]. - Website presenting a report about the ways in which illegal tree cutting is conducted.

[7] D. L. Chaum, *Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups,* Berkeley: University of California, Berkeley, 1982.

[8] D. Bayer, S. Haber și W. S. Stornetta, „Improving the Efficiency and Reliability of Digital Time-Stamping," în *Sequences Vol. 2*, Columbia University, 1992, pp. 329-334.

[9] G. Becker, „Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis," Ruhr-Universitat, Bochum, 2008.

[10] A. M. Antonopoulos, Mastering Bitcoin. Unlocking Digital Cryptocurrencies., O'Reilly Media, 2014.

[11] M. Milutinović, „Cryptocurrency," *Ekonomika,* vol. 64, nr. 1, pp. 105-122, 2018.

[12] R. d. Best, „Overall cryptocurrency market capitalization per week from July 2010 to May 2022 (in billion U.S. dollars)," Statista.

[13] R. Böhme, N. Christin, B. Edelman și T. Moore, „Bitcoin: Economics, Technology, and Governance," *Journal of Economic Perspectives.,* vol. 29, nr. 2, pp. 213-238, 2015.

[14] J. Song, Programming Bitcoin: Learn How to Program Bitcoin from Scratch, O'Reilly, 2019.

[15] A. Lewis, The Basics of Bitcoins and Blockchains: An Introduction to Cryptocurrencies and the Technology that Powers Them, Mango Media, 2018.

[16] L. Lamport, R. Shostak și M. Pease, „The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems,* vol. 4, nr. 3, pp. 382-401, 1982.

[17] V. Buterin, Ethereum Whitepaper, 2014.

[18] A. M. Antonopoulos și G. Wood, Mastering Ethereum: Building Smart Contracts and DApps, O'Reilly Media, 2018.

[19] G. Michaelson, „Programming Paradigms, Turing Completeness and Computational Thinking," *The Art, Science, and Engineering of Programming,* vol. 4, nr. 3, pp. 2-21, 2020.

[20] „Gas and fees," Ethereum, 20 May 2022. [Interactiv]. Available: https://ethereum.org/en/developers/docs/gas/. [Accesat 22 May 2022]. - Website of the official documentation of Ethereum gas fees.

[21] P. Karhula, V. Vallivaara, N. Lehto și V. Pentikainen, „Asset Tracking With Smart Contracts," în *IEEE Wireless Communications and Networking Conference*, Marrakech, 2019.

[22] S. Dhar și I. Bose, „Smarter banking: Blockchain technology in the Indian banking," *Asian Management Insights,* pp. 46-53, November 2016.

[23] C. D. Antal, T. Cioara, M. Antal și I. Anghel, „Blockchain platform for COVID-19 vaccine supply management," *IEEE Open Journal of the Computer Society,* vol. 2, pp. 164-178, 2021.

[24] M. Wilkes, „client-server architecture," Encyclopedia Britannica, [Interactiv]. Available: https://www.britannica.com/technology/client-server-architecture. [Accesat 24 May 2022]. - Website presenting the Client-Server architectural pattern.

[25] „web3.js - Ethereum JavaScript API," web3js, [Interactiv]. Available: https://web3js.readthedocs.io/en/v1.7.3/. [Accesat 24 May 2022]. - Website of the official web3.js documentation.

[26] Trezro, „Cryptocurrency Transaction Speeds in 2022," Trezro, 13 May 2022. [Interactiv]. Available: https://blog.tezro.com/cryptocurrency-transaction-speeds/. [Accesat 28 May 2022]. - Website presenting a report of the speeds of each major cryptocurrency currently in the market.

[27] Trustnodes, „Ethereum dominates crypto developement says Andreessen Horowitz," Trustnodes, 19 May 2022. [Interactiv]. Available: https://www.trustnodes.com/2022/05/19/ethereum-dominates-crypto-development-says-andreessen-horowitz. [Accesat 28 May 2022]. - Website presenting a report of the dominance of the Ethereum blockchain and Ethereum built decentralized applications in the current market.

[28] „Ethereum Provider API," ConsenSys, 18 January 2022. [Interactiv]. Available: https://docs.metamask.io/guide/ethereum-provider.html. [Accesat 8 June 2022]. - Website of the official MetaMask documentation about Ethereum Providers.

[29] „Solidity documentation," Solidity, 1 March 2022. [Interactiv]. Available: https://docs.soliditylang.org/en/v0.8.14/. [Accesat 13 June 2022]. - Website of the official documentation about the Solidity programming language.