SECURE DELIVERY SYSTEM USING BLOCKCHAIN TECHNOLOGY

Axel Vallin

Computer Science and Engineering, master's level
2018

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

Abstract

Online trading of goods depends a lot on trust between the involved parties. There exist different services that improve the situation for sellers and buyers who do not know each other offered by third parties. This thesis explores the possibility of providing such a service using blockchain technology. Since it was introduced by Bitcoin, blockchains have allowed people to transfer money between each other without involving a centralized authority.

A decentralized system that stores the payment and agreement information of a trade on a blockchain was developed in this thesis. Compared to a third party service, this greatly reduced the possibilities of frauds and improved the overall trust of the system. It also did not have a limitation on how expensive the goods in the agreement could be, which is usually the case for a service offered by a third party. However, blockchain technology have other limitations that reduced the usability of the system. Blockchains are not good at storing a lot of information, it takes time before data is written and each data transfer costs money.

In addition to analyzing the pros and cons of a blockchain based trade agreement, an effort was made to include the logistics company in this contract as well. Here, the limit on data storage and upload speed was an even bigger problem. It is clear that a blockchain solution can not improve every aspect of a service like this one, but the parts that it can improve makes it interesting to certain users.

Contents

1 Introduction

1.1 Background

Purchases on the internet has rapidly increased with the growth of web stores. Many companies that are offering their products and services online are able to reach a larger customer base. This make purchases more convenient for their customers. Others make use of the opportunities that the internet provides by creating a market for trade between individuals. This allows people in different parts of the world to buy things from each other. Unfortunately, this also increases the possibilities of fraud with missing payments or goods not being of the promised quality.

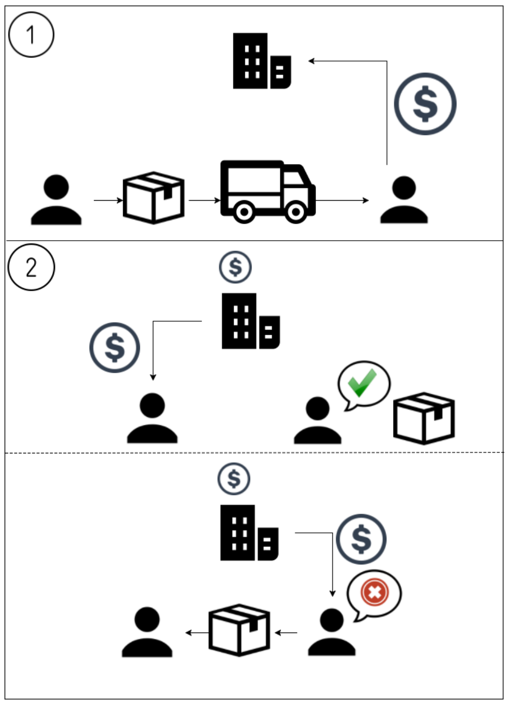
Today, Blocket.se provides a secure merchandise service [1] between two parties which includes payments and the distribution of goods. This will be referred to as the 'original merchandise service'. By using this service, the seller and buyer agrees on the terms of the transfer, the payment and the condition of the goods. If the buyer is satisfied when the package arrives he or she confirms the transfer and pays for the goods, otherwise the package is then returned to the seller and the money is returned to the buyer. In this case, a third party handles the money until the buyer has made a decision.

Improvements of this service can certainly be made. Since the agreement is only between the buyer and seller, a third party should not have to be involved. This forces the mistrusting parties to trust a company instead. Additionally, attacks on the database could be disastrous to anyone using the service. These are two of the things described in Section 2 when discussing the motivation behind the project.

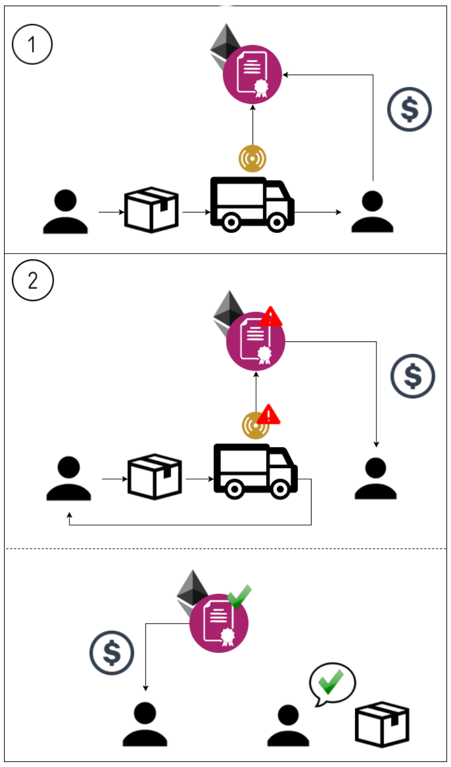
The aim of this thesis is to emulate the original merchandise service but exclude the third party and let the money transfer be handled using blockchain technology, more specifically Ethereum [2]. Blockchain is a technology that has gained a lot of attention in recent years. The most common topic has been the economic value of cryptocurrencies such as Bitcoin [3]. Another aspect of this technology is the software that can be built on top of the blockchain called decentralized applications [2] (or DApps for short). A developer makes use of so called 'smart contracts' to create these DApps on a blockchain platform such as Ethereum. This can for example remove the need of a traditional backend such as a database since the data can be stored on the blockchain instead.

A DApp can therefore not only be used to manage the payment, but also provide more functionality that can be interesting to an actor in this system. The goal of this project is not to compete with established marketplaces such as Blocket.se, but rather piggyback on their success while providing a better experience for their users. By doing so the problem of growing a community is avoided, since getting users to switch to a completely new service can be very hard. This also removes the issue with an entire marketplace on the blockchain. For example, such a service would be very expensive because of the fees that have to be payed for blockchain data storage. By focusing on the agreement between seller and buyer, these fees can be reduced. Other drawbacks such as the large energy consumption needed for the consensus algorithms in blockchains are closely related to this as well.

The main goal of the thesis is to deal with the trust issues between buyer and seller and also between the traders and the logistics company. The possibility of solving the first issue with cryptocurrency payment and agreement information in smart contracts is examined, and sensor data is introduced to improve the second problem. A choice has been made to focus on the software part of such a system, which means that hardware issues of sensors for example are only briefly covered in the discussion part of the thesis. The thesis also discusses problems with a blockchain service. Additionally, it addresses the challenges that comes with both general Ethereum applications and this system particular. A visual comparison between this proposed service and the original one can be seen in Figure 1. Figure 1b shows money and sensor data stored in a smart contract, where the the buyer either returns or keeps the package.



(a) The original service



(b) The proposed service

Figure 1: Comparison between the services.

1.2 Blockchain

1.2.1 Blockchain basics

A blockchain can be seen as a sort of ledger that is distributed among nodes in a peer-to-peer network. The chain contains every transaction, or state change, that has occurred since its deployment. Transactions are grouped into blocks to simplify the validation of the transactions. Because no one is controlling the information stored in a peer-to-peer network, different information on different nodes would be disastrous. Therefore, there needs to be a way to find consensus where the nodes agree on the current state. A well known issue called the 'double spending' problem was not solved for decentralized digital currencies until blockchain and Bitcoin was introduced [3].

In a digital currency without a solution to this problem, it would be easy for anyone to copy their money since they are represented by digital files. These copies can then be sent to other users as payment for a product or service. This is basically the same thing as creating counterfeit money in a physical currency, but it is a lot easier. A centralized system can control that no such thing happens. When the system is decentralized, deciding what transactions should be accepted becomes a lot harder. In a blockchain, transactions are sent to all nodes with the task to add them to the chain.

After checking that a proposed transaction is valid according to the current state of the blockchain, the node adds the transaction to a block not yet part of the blockchain. An algorithm called **proof of work** [3] is run on the block which is needed for the system to reach a consensus. These nodes are called 'miners' and will try to find a number that together with the block can be hashed to a value confirming the block. In the case of the Bitcoin implementation, this hash requires a certain number of leading zeros and can only be found by trying different numbers chosen randomly by each miner. The exact proof of work algorithm used can be different for different blockchains. Once a block is mined, it will be added to the chain. Any solutions to the hashing arriving later will be discarded. Miners will then continue to add build the chain on this block if they think that the block is valid. Otherwise those miners will ignore the block and build on what they think is the last valid one. The chance for a certain node to find the hash first is proportional to its computing power. The more power a node has, the more likely it is to add the next block to the chain.

More nodes joining the network increases the speed at which a hash can be found. To keep a somewhat stable interval between each addition of a block, the difficulty of mining is adjusted. If the last block was added too quickly, the blockchain can change the requirements so that the next block would need an additional zero in its hash in order to be valid. With a growing community, large blockchains therefore consume an enormous amount of power. Many new blockchain solutions, including future versions of Ethereum [2], swap **proof of work** with **proof of stake** [4] to reduce the amount of power required to run the consensus algorithm. The main difference is that a node's chance adding the next block to the chain in **proof of stake** depends on how much money (bitcoin or ether for example) the node has. This makes some people worried that the blockchain is going to be controlled by large companies, effectively making it more centralized. However, buying more computing power is already possible in a **proof of work** system, and some argue that **proof of stake** will actually help make the blockchain less centralized [5].

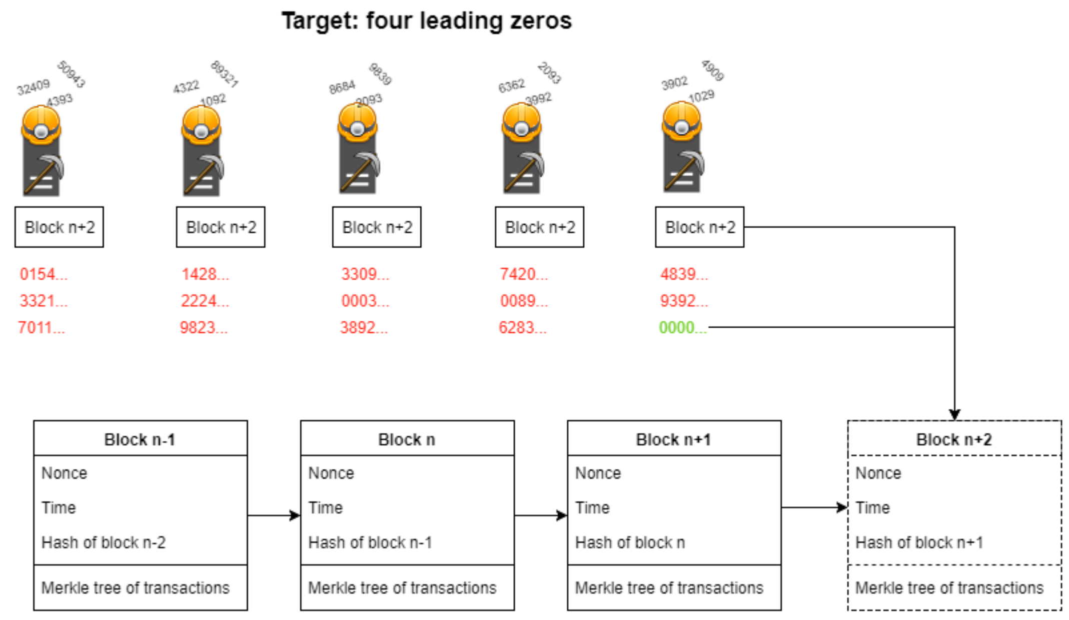


Figure 2: Miners try to find a suitable hash of the next block and a random number.

The contents of each blocks displayed in the mining process in figure 2 are listed below and later elaborated further.

- Nonce: The random number which gives a valid hash.
- Time: The time when the node mined the block.
- Hash of the previous block.
- Merkle tree: The transactions of the block.

A valid block with a nonce, timestamp and hash of the previous block is replicated to every node. The nonce is the random number which gave a valid hash for the block. The block also contains all its transactions which are organized into a Merkle tree [6] (described later). A block arriving to the chain at the same time as another one will create a fork. This means that there are now two chains (originating from the same chain) that are considered correct. New blocks arriving will always add themselves to the longest chain. If there are more than one it can choose either one of them. Since the chance that more than one block arrives at the same time is very small, it is extremely unlikely that multiple chains of the same length survive more than a couple of iterations.

These forks can cause additional problems with double spending. For example, if user A tries to send all their money to user B and all their money to user C, miners might decide differently on what transaction to accept, depending on which one they received first for example. If two miners that have decided on a different order have mined the block at the same time, the transactions to both B and C would be added to the blockchain. Therefore, a transaction is not considered final as soon as it is mined. After a certain number of blocks (five for example [2]) have been added on top of it, it is considered confirmed. Because of the negligible risk that both chains continuously have blocks added to them, once one fork is longer, the transaction in the other fork is considered invalid.

So why can't an adversary in the double spending problem go back to a state before the transaction was added to the blockchain and create a longer fork than the currently active chain? This would potentially result in a new state where the money was never spent. It would not be sufficient to only mine this new block, but all blocks after it also have to be mined to create this longer chain. This is since blocks contain hashes of previous blocks, therefore any changes to a block will change hashes in the next block and so on. Meanwhile, the nodes in rest of the network are adding blocks and making their chain longer. To overtake the length of the current chain, an adversary is required to control more than 50% of the total computing power of every node, eventually creating the longest fork. While this is possible in theory it is too expensive to actually be an issue for large blockchains.

One important property of the blockchain is that while the **proof of work** requires a lot of computing power, checking if a block is valid has to be cheap. To quickly check if all transactions in a block are valid the block contains something called a 'Merkle tree', displayed in figure 3. The leafs of this tree are the hashes of the transactions and each node is the hash of its children. Each member of the blockchain can therefore see if a block in another node has the exact same transactions by comparing the root (effectively a hash of all transaction hashes) of the tree. By stepping down in the tree and comparing hashes the exact faulty transaction (from the one doing the check's point of view) can be determined.

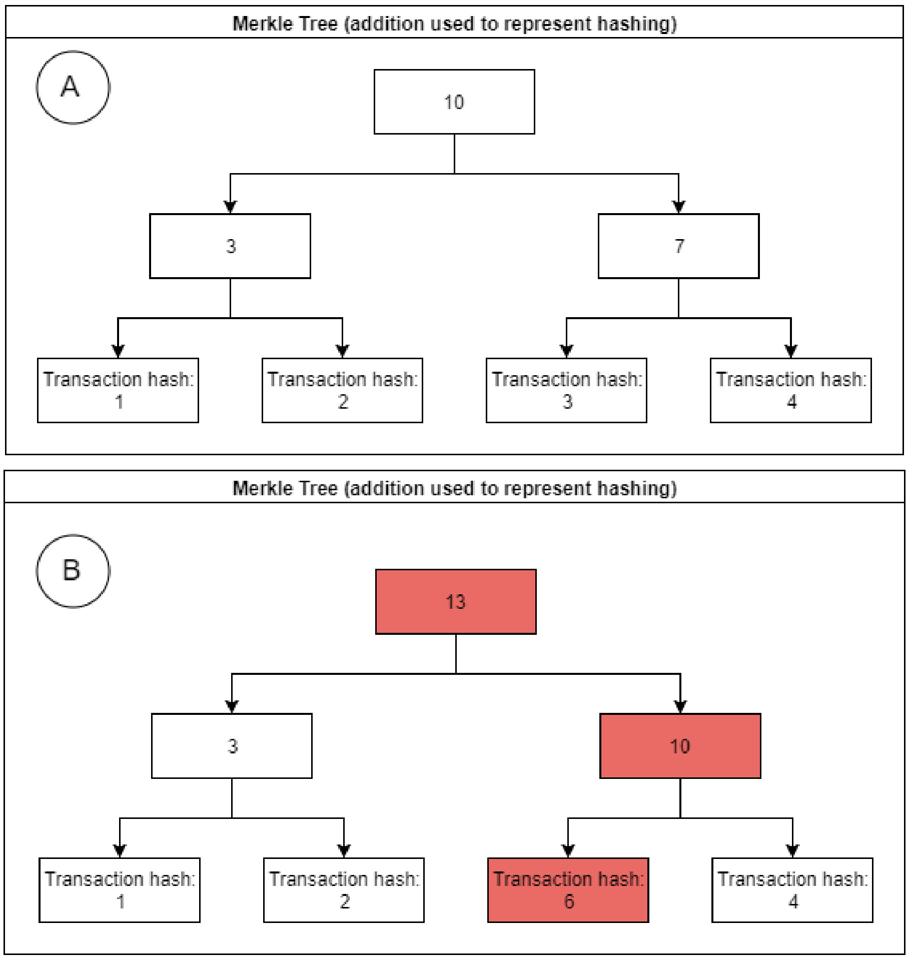


Figure 3: Two Merkle trees. Only the root needs to be checked to see if they contain different transactions.

1.2.2 Blockchain in this project

Why would you want to use a blockchain solution for a merchandise trading service? According to Wust et. al [7], blockchain should only be used if there are multiple entities that do not trust each other. Wust et. al [7] also provides a flowchart to help decide if you need a blockchain solution and what kind of blockchain you should use.

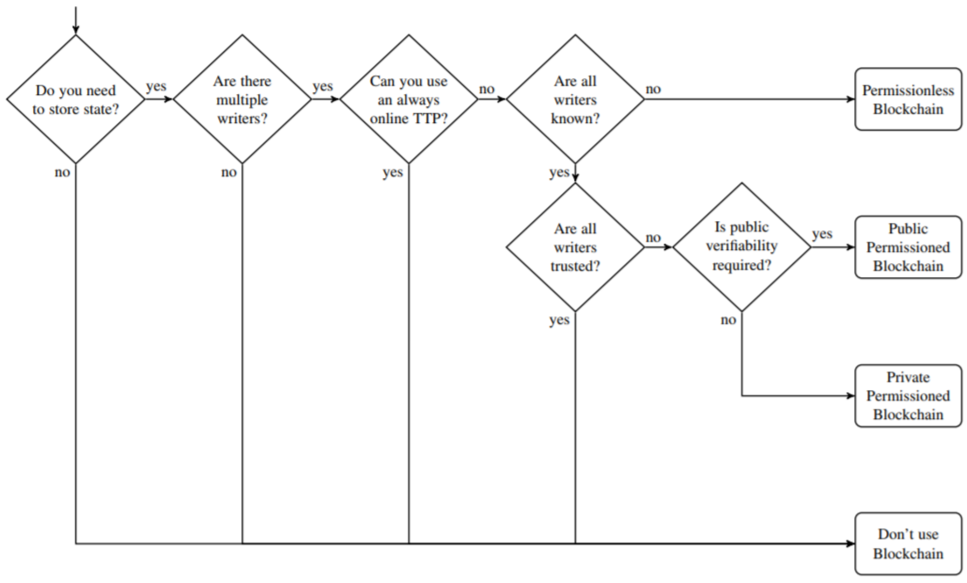

Do you need to store state?
yes
no
Are there multiple writers?
yes
no
Can you use an always online TTP?
no
yes
Are all writers known?
no
yes
Permissionless Blockchain
Are all writers trusted?
no
yes
Is public verifiability required?
yes
no
Public Permissioned Blockchain
Private Permissioned Blockchain
Don't use Blockchain

Figure 4: Flowchart by Wust et. al [7] to determine if blockchain technology should be used.

Bitcoin and Ethereum are examples of permissionless blockchains and Hyperledger [8] is an example of one kind of permissioned blockchain. Short answers to the questions in the flowchart for this particular project is provided below.

Do you need to store state? Yes. The state in this service is the money being transfered and any information about the delivery (sensor data for example).

Are there multiple writers? Yes. Buyers, sellers and sensors are all writers.

Can you use an always online trusted third party? No. To solve the problems with the original merchandise service described in Section 2 the third party can not be used.

Are all writers known? No. Anyone can be a buyer and seller in this system.

Ethereum was selected as the platform for this project since it's the most commonly used permissionless blockchain for DApp development.

1.2.3 Ethereum

Ethereum [2] is a blockchain technology that is designed to run Turing complete code in so called 'smart contracts'. It also comes with a cryptocurrency called **ether** that is used in transactions and can be owned by people in externally owned accounts. The smart contracts are ensured to be running the same code everywhere because of the **proof of work** consensus algorithm, described in Section 1.2.1. Contracts also have their own accounts that work the same way as the externally owned ones, except that the contract accounts also have code that can be run.

Using these smart contracts to create DApps is not very different from other forms of programming. The contracts can be seen as classes that can call each other and for example provide functions that can interact with a user interface.

One of the main differences is that all computations that are part of a 'transaction' and takes place on the blockchain requires 'gas'. Transactions are function calls that change the state of the blockchain (updates values stored in a contract for example), regular 'calls' that don't change the state can be done for free. A transaction therefore needs to be mined (e.g. processed) according to the **proof of work** consensus algorithm to make sure that every node has the same state.

If there were no limitations on the code running on the blockchain, anyone could write a program with an infinite loop that increments a value and someone would have to mine it. This would slow down the process of adding new blocks to the chain, which deters serious applications from being deployed on Ethereum.

Gas is payed in **ether** by the one executing the code and is given as a reward to the miners. It is not described as being payed in **ether** directly because the one executing the code can decide how much he or she is willing to pay for the gas. A higher gas price will make it execute faster since it is prioritized by the miners (they will get payed more for including it in the blockchain). A piece of code can therefore be specified to always cost 1000 gas for example. The actual price for it can still differ each time someone executes it.

2 Problem statement

There exist a number of problems with the original merchandise service. In this section the issues that this project tries to solve are discussed.

2.1 Trust

Since a company currently holds the money until the deal between buyer and seller is complete, both actors are required to trust this third party. This also includes any personal information and details about the deal. It is not uncommon that people are skeptical towards sharing information with and trusting large corporations.

The third party is currently needed because the seller and buyer don't trust each other. It does not remove the trust issues, instead it moves the trust from individuals to companies. This might be sufficient in some cases, but there are definitely room for improvements. An opinion of a company can also be very hard for a person disagree with. The company has a lot more power and money to back up their decisions.

2.2 Reusable system

A secure trade system offered by an online marketplace is usually tied to the company running the site. This means that a new merchandise trading service has to provide their own way of safety for their users. The issues discussed in Section 2.1 is more clear in this case. An established company can rely on their reputation to gain the trust of its users, but if a proven system could be used for trading in different places, this issue could be removed.

For example, it is quite common to sell things on social media such as Facebook. In this case, no one takes responsibility for users being fooled and it is up to them to decide who to trust. It is hard to believe that someone would offer a database solution to secure the trade on this platform. But if an existing software could be easily integrated, people might see it as a viable option.

2.3 Attacks

Hacks directed at the databases handled by a third party will put their customers at risk of losing their money. Even if the database in question is well protected, successful attacks can not be ruled out. If the third party uses a traditional way of storing the agreements for the trades, such a security breach can be devastating for all active trade deals.

2.4 Trace payments

One issue closely related to trust, see Section 2.1, is if someone claims not to have received the payment. This could be the seller or in a traditional system the third party for example. Depending on if some actor is trying to trick someone else to steal their money, or if someone has made a mistake, this might be difficult to solve. Whatever the case, the issue leads to uncertainty for all involved parties.

2.5 Secure sensor data

If a package arriving to a buyer has bad quality goods, the buyer might want to return the package. Similarly, a damaged product that is returned to the seller might result in them wanting some kind compensation. In both these cases, the dissatisfied party can't be certain if their trade partner tried to trick them, or if an accident happened during the transportation.

Information about the transit can help in this case, but if the logistics company is in charge of this information, they could easily provide false data to hide accidents.

2.6 Reduce cost

A company has to make money off a service to have a reason to provide it. In addition to that, there are also different forms of maintenance costs that leads to an increased price for a user. For example, the price for the secure package service provided by Blocket.se [1] starts at 109 SEK (delivery cost included). A solution that removed the third party might be able to reduce the cost for the service.

It could also be made more customizable, as smaller trades might want some part of the service, but are not willing to pay for everything. For trades including expensive goods, the price is not an issue if it is small compared to the value of the trade.

3 Security and threats

Since no one is controlling the smart contracts on the blockchain, the correctness of the code is crucial to avoid disastrous failures. In this section some of the more well known mistakes developers have made are discussed. These need to be addressed along with the previously discussed problem statement in Section 2 to get a working system.

3.1 Race Conditions

The most famous smart contract hack in Ethereum is probably the DAO hack. The DAO [9] was a framework for creating Decentralized Autonomous Organizations on Ethereum. The idea behind such a system is an organization governed by the code in the smart contracts. In a community using this DAO, a user could propose their idea to the rest of the users and initiate a vote. Depending on how this vote turned out, a reward would be given to the proposer. Around \$150 000 000 worth of ether was spent on its crowd funding [10], making it one of the most hyped Ethereum projects ever.

However, shortly after the DAO was deployed on the Ethereum network, an exploit was discovered by a hacker. It is important to remember that this was not because of a security issue in Ethereum, a mistake made by the developers of the DAO made this hack possible.

The hacker discovered that when a payout for a reward was made, the amount that a user was allowed was updated after the actual reward was given, as explained by Phil Daian in his analysis of the hack [11]. Furthermore, a call to the address which should withdraw the reward was made. This meant that the adversary could create a contract based on this DAO framework, start a vote and send a small amount of ether to the vote as reward. Normally, sending a reward to ones own proposal would not make any sense since the payout would increase by the same amount.

Nonetheless, this meant that the hacker was allowed to retrieve the reward after the vote was over. When the DAO called the contract controlled by the hacker to send the reward, the function in that contract was setup to recursively call the DAO withdraw function again. Since the reward size was not updated until the end of the withdrawal, the same amount could be withdrawn again each time it was called.

3,6 million ether was stolen during this attack which used yet another exploit to avoid the block gas limit for transactions. This was a huge blow not only to the DAO, but to Ethereum as a whole. Before the attack, 15% of all ether was part of the DAO [12] and the price of ether dropped from \$20 to \$13.

A decision was made by the Ethereum developers to fork the Ethereum blockchain. In this 'new' network, all stolen ether from the DAO would be available for the original owners to withdraw. This caused huge controversy since one of the main ideas behind blockchain technology is that no one should be able to control the network. Someone claiming to be part of the attack and that it was a team effort argued that it was not a hack at all [13]. Since the rules of the smart contract had been followed, the ether had not been drained by any illegal means.

The Ethereum smart contract best practices [14] provided by ConsenSys [15] talks about two main types of common race conditions: **Reentrancy** and **Cross-Function Race Condition**.

Reentrancy Reentrancy can happen if a call is made to another contract. The function in the other contract could then call the same function again, before the first call finishes. This call might be trusted, depending on how the call is made. If the called contract is known to the caller, it can be determined if that contract makes a recursive call or not.

The DAO contract made a call directly to another address, without knowing the nature of that contract. If this can't be avoided, the contract should update all state variables before such a call is made. For example, the amount of ether that a user can withdraw should be updated before the actual withdrawal is made.

Cross-Function Race Condition A function that makes no calls to an untrustworthy contract can still be affected by race conditions. This can happen if it shares some state with a function that does call an untrusted contract. This can be avoided in a similar way by making sure that all updates are done before such a call is made.

3.2 Transaction Order Dependence

One common issue in smart contracts which is especially interesting for this project is the order of transactions. If two transactions are sent at roughly the same time, one can not be sure which one will be executed first. This can be seen as a special case of race conditions and because of miners' influence might be an even larger problem.

The main difference between transaction order dependence and other race conditions is that they are done in a single transaction. The DAO hack for example could not have been done by sending a large number of transactions that withdrew the reward. One of these transaction would be chosen as the first one and set the possible reward to 0 before any other transaction could run, making the rest invalid.

Luu et al. [16] discusses transaction-ordering dependent attacks as either benign or malicious. In the benign scenario, the random order at which the mined transactions are placed might cause some unexpected behavior. For example, in this project, if a seller accepts a proposal and the buyer updates it at the same time, the seller might find that the accepted proposal is not the expected one.

A malicious attack can be performed by investigating the pending transactions of the blockchain. In the previous scenario, the buyer can see that the proposal is about to be accepted and then update it in his or her favor and hope that the miners put the update transaction first. A number of actions can be taken in order to increase the chance of a certain transaction being mined faster such as setting a higher gas price. While in the benign scenario an unfortunate outcome can occur by pure chance, in this case an attacker actively tries to fool another user.

Because of the nature of this problem being closely tied to the implementation of the blockchain, there is no single solution that can solve these issues in smart contracts. Kalra et al. [17] even considers this issue more of a limitation than a bug. Either way, the harm this problem can cause is very significant and as such the system needs to limit any damage that it can do. Following the logic of the Oyente tool [16], two scenarios where two transactions arrive in different order should either produce the same result or fail in one of the cases.

3.2.1 ERC20 Approve

A specific case of transaction order dependency can be found in the ERC20 Token standard [18]. The 'approve' function is used by the owner of some tokens to let another specific address withdraw a certain amount of tokens. By calling the 'transfer from' function, an address can take tokens from an account that has allowed that address to do so, up to the specified limit.

If the owner wants to change the number of allowed tokens to an address, the 'approve' function needs to be called again. Similarly to the scenario discussed earlier, an adversary can discover that this change is about to happen. By quickly sending a transaction to withdraw the tokens from the contract, this transaction might be mined before the allowance is updated. When the update later happens, another transaction can be initiated by the receiver. If the owner does not notice whats happening in time or is unlucky with the order of transactions, he or she might lose more tokens than expected.

3.3 Denial of Service

In 2016, the King of the Ether Throne game encountered a critical bug only a day after its release [19]. The game worked as follows:

1. A user pays an initial amount of ether to the contract and becomes the first king.
2. Another user overtakes the throne by paying more ether than the previous king.
3. The previous king receives the ether offered by the current king.
4. New kings arrive hoping to make a profit off of future kings (or forever being known as the King of the Ether Throne).

In step 3, ether was sent to the address of the previous king. If this failed, there would be no way for the previous king to claim the ether which would then be stuck in the contract.

A similar issue with different consequences would occur if the contract required that the ether was sent to the address and otherwise reverted the transaction. A user could then, intentionally or not, block the payment, preventing any other user from becoming the new king since the whole transaction would always fail.

3.4 Initialization

Parity [20] is an Ethereum client whose multi-signature wallet has been hacked twice [21], [22]. Since it is an Ethereum client, it can be used to browse DApps or access Ethereum wallets to trade ether. A multi-signature wallet is offered as additional security for Parity users. In regular wallets, if someone gets access to a user's private key for example, they can transfer money from that wallet. To prevent this, multiple private keys can be required to sign a transfer before it becomes valid.

There are no issues with the idea behind multi-signature wallets, but as is usually the case when large amounts of ether is stolen, flaws in the code of the smart contracts allowed this to happen. The target of both hacks was the library that all existing Parity multi-signature wallets made calls to.

3.4.1 First hack

In July 2017, an attacker made use of the default function in smart contracts to steal $31 million worth of ether [21]. This is the function in a smart contract that runs if a nonexistent function is called. Usually, this does not do anything interesting, but the developers of the Parity multi-signature wallet had decided to add some additional code to this function. If there were no ether that arrived with the transaction, they assumed that a call to some library function was made. Since the call ended up in the default function, it did not exist in the multi-signature wallet, but it might have existed in the library and was thus delegated to that contract.

The delegate call method in solidity smart contracts made it look like the transaction came from the last caller. So if a user would call a nonexistent function in a wallet, it would be sent to the library with the wallet as the sender.

The library function which initialized the multi-signature wallets had no protection against being called multiple times. By using the method described earlier, this function could be called by first making the call to any wallet (the attacker would not have to have access to it) and provide the attacker's own address as the only address required to make transactions from the wallet. Any wallet called in this way would make a call to the library to update its owners and the attacker could then withdraw all the money.

Luckily, the attack was discovered quickly and 'white hat hackers' (hackers that are not malicious) were able to save the remaining ether from being stolen. Since the contracts where deployed on the blockchain with no way of altering the code, the only way of stopping the hacker was by using the exact same method as the attacker to drain the remaining accounts. They were then able to return the ether to the correct owners.

3.4.2 Second hack

While the first hack netted a huge gain for the adversary, the second time the same contract was hacked, all funds that relied on the multi-signature library was frozen [22]. The story about this hack is quite peculiar, and no clear motive is known. The attacker claims that it was a mistake and created an issue on github about the flaw in the contract [23].

Once again, the exploit was because of the initialization function, but this time it was not a wallet that was initialized, but the library itself. According the hacker, they were a newbie in blockchain technology and were experimenting with different contracts on Ethereum that were deployed by larger companies. By making a call to initialize the library, they became the owner of the contract. This was not really an issue since the library was not in charge of any money. The problem was that the owner could kill the contract, breaking all wallets dependent on the library.

By calling these functions, the hacker froze $300 million in the multi-signature wallets with no way of getting them out of there. Because this required two different functions to be called, and since the function to kill the contract was called right after the initialization function, most people believe the attacker knew what was going to happen and is lying about being a beginner that accidentally made a mistake.

4 Related work

4.1 Canya

Canya [24] is an Ethereum based peer-to-peer marketplace for services. It soft launched in Australia in March 2017 and plans to launch globally during the first half of 2018. The application allows for people, a plumber or gardener for example, to register and offer their services to other users.

The payments are handled by a blockchain to counteract fraud and safely manage money without the interaction of a third party. A provider proposes an invoice which can be rejected by the client. When they agree, money is sent to the smart contract. Canya supports milestone and recurring payment, meaning that money will be automatically sent to the provider after a certain goal has been met or some amount of time has passed. A client can then cancel the payment at any time. The provider will then have gotten paid for all the work that has been made, but the remaining money in the contract will return to the client.

Since no company is involved, no administrative costs exists. Because of this the provider can earn more money while still offer a cheaper cost to the client. This is one of Canya's biggest advantages.

Canya also provides a way to deal with the price volatility of cryptocurrencies. Since the client has to enter money into the smart contract when the agreement is made and it is not paid out to the provider until the job is done, the value of the payment might have changed. A 'store-of-value' in Canya will allow for a hedge contract that can either collect the extra value of the payment or, more importantly for the client and provider, pay the extra money needed to reach the original value.

The payment process and agreement made by client and provider resembles the idea behind the project described in this report. Differences are mainly caused by the fact that Canya deals with services and not goods. Payment can therefore rarely be made incrementally in the same way since a buyer usually buys one thing and is either satisfied or want all their money back. This can lead to one or both parties to be unhappy with the outcome and there need to exist some support for solving disagreements.

Canya also establishes an entire platform for this marketplace which introduces challenges regarding the growth of the community. That is avoided in this project by using existing platforms and focusing on the payment and delivery. The transportation is of course nonexistent in Canya since they're dealing with services.

4.2 ShipChain

ShipChain [25] proposes a solution to problems regarding transportation of goods across the globe. A blockchain is used here to replace multiple centralized services with a decentralized one. With long transports comes different methods of shipping which often leads to uncertainty for the customers. The information about the transport can instead be stored on the blockchain from when the product leaves the factory or farm for example, until it reaches the customer. This will increase the transparency of the system.

To achieve this the people behind ShipChain make use of smart contracts developed for Ethereum that handles the data gathered during the transportation. ShipChain calls this service the 'side-chain' and can be initiated by anyone to use. Contracts are finalized when the product has passed all waypoints on the transport route. During the transportation, all involved parties will have access to the smart contract information so that they can review it.

Expensive brokers are currently in charge of the transition between different parts of the transport. This is not only a cost that reduces the revenue for carriers, it also makes it harder to track the loss of goods. ShipChain wants to simplify this process by providing a service for carriers to find the best available next shipment for a delivery.

A decentralized broker system is developed to provide this service. ShipChain will create the first web application based on this system but anyone can develop their own frontend using the same blockchain backend. By letting the customer specify their needs, the system can find the best suitable shippers.

A centralized version of this system is certainly possible, but that would make it the new global broker and probably not reduce the cost in any significant way. The transparency would also disappear since the system could hide any information it doesn't want to show. This leads to the same trust issues that are present today and is therefore not a good alternative.

ShipChain focuses on the tracking of products that is being transported and handled by many different companies before reaching it's final destination. Creating a decentralized service that keeps track of data during the transfer is closely related to parts of this project. The main difference is that it focuses on large companies and is not designed for trade between individuals. It does not provide a solution for secure payment without involving a third party company.

Furthermore, each delivery in this project is unique. Therefore each contract will be different and the data provided is never the same. The agreement between buyer and seller as well as solving disagreements are important aspects of this project that is not touched by ShipChain.

5 User stories

A system was developed based on the problem statement in Section 2 and with challenges such as the ones described in Section 3 in mind. Ethereum was used as the backend and stored the agreement contracts including the payment along with any sensor information that could be gathered during the transportation. A simple web page frontend was created for the users to interact with.

Different scenarios about the potential use of this system is discussed in this section. These include different types of traders as well as different outcomes of the agreement.

5.1 Basic agreement

Thomas wants to sell a pair of gloves and puts up an advert on an online marketplace. Since he does not trust strangers he wants to make sure that he is not tricked by the buyer. He also finds it difficult to trust a third party as he has had bad experience with large companies fooling him in the past.

He therefore offers this blockchain based solution for payment and agreement storage. Hasse who is interested in the gloves but has no idea of who Thomas is likes this service compared to a traditional alternative. After he and Thomas have agreed on the deal, he sends the payment to the blockchain. When the gloves arrive and Hasse is satisfied, the payment is automatically transferred to Thomas.

5.2 Expensive trades

After having experienced a successful agreement carried out by the service, Hasse decides to use it again a few months later when he is going to sell half of his very valuable stamp collection. Since he is worried that water might damage the stamps, he wants to add a humidity sensor to the package.

Hasse includes the sensor information in the agreement and sends the package to Kajsa who wants to add to her collection. By including the sensor data in the contract, Hasse and Kajsa can be certain than no one tampers with it afterwards to cover something up. Kajsa receives the package and can see that no violation regarding humidity has occurred.

5.3 Damaged goods

When Hasse later decides to give up on the stamps altogether, Kajsa buys the rest of his collection. They decide on using the same service and add a sensor exactly as before. This time, Kajsa is not happy with the condition of the stamps and sends them back.

Hasse, as he receives the returned package, agrees with Kajsa that an accident has happened during the transport and the sensor data confirms that water has come into contact with the stamps. When the logistics company provides compensation for the damaged goods, both Hasse and Kajsa will get their money from the service.

6 Method

This section describes the method used to develop and evaluate the system. It also shows roughly in which order the different parts of the project was carried out.

The early stages of the project contained a lot of theoretical studies, both regarding the blockchain technology and the development of decentralized applications. This included tutorials on how to set up an environment where the system could run. Tools and frameworks were investigated, most of which did not have more than a couple of alternatives.

Agile development was used during the implementation process which consisted of a few larger sprints with different focus. Documentation of the system, i.e. the report, was done in parallel.

Sprint 1 An initial design for the service was proposed and some experimental contracts were developed. A basic user interface was introduced as well. The experiments showed a lot of promise for the future as well as issues that had to be addressed.

Sprint 2 When an analysis of the larger obstacles in the implementation process had been made a more thorough design was made. The desired functionalities were added to the contracts. Some reworks of the contracts had to be made as problems were encountered and better solutions were discovered with more experience.

Sprint 3 The security of the contracts were examined and the functions were updated accordingly. To check that the service worked as expected, a number of tests was written. This included the things that should not be allowed by the contracts, which tested the system security.

Sprint 4 The user interface used for the early experiments was updated to work with the more complete system and to have a cleaner look. A simulation of the logistics was developed in collaboration with Maxim Khamrakulov which worked on a similar software but with a database instead of a blockchain.

Sprint 5 Some optimizations were made regarding gas usage to improve the system. This required other parts such as the tests and user interface to be updated.

Sprint 6 Finally, the evaluation of the system was carried out.

7 Solution

This section describes the proposed solution to the problems described in Section 2 as well as the implementation of the software part of the system. The application consisted of smart contracts written in solidity and the structure of those can be seen in Section 7.2.

Starting with Section 7.5, the functions of the system are described more thoroughly and some techniques that were used to solve various problems during the implementation are discussed. These functions were tested with full coverage unit tests.

A frontend written in javascript was created to show the usage of the system. A simulation, also in javascript, was used for the transport and sensor data. Most of the functions in the smart contracts were supported on the frontend as well. Event subscriptions were left out because of problems with a couple of choices of frameworks and tools. This meant that the functions dependent on events were not included.

No tests were written for the frontend, it was instead manually tested to see how it handled different scenarios. This was done because it was mainly used for demo purposes, anyone wanting to include the system in an online marketplace could potentially develop their own frontend for it. That being said, an effort was made to make it as complete as possible. The frontend was also used to test the gas usage for different parts of the system, which resulted in a number of updates of the contracts during the implementation process to reduce this cost.

7.1 State machine

Various functions were supported by the system at different points during the agreement. This can be seen as a state machine where for example the seller accepting a buyer's proposal or the goods being delivered changes the state.

Finding a logistics solution for the package transfer is made in a traditional way outside of the blockchain. Therefore, in this thesis, the logistics company is assumed to enter the agreement contract once they agree to transport the goods. As such, they have a limited representation in said state machine. In Section 10.1, an increased involvement of the logistics company is discussed.

A different thesis work was done in parallel by Maxim Khamrakulov to analyze the differences between a blockchain and database solution for secure merchandise trading. To help compare the two solutions, the state machine was designed through combined effort. The explanation and visual representation of the machine was written by Maxim and can be seen below.

7.1.1 Deterministic finite automaton

In order to visually represent the schematics of the states of a given agreement and relations between those states, a special type of Turing Machine was constructed, called a deterministic finite automaton (DFA). A deterministic finite automaton, in general, is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states.
- Σ is an alphabet called the input alphabet.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the starting state.
- $F \subseteq Q$ is the set of accepting states.

Using the definition above, a DFA, denoted as M, was constructed. The set of states Q consists out of variations of `state` parameter. A drawback which is associated with DFAs is that they do not possess memory. Thus, some of the options in `state` are duplicated, depending, whether the `violated` boolean parameter is `True` or `False`. The set of states Q is listed in Table 1. The transitions are triggered by a range of agreement-related events, which construct the input alphabet Σ. These events are also listed in Table 1.

State $\forall q \in Q$	**Description**
CREATED (q_s)	Contract is initiated with the seller's terms, awaiting a buyer's proposal.
LOCKED (q_l)	The buyer's terms have been accepted by the seller.
TRANSIT (q_t)	The item is being transported by the logistics company.
TRANSIT$_V$ (q_{tv})	Sensors have detected violation of the threshold during the transfer.
DELIV (q_{dv})	Item is received by the buyer (threshold was not violated during the transfer), money is transferred to the third party.
DELIV$_V$ (q_{dv})	Item is received by the buyer (threshold was violated during the transfer), money is transferred to the third party.
RETURN (q_r)	Condition of the goods is poor, item is being returned to the seller (item was not damaged during the initial transport).
RETURN$_V$ (q_{rv})	Condition of the goods is poor, item is being returned to the seller (item was damaged during the transport according to sensor).
RETURNED (q_b)	Return is received by the seller (item was not damaged during the initial transport).
RETURNED$_V$ (q_{bv})	Return is received by the seller (item was damaged during the initial transport).
CLERK (q_j)	Seller did not accept the return, conflict request is initiated, contract is under the clerk review.
CLERK$_V$ (q_{jv})	seller did not accept the return, contract is under clerk review (logistics company's responsibility is investigated).
INACTIVE (q_a)	Contract is not finalized.
COMPLETED (q_c)	Buyer is satisfied, money is transferred to the seller.
Event $\forall e \in \Sigma$	**Description**
e_{prop} (e_p)	Buyer sends a proposal.
e_{accept} (e_a)	Proposed terms are accepted by the seller.
e_{decline} (e_d)	Proposed terms are declined by the seller.
$e_{\text{s\_post}}$ (e_{sp})	Seller takes the item to the service point of the logistics company who sends the item.
$e_{\text{b\_deliver}}$ (e_{bt})	Goods are delivered to the buyer's service point
$e_{\text{s\_deliver}}$ (e_{st})	Goods are delivered back to the seller's service point
e_{violate} (e_v)	Sensor detected a violation of predefined threshold.
$e_{\text{b\_approve}}$ (e_{by})	Condition of the goods is approved by the buyer.
$e_{\text{s\_approve}}$ (e_{sy})	Return is approved by the seller (seller is satisfied).
$e_{\text{b\_reject}}$ (e_{bn})	Condition of the goods is disapproved by the buyer.
$e_{\text{s\_reject}}$ (e_{sn})	Return is disapproved by the seller, conflict request is generated.
$e_{\text{b\_nofeed}}$ (e_{bf})	Item feedback deadline has been expired (24 hours).
$e_{\text{s\_nofeed}}$ (e_{sf})	Return feedback deadline has been expired (24 hours).
$e_{\text{b\_abort}}$ (e_{bc})	Buyer changes his mind and aborts the transaction.
$e_{\text{s\_abort}}$ (e_{sc})	Seller changes his mind and aborts the transaction.
e_{clerk} (e_c)	Clerk decides how to proceed with the conflict.

Table 1: States and events of the agreement.

The relation between members of Q is established by the occurrence of the selection of events, defined in Σ, using the transfer function $\delta : Q \times \Sigma \rightarrow Q$. In context of the Secure Package system, δ takes following shape:

δ	e_p	e_a	e_d	e_{sp}	e_{bt}	e_{st}	e_v	e_{by}	e_{sy}	e_{bn}	e_{sn}	e_{bf}	e_{sf}	e_{bc}	e_{sc}	e_c
$\boxed{q_s}$	q_s	q_l	q_s	-	-	-	-	-	-	-	-	-	-	q_s	$\underline{q_a}$	-
q_l	-	-	-	q_t	-	-	-	-	-	-	-	-	-	$\underline{q_a}$	$\underline{q_a}$	-
q_t	-	-	-	-	q_d	-	q_{tv}	-	-	-	-	-	-	-	-	-
q_{tv}	-	-	-	-	q_{dv}	-	q_{tv}	-	-	-	-	-	-	-	-	-
q_d	-	-	-	-	-	-	-	$\underline{q_c}$	q_r	-	-	-	q_c	-	-	-
q_{dv}	-	-	-	-	-	-	-	$\underline{q_c}$	q_{rv}	-	-	-	q_c	-	-	-
q_r	-	-	-	-	-	q_b	q_{rv}	-	-	-	-	-	-	-	-	-
q_{rv}	-	-	-	-	-	q_{bv}	q_{rv}	-	-	-	-	-	-	-	-	-
q_b	-	-	-	-	-	-	-	-	$\underline{q_a}$	-	q_j	-	$\underline{q_a}$	-	-	-
q_{bv}	-	-	-	-	-	-	-	-	$\underline{q_a}$	-	q_{jv}	-	$\underline{q_a}$	-	-	-
q_j	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	$\underline{q_a}$
q_{jv}	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	$\underline{q_a}$
$\underline{q_a}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
$\underline{q_c}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 2: The table, which defines the transition function $\delta : Q \times \Sigma \rightarrow Q$.

The starting state is q_s. It is initiated when a seller creates the agreement with any initial terms. The accepting states are: q_c (when the transfer of goods is completed and the seller is satisfied) and q_a (when the transaction is aborted). The accepting states define the terminating states of the contact (when it becomes inactive and is not modifiable anymore).

7.1.2 Formal definition and visual representation

To sum up the contents of Section 7.1.1, a formal definition of NFA M is constructed for purpose of contract state representation of the Secure Package system:

- $Q = \{$CREATED, LOCKED, TRANSIT, TRANSIT$_V$, DELIV, DELIV$_V$, RETURN, RETURN$_V$, RETURNED, RETURNED$_V$, CLERK, CLERK$_V$, INACTIVE, COMPLETED$\}$ is a set of states (abbreviation of each state is described in table 1).
- $\Sigma = \{e_{\text{prop}}, e_{\text{accept}}, e_{\text{decline}}, e_{\text{s\_post}}, e_{\text{b\_deliver}}, e_{\text{s\_deliver}}, e_{\text{violate}}, e_{\text{b\_approve}}, e_{\text{s\_approve}}, e_{\text{b\_reject}}, e_{\text{s\_reject}}, e_{\text{b\_nofeed}}, e_{\text{s\_nofeed}}, e_{\text{b\_abort}}, e_{\text{s\_abort}}, e_{\text{clerk}}\}$ is the input alphabet, which consists of events, derived in 7.1.1.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, derived in table 2.

- CREATED $\in Q$ is the starting state.
- $F = \{$INACTIVE, COMPLETED$\} \subseteq Q$ is the set of accepting states.

The formal definition of M can be used to create a visual representation of it in form of a graph, as shown in Figure 5. Each state $q \in Q$ corresponds to one single node and each transition in δ corresponds to a single edge between given nodes, labeled with the corresponding set of events from input alphabet Σ.

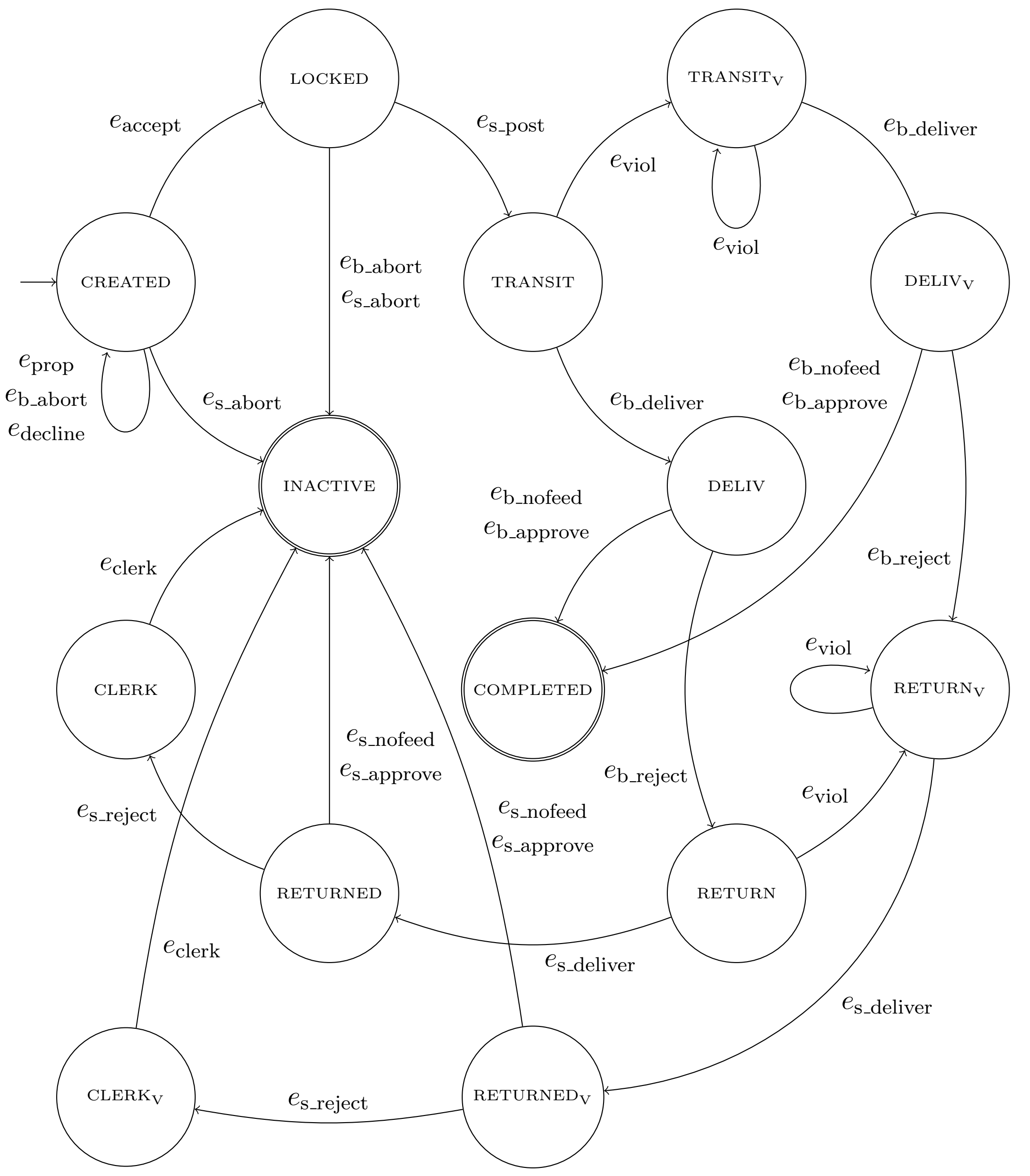


Figure 5: Visual representation of the deterministic finite automaton M.

7.2 System overview

With the state machine as a starting point, the contracts where developed through an iterative process where the states were represented by a state enum in combination with violation variables for the sensor data. Diagrams showing the structure of the contracts can be seen below with short explanations.

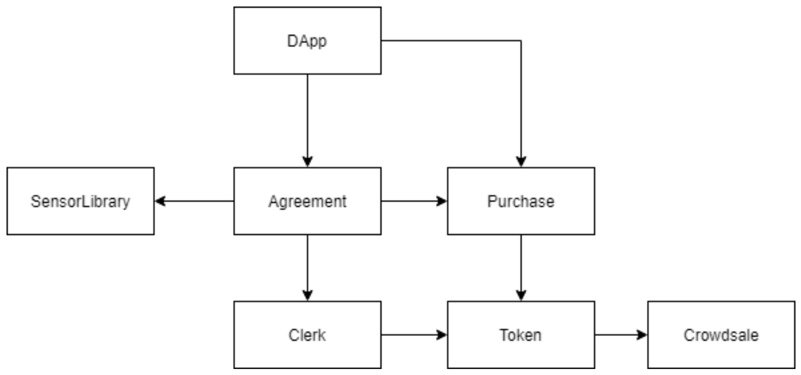


Figure 6: System diagram.

The system consisted of seven parts that can be seen in Figure 6. Each took care of a different task explained below.

DApp: The DApp contract created new agreements and kept track of the addresses of all purchases.

Agreement: A number of contracts were used to store and update the information about each agreement.

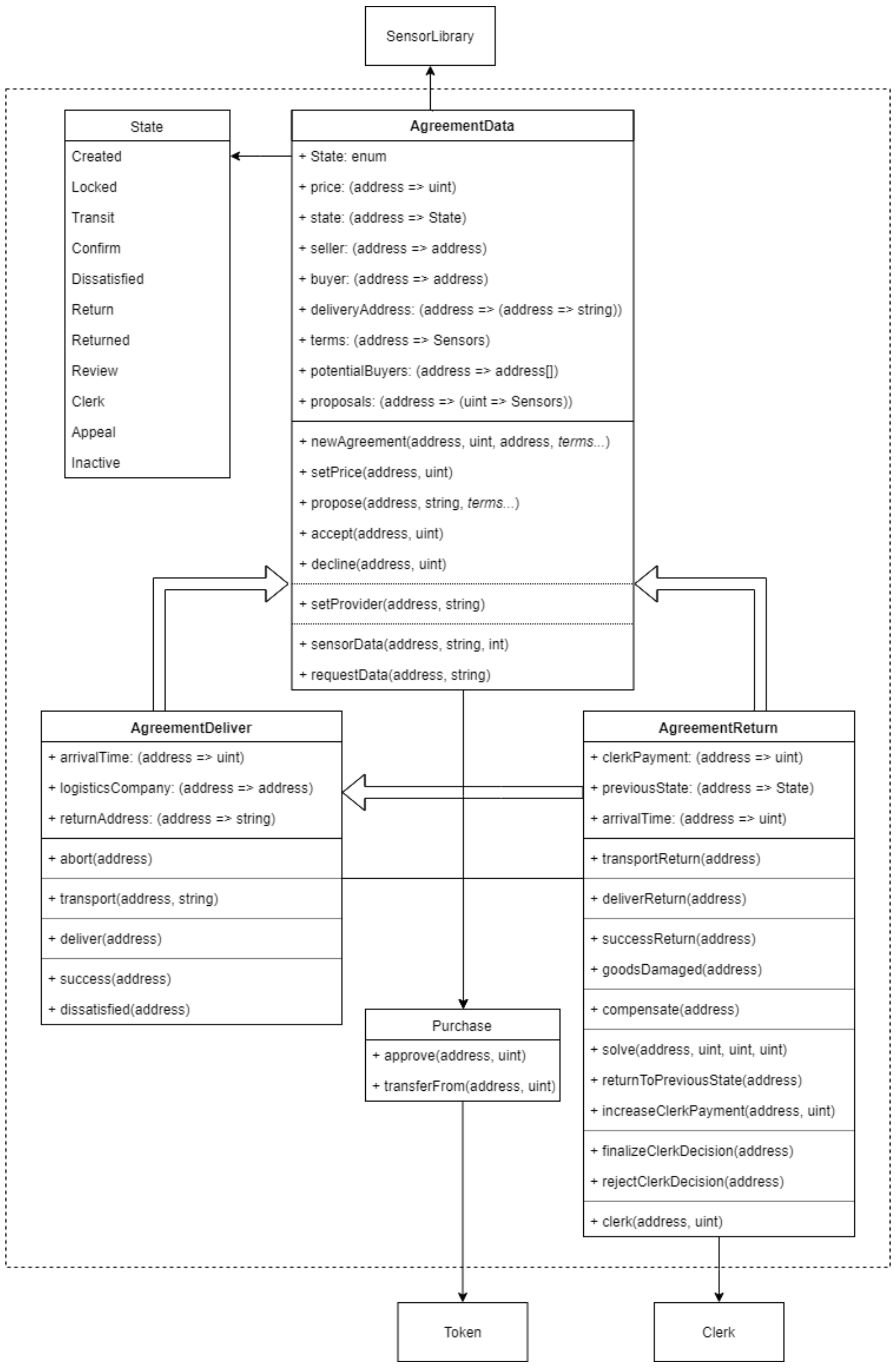
Purchase: Every agreement had a purchase contract which stored the tokens associated with it.

SensorLibrary: The sensor data corresponding to each agreement was handled by a library.

Clerk: A contract contained the list of clerks and the voting process.

Token: The payment method in the system.

Crowdsale: The way of acquiring tokens was through the crowdsale contract.



SensorLibrary
State
Created
Locked
Transit
Confirm
Dissatisfied
Return
Returned
Review
Clerk
Appeal
Inactive
AgreementData
+ State: enum
+ price: (address => uint)
+ state: (address => State)
+ seller: (address => address)
+ buyer: (address => address)
+ deliveryAddress: (address => (address => string))
+ terms: (address => Sensors)
+ potentialBuyers: (address => address[])
+ proposals: (address => (uint => Sensors))
+ newAgreement(address, uint, address, terms...)
+ setPrice(address, uint)
+ propose(address, string, terms...)
+ accept(address, uint)
+ decline(address, uint)
+ setProvider(address, string)
+ sensorData(address, string, int)
+ requestData(address, string)
AgreementDeliver
+ arrivalTime: (address => uint)
+ logisticsCompany: (address => address)
+ returnAddress: (address => string)
+ abort(address)
+ transport(address, string)
+ deliver(address)
+ success(address)
+ dissatisfied(address)
AgreementReturn
+ clerkPayment: (address => uint)
+ previousState: (address => State)
+ arrivalTime: (address => uint)
+ transportReturn(address)
+ deliverReturn(address)
+ successReturn(address)
+ goodsDamaged(address)
+ compensate(address)
+ solve(address, uint, uint, uint)
+ returnToPreviousState(address)
+ increaseClerkPayment(address, uint)
+ finalizeClerkDecision(address)
+ rejectClerkDecision(address)
+ clerk(address, uint)
Purchase
+ approve(address, uint)
+ transferFrom(address, uint)
Token
Clerk

Figure 7: Agreement and purchase contracts diagram.

As explained earlier in 7.14.2, the agreement was handled by three different contracts. Figure 7 shows the structure of these and the purchase contract.

The states in the data contract are closely related to the states in Figure 5, with the addition of the appeal, dissatisfied and review states. The state could be updated by any agreement contract. Other functions that set variables in the data contract had to be present in that contract. The thin dotted lines were used to represent groups of functionality available in different states. Data and deliver contract started with the 'created' state while the first function in the 'return' contract could be called in the 'return' state. A few functions such as 'sensorData' was available in multiple states (transit and the return in that case), which is not displayed in this figure.

The address parameter in the agreement functions is the Ethereum address of a purchase contract and used as a key to find the information for the corresponding agreement. The *terms...* in 'newAgreement' and 'propose' is a number of arguments that correspond to the terms either set initially or proposed by the buyer.

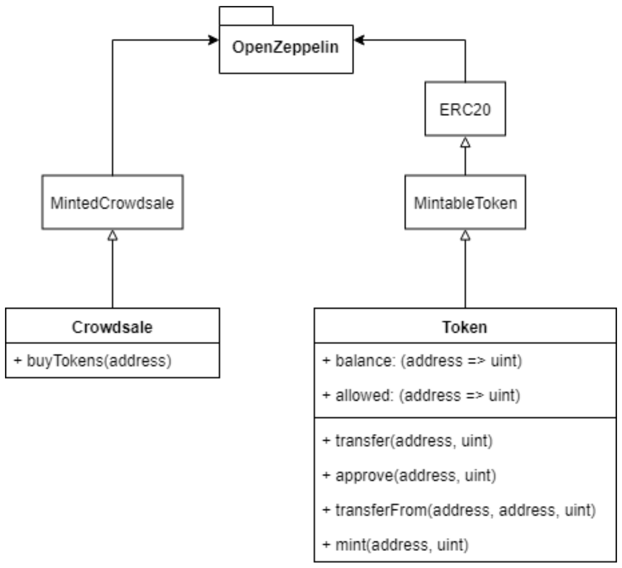
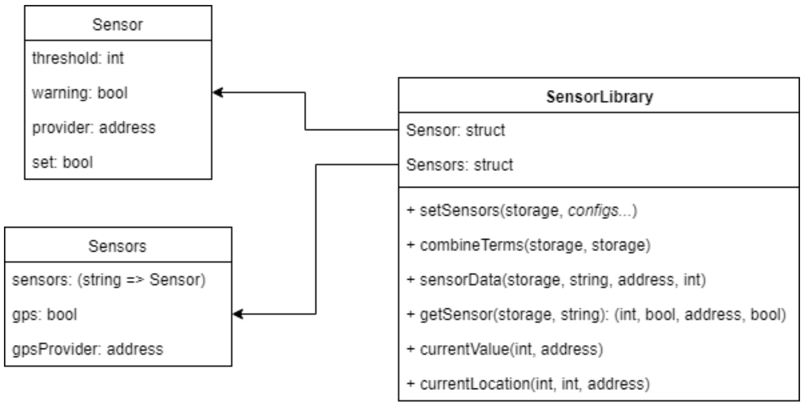


Figure 8: Token and crowdsale contracts diagram.

The more interesting functions of the token and crowdsale contracts in this project are displayed in figure 8. Both inherits classes from the OpenZeppelin [26] library.

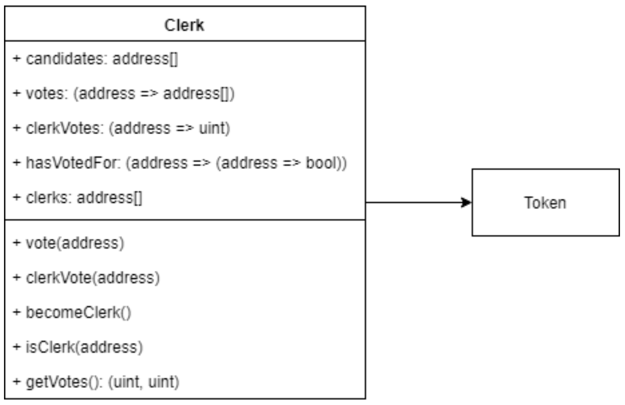


Sensor
threshold: int
warning: bool
provider: address
set: bool
SensorLibrary
Sensor: struct
Sensors: struct
+ setSensors(storage, configs...)
+ combineTerms(storage, storage)
+ sensorData(storage, string, address, int)
+ getSensor(storage, string): (int, bool, address, bool)
+ currentValue(int, address)
+ currentLocation(int, int, address)
Sensors
sensors: (string => Sensor)
gps: bool
gpsProvider: address

Figure 9: Sensor library diagram.

Each available sensor for an agreement was represented by an instance of the sensor struct and stored in a mapping. As can be seen in figure 9, this mapping was part of yet another struct which also contained information about the gps. *Configs...* is basically the same as *terms...* in figure 7.

'currentValue' and 'currentLocation' were not called by any contract. These were the functions called directly by the sensors when they received a 'Request' event from a user (the 'requestData' function used to send that event can be seen in figure 7).



Clerk
+ candidates: address[]
+ votes: (address => address[])
+ clerkVotes: (address => uint)
+ hasVotedFor: (address => (address => bool))
+ clerks: address[]
+ vote(address)
+ clerkVote(address)
+ becomeClerk()
+ isClerk(address)
+ getVotes(): (uint, uint)
Token

Figure 10: Clerk contract diagram.

Figure 10 shows the contract in charge of the clerk voting.

7.3 Addressing the problems

A short description on how the problems in Section 2 were solved can be seen in Table 3. These are explained in more detail later in this chapter and in Section 8 and 9.

Table 3: Problem solutions

Problem	**Solution**
Trust	Payment and agreement information were stored in a blockchain without third party involvement.
Attacks	Database related attacks are removed by using a blockchain. Research have been made to secure the contracts.
Reusable system	Since no third party was controlling the system, it could be used by any merchandise trading platform.
Sensor data	Sensor data are stored in smart contracts to provide immutability.
Trace payments	Feature of any Ethereum application.
Cost	Any payment to a third party was nonexistent. The added cost for gas was kept as low as possible.

7.4 Tools

This section shortly describes the tools and frameworks used in this project.

7.4.1 Truffle

Truffle [27] is a framework that makes it easier to develop smart contracts for Ethereum. It provides a number of features including compiling contracts and deploying them on a blockchain. On the truffle website, a number of example projects called 'boxes' can be downloaded to quickly setup the development. These usually consists of a basic token that can be transferred between accounts and a basic frontend with using one or more javascript frameworks.

Javascript commands can be written directly into the Truffle console to communicate with a deployed contract. This is done via the javascript API for Ethereum called Web3. A developer can therefore try the contract functions without a working frontend.

There also exists a testing framework within Truffle which helps with automated testing of the smart contracts. The tests can either be written in javascript to test interactions with a DApp, or as unit tests in solidity.

Because of gas cost, testing the functionality of smart contracts should not be done on a real blockchain. Truffle offers a personal blockchain service for development called Ganache, where a number of test accounts with fake ether can be used to send transactions. By default, the truffle boxes deploys the contracts on this blockchain but this can easily be modified.

7.4.2 MetaMask

The frontend of a DApp needs some way of communicating with the blockchain using the Web3 API. There are browsers such as Mist [28] that is specifically developed for this. An alternative is the MetaMask [29] extension for Google Chrome that works similarly but doesn't require downloading a different web browser. Users can connect to an Ethereum network and see their Ethereum wallets, this works with test networks as well. When sending transactions to a contract, the user can set gas limit and price through this plugin. Since it works as an Ethereum wallet as well, buying and sending ether is also possible.

7.4.3 Zeppelin Solidity

OpenZeppelin [26] is a solidity library for mainly token related smart contracts. It provides a couple of token standards, including ERC20. The developers of OpenZeppelin hope that their library can prevent some of the more common security issues that have been found in existing smart contracts.

Since smart contract systems involving tokens has to have some way of distributing them, OpenZeppelin also provides a number of ways to implement crowdsales. These can be combined and customized to support the needs of each individual project.

7.4.4 Solidity coverage

Unit testing is an important aspect of DApp development, especially since such applications often involve transferring money. It is highly recommended that smart contracts have 100% test coverage before releasing the application [14]. The test framework provided by Truffle does not show coverage, instead the Solidity Coverage [30] tool is needed.

7.4.5 Oyente

As explained earlier, mistakes made by smart contract developers can cause security issues. To help developers to not fall in the same traps as others have before, Oyente [31] analyses contracts and tries to find these common mistakes. The software looks for patterns in the contracts that have been proven to pose security risks.

In 2016, the Oyente developers gathered almost 20 000 contracts deployed on the Ethereum network. Out of these, 8 000 were flagged by the software which indicates that these security issues are very common [16]. The developers also checked the source code of a number of the flagged contracts to make sure that the issues indeed existed.

7.4.6 Angular

The frontend part of this project is written in javascript. There are a large amount of frameworks available that makes things easier and provide numerous functionalities when developing web applications. The user interface is not the main focus of this project and as little time as possible should therefore be spent on it. Because of that, Angular was chosen as the fronted framework since I have previous experience with it.

7.5 Payment

The first step to remove the third party from the agreement was achieved by implementing a payment method. This could potentially have been solved by simply using ether, but it would be rather limited in functionality. The ERC20 token standard [32] gives more flexibility, for example it allowed for the agreement contract to withdraw the payment from the buyer when the seller accepted a proposal as one atomic action. If the contract could not withdraw the payment from the buyer, the seller was not able to accept the proposal. There are different kinds of token standards for decentralized application that are used to represent something of value. The ERC20 token is one of the most commonly used for handling different forms of payment. Other standards might be geared towards tokens being used as rewards or medals for users in a system for example.

Each agreement created a new contract with an Ethereum address that could send and receive tokens. The Ethereum address was bound to a token account by the token contract. A new contract had to be created for each agreement since that is the only way to get a unique address without creating a new personal Ethereum wallet. Because the tokens were distributed among all agreements and users, this reduced the damage caused by a potential attack. Everyone could see the transfer and balance of tokens, providing additional sense of security for the involved parties since they could make sure where their money was.

Usually, the contract did not release any tokens before the agreement became inactive (the only exception can be seen in Section 7.10). That being said, the actors could retrieve their tokens allowed by the contract at any point in time.

These tokens are also an important aspect for the financing of many Ethereum applications. In this system, tokens could be bought for ether at a fixed rate. The sale was always available to improve the user friendliness of the system. Both the token and sale contract inherited contracts from the OpenZeppelin library [26]. There were no initial coins, instead they were minted as they were bought by some user. The sale of tokens had no time limit which meant that at any point during the contract's lifetime, a user could buy tokens with ether to trade with someone else.

This design was not good for the value of the tokens since they could be bought at any point at a fixed rate. Therefore there was no incentive for anyone to offer a higher price for the tokens, leading to a token that could only decrease in value compared to ether. Especially since after a purchase was finished, the actors who ended up with tokens might want to trade them for some other currency (supposedly some kind of fiat).

If the tokens instead could only be minted during a certain time period, there would be a limited supply which could lead to an increase in value as the system's community grew. But for the purpose of this project, this was not an issue and testing the system was simpler with a token that could be minted at any time.

7.6 Agreement information

To completely remove the involvement of a third party in the agreement, all necessary information also had to be stored in the blockchain. Because of the gas costs for transactions and data storage, some limitations of the data had to be made. This is a drawback of blockchain technology, especially public blockchains, since it can never compete with a database when it comes to the cost of storing large amounts of data.

The information about the agreement between seller and buyer that can be stored in the smart contracts is listed below. It is not an exhaustive list of everything that is stored in the system, instead it focuses on the initial agreement. For example, the Ethereum address of the logistics company is also necessary information, but it is not included until later when the transportation begins. The bold items are required in order to initiate the deal.

- **Seller** - The seller's Ethereum address.
- **Buyer** - The buyer's Ethereum address.
- **Price** - The price of the goods.
- Sensor configuration - What sensors to include and their thresholds.
- Description - A short description of the goods.
- **Delivery address** - Where to deliver the package.

7.6.1 Proposals

When a contract is created, the price and seller's address were set and the price could be updated as long as a proposal had not been accepted. The description and initial sensor configurations would also be written to the contract if the seller wanted to include them. Any sensor configurations could be updated later with a proposal from the buyer.

In the early stages of development the contract only allowed one potential buyer at a time and there were no restrictions on that user. Even if the seller declined the proposal, the user could then instantly send a new proposal, blocking any other user from buying the goods.

Even though the adversaries in this case are mostly wasting their own money, the seller has to pay for each decline. It was therefore something that needed to be solved.

If the seller has decided on a specific buyer, that user could be the only one allowed to make a proposal. While this solves the problem, it restricts the use cases of the system. Additionally, if the set buyer did not send a proposal, the seller would have to update the address of the buyer. If the buyer did not respond, deciding on when to update the address of the buyer would not only be a hassle for the seller, it would also waste gas and could be expensive.

Instead, the system was redesigned to keep track of multiple potential buyers for each agreement. When the seller accepted a proposal, the buyer for the contract was set.

If the buyer had proposed the use of any sensors, these would overwrite the original thresholds set by the seller. Sensors not included by the buyer would use the seller's initial configuration if that existed. The accept and decline functions were bound to each individual proposal. To deal with a change of price while there were pending proposals, all proposals were deleted when a new price was set.

A declined proposal was deleted from the array. Using the **delete** keyword in solidity left a gap in the array where the deleted value previously was. This means that all entries after the deleted element was still in the same position.

Many proposals would result in a very long array, but that barely affected the performance of this system. Most of the time, only one element was interacted with using its known position in the array. Obviously, the cost of that was the same no matter how long the array was. The only function dealing with the whole array was a constant function which returns its content. This was only used by the frontend and since it's a constant function and not a transaction, it did not consume gas.

Secondly, the seller did in fact not need to decline any proposals at all. When one proposal was accepted, the array had served its purpose and could potentially be deleted. This was not done in practice because while deleting they array cost gas, just leaving it as it was did not. Someone sending a large amount of proposals to the contract could therefore not force the seller to decline them. The worst thing that could happen was that the seller would have to go through a lot of proposals.

The fact that the array did not change in size when an element was deleted helped prevent some unwanted behavior. If a proposal was accepted and another was removed at the same time, the accepted element would still be in the same position, preventing race conditions. If the accepted proposal was removed, a quick check that it contained an invalid address could be made.

Another possibility for race conditions was if a buyer wanted to update a proposal. Since updates were not instant, the seller could accept an old proposal without knowing that it had been updated. Therefore, a buyer wanting to update a proposal had to withdraw the old proposal and send a new one.

Unfortunately, a seller accepting a proposal shortly after a buyer withdrew it could still be accepted. That depended on the order which the miner mining the block received the transactions. Because of the nature of the blockchain, one could never be sure which transaction would be put first.

While the system supported multiple proposals, the gas cost limited the usability of this function. Hopefully, a seller and buyer could come to an agreement before writing their decision to a smart contract to reduce the number of times a buyer had to send a proposal.

7.6.2 States

Once the buyer and seller had agreed on the terms, the most interesting information (unless perhaps if sensors were included) was the current state of the agreement. The states of the contract was closely related to the state machine in Section 7.1.

Table 4 shows the relationships between the state machine and the states in the

Table 4: Contract states

Contract state	**State machine equivalents**	**Short description**
Created	Created	The seller waits for a proposal to accept.
Locked	Locked	The package is brought to a logistics company that starts the transportation.
Transit	Transit, Transit_v	The package is being transported.
Delivered	Deliv, Deliv_v	The package is delivered.
Dissatisfied	Deliv, Deliv_v	The buyer is dissatisfied and brings the package back to the logistics company.
Return	Return, Return_v	The package is being transported back to the seller.
Returned	Returned, Returned_v	The package is returned.
Review	Returned, Returned_v	The seller wants to get compensation for damaged goods.
Clerk	Clerk, Clerk_v	A clerk is called to solve disputes.
Appeal	Clerk, Clerk_v	A user has appealed against a clerk's decision.
Inactive	Inactive, Completed	The agreement is completed.

contracts. In the state machine, a violation of a sensor threshold was represented by a new state, resulting in two transit states for example. The system kept track of this by using a variable for each sensor instead, thus the two transit states were merged to one. The same thing applied to the other states that had a violated version in the state machine.

The main purpose of the states in the system was to restrict what a user can do at different stages of the agreement. Because of this, a few states were split in two. These were the delivered, returned and clerk states.

Returning the package to the seller should only be possible if the buyer was not happy with the goods. Therefore this new 'dissatisfied' state was created to provide similar functions to the locked state. The review state was introduced to let the logistics company review a compensation request from a seller. Finally, when a decision has been made by a clerk (discussed in more detail in Section 7.10) to resolve a conflict, the agreement is moved to a state that allows a user to appeal against the decision.

Another thing that differs from the original state machine is that the clerk can be called at any point during the agreement. The state machine was not changed because it was used by another system as well.

7.7 Logistics

Of course, the transportation of the package has to be done by a logistics company. They are also responsible if the delivered goods have been damaged during the transit. Taking Postnord as an example, unless the weight of the package is below 2 kg, they provide a free insurance when the value of the goods is not more than 50 000 SEK [33]. There is also a service for sending a small envelope with an insurance for goods worth at most 10 000 SEK. There are also other services that provide insurance for more valuable deliveries for an additional cost.

While the payment for the delivery was not included in the contract, the project tried to remove as many trust issues as possible. Therefore, instead of providing the regular insurance, the logistics company could deposit tokens into the contract which they would get back if no accidents happened. A decision was made to require the one who started the transportation to send the same amount of tokens to the contract as the buyer.

This was a very optimistic solution if the system was to be used by many transports. The chance that a logistics company would agree to have a large portion of their assets locked in contracts is very small. Nevertheless, this was necessary to provide security for a seller that they could receive compensation for damaged goods. Without a way to provide the insurance in the contract, some parts of the system would leave loose trails without a clear solution. So whatever might be the best way to solve this in practice, this was the implementation chosen for this project.

7.8 Sensor data

The system did not only provide a way to store payment and agreement information, but also offered users to include sensor data. Since this project focused on the software part of the system, hardware issues and limitations were only briefly examined. These problems are discussed later in Section 10.2.

Sensor data could be used to prove that the package had been mishandled by the logistics company during the transportation. There were two different ways that sensor data could be sent to the blockchain, either by a violation of a threshold or by request from a user. This was allowed during the transportation of the package, i.e. during the transit and return states.

7.8.1 Threshold violation

As shown in Section 7.6, the seller and/or buyer could include sensor thresholds in the contract. Four sensors were supported: temperature, acceleration, pressure and humidity. Temperature could have a maximum as well as a minimum threshold while the others only had a maximum. This limitation was made to reduce the gas consumption.

A warning was set if the contract received data above the threshold set for that sensor (or below in the case of a minimum threshold). An event with the value sent would also be emitted. This was the only time the sensors were allowed to send data by itself, as the cost for gas would be ridiculously high if the sensors constantly wrote their data to the blockchain. If the threshold already had been violated once, the contracts did not accept a second data write. This was necessary to get a system that produced fairly good results while offering a reasonable price.

Each sensor in the package had an Ethereum account that was used to pay for the transactions made by it. This was also used as an id to notify the contract what sensor was tied to what threshold. The parties of the agreement interested in sensor data would have to make sure that enough ether existed in the accounts for it to make the desired data writes.

7.8.2 Requesting sensor data

The information about the current location of the package had to be available somehow. Using a GPS sensor, this information could be sent to the smart contract. The problem with location based data compared to the other sensors was that it could not have a similar threshold telling it when to send data as the rest of the sensors.

The sensor could instead have an interval between each data transmission. Let's say the sensor sent data once every hour and the delivery took one day. The cost of a transaction is at minimum 21000 in initial transaction gas and 5000 for a store operation [34]. There are also some other transaction fees such as 68 gas for each byte of the transaction, these will not be included in this calculation since they vary for each data write. Using these numbers, the gas usage for storing location based data every hour would be $(21000 + 5000) \cdot 24 = 624000$.

Due to the problems with setting an interval that both updates frequently enough and still offers a reasonable price, a different approach was chosen. When a user wanted to see the location of the package, a function in the smart contract could be called. This function emitted an event that the sensors listened to. This event included the address of the sensor which kept track of the desired data. When a sensor received an event with their address, it sent an event back to the contract log with its current value.

This let the users limit the cost of the sensor data while still getting information about the location of the package when they wanted to (with a delay caused by the blockchain depending on the gas price). An application could not only subscribe to events, all past events were available and could show what had happened previously.

Requesting data was not limited to the GPS sensor. The current data of any sensor could be retrieved in the same way even if the threshold had not been violated.

7.9 Agreement completion

If the contract is not aborted by the seller before it is locked, there are three states where the contract can be finalized. These are the delivered, returned and insurance states.

The first possible completion state is when the package has been delivered to the buyer. That person can then examine the goods for one day before deciding whether to return the package or not. To remove the risk of the buyer simply disappearing, after this 24 hour period, anyone could complete the agreement. This would most likely be the seller since they would like to get the payment for the goods.

Similarly to the delivered state, when the package has been returned to the seller, he or she could review the goods for 24 hours before the deal could be finalized by anyone. If the goods were damaged, the seller could ask the logistics company for compensation. The insurance would then be payed out to the seller if accepted by the logistics company.

If none of these states were able to complete the agreement, the clerk part of the service had to be called.

7.10 Clerks

Since it is likely to happen that some agreement would result in one or more parties being unhappy with the outcome, a way to solve this disagreement had to be included in the service. These can potentially be very complicated, and implementing a software that is able to decide on an outcome that the involved parties can accept is probably impossible. Besides, the logic required for that is not suitable for blockchain applications.

Therefore, if the system could not help the parties to come to a solution, some trusted entity was required in this case. These would presumably be a court or similar, but could in theory be anyone. As everyone else in the system, these were identified via their Ethereum address. Adding clerks could not be made by some owner or creator since that would make the system more centralized. Therefore, this was up to the users to decide.

7.10.1 Clerk Responsibilities

The clerks in the system were used to solve problems during and after the delivery of the merchandise. Anyone could call a clerk but it required a certain number of tokens to be payed to the person solving the issue. For more complex disagreements, a larger payment might be required. This payment could be increased if the contract waited for a clerk to take on the task.

Any clerk could try to solve the problem, but any of the involved parties could appeal against the decision before a day had past. If that happened, the contract returned to the clerk state, waiting for a better solution proposed by either the same clerk or someone else. This prevented any ridiculous decisions from being made. Since no one could get some tokens from the contract unless they accepted the clerk's decision, it is not in their best interest to decline indefinitely. Besides, declining costs gas, which meant that a party that was never satisfied would have to spend money to keep the tokens locked in the contract. If the clerk was a court, not accepting the outcome in this case would probably come with additional consequences.

There also existed another way for a clerk to resolve a conflict. If a user for example called for a clerk to solve an agreement where a package had disappeared and it was later found, the clerk could decide to return the contract to its previous state with all the tokens left in the contract.

It is important to remember that the clerks were a last resort to solve disagreements. For a large majority of trades in this system, the clerks would not be involved at all. They could also not in any way influence the outcome of an agreement if not requested by at least one of the involved parties. Even then, the decision could be appealed against.

When possibly some centralized authorities are forced to step in, the tokens and contract information were still decentralized and not controlled by anyone. No one could run away with the money without the permission from all involved parties. If that was possible, the clerk system could potentially be exploited.

7.10.2 Voting

To decide if an address was going to be added as a clerk, a voting system was implemented. Anyone could add themselves as a candidate and hope to get enough votes. A user would vote using their address and could of course only vote once. The system had no way of keeping track of its active users, meaning that determining when someone had enough votes to become clerk was a slight problem. Introducing some way of deciding the number of users could be made, this would lead to more design choices regarding inactive users. Especially since this kind of merchandise trading system would probably not be something most users access on a regular basis. Furthermore, an attacker could create a large amount of Ethereum addresses to manipulate the vote.

Some inspiration can be found in **proof-of-stake**, where the assets of the network is used to prevent someone from taking control of any important decisions. If a certain number of tokens is required to vote, it becomes much more expensive to create a lot of voting accounts. This did not prevent anyone from spreading their tokens among many different accounts to get more votes.

Since this was not a problem that could easily be solved without implementing a standalone voting system, a decision was made to weight the vote by the number of tokens an account has. This was basically the same as forcing an address to have some minimum balance (because of the possibility of sharing assets among different accounts owned by the same person). Since the total number of tokens in rotation can easily be counted, deciding when a candidate can become clerk is a lot simpler.

Of course, a lot of money could be spent to control the vote (depending of the size of the community, this would be more or less plausible). To reduce the number of times this became an issue, any aspiring clerk also required votes from half of the current clerks. Besides, the user paying to become a clerk could not use that position to steal any tokens.

The design of a voting system and deciding on the requirements of voters can be made very complex. A lot more time could be spent on improving this and further development is discussed in 10.3.

7.11 User Interface

The web interface for the application displayed a list of each active contract. Since the service was not a marketplace, no effort was made to make the web page act like one. Each box containing the contract could instead be seen as something included in an advert on any online trading site.

Depending on the user and current state, different inputs could be made to interact with them. Of course, the contracts themselves would only accept a certain number of functions even if all of them were always available on the frontend. The information about the agreement regarding price and sensors were always visible and updated when changes happened. In addition to the agreements, the token sale and clerk system was also present.

Because of an incompatibility between Web3 1.x and Metamask's Web3 provider, subscribing to events from smart contracts did not work when the fronted was developed [35]. At the time of writing, there was an open pull request on the Metamask Github page that might solve the problem, but there was no time to test that before the end of the project. Alternatively, the way the application is built could be redesigned to use a different version of Web3 where this issue did not exist. That solution was also scratched because of time constraints. The problems caused by this issue could mostly be solved by letting the frontend poll the data from the smart contracts.

7.12 Simulation

To be able to test how the system reacts to different situations, a rather simple simulation for the transport of packages was created in collaboration with Maxim Khamrakulov who used the same simulation for a database service. Since the focus of the project was the smart contract part of the system, it was decided to not depend on real deliveries and sensors, which could complicate and slow down the process of testing the functionality. As an additional benefit this made it possible to demo the whole system in a simple and clear way.

The simulation was developed as a web page with a map and graphs that showed the simulated data. The Google Maps API was used to find a route to the delivery point specified in the contract. This gave an array with steps that represented the turns that a car would have to make to reach the destination. To deal with the difference in length between the turns, the estimated travel time provided by the API for each step was used to wait at each step. After one second, the total travel time was updated by a specified amount, and if that was more than the total travel time required to reach the next turn, the map was updated. At each travel time update, a data point was added to each sensor graph. Each step along the route also had a longitude and latitude position associated with it, this represented the data from the GPS.

For the rest of the sensors (temperature, acceleration, pressure and humidity), a set starting value randomly increased or decreased by a small amount during each step in the delivery. To simulate accidents like dropped packages, each sensor also had a small chance to drastically change its value.

Since all deliveries were different, the simulation gathered information about the agreement to determine when to send data to the smart contracts. By providing the address of the smart contract, the simulation could pull data from it that was specified by the buyer and seller. This represented the sensors updating their configuration based on that information.

The configuration of the sensors (except the GPS) had a threshold, which made sure that the sensors only sent data to the smart contract when a violation of the agreement had occurred. This obviously required much less gas than sending data all the time. The contracts also forbid the sensors from sending data when a violation of the corresponding threshold had been set. Because of mining delays, the sensors might send data twice if the warning had not been set in the contract. The second transaction would fail and generate a REVERT error message. The good thing was that this would return the gas to the sender, otherwise it could result in a lot of wasted gas.

Unfortunately, because of the problems with event subscriptions [35], the requesting of sensor data by users was not implemented.

7.13 Security

7.13.1 Race conditions

The re-entrance issue can have devastating consequences as can be seen in 3.1. Two layers of security was implemented to deal with that problem. First of all, no calls to unknown contracts were made, which in theory would mean that there were no possibility for recursive function calls.

To support calls to unknown contracts in the future, the check-effects-interaction pattern [36] was used in all functions. This also helped prevent any overlooked security flaws in the contracts from causing serious damage.

A function following this pattern starts by checking that all requirements are met before executing any code. For this system, it meant that for example the state and caller had to be correct. Then the 'effects' of the function was carried out, these mostly involved setting and updating variables. At the end of the function any calls to other functions were made. This meant that even if they allowed for any recursive calls, all updates that was going to be made by the function had already happened. It would then be no different from making two separate calls to the function.

7.13.2 Withdraw pattern

To prevent any of the problems with denial of service described in Section 3.3, the withdrawal pattern [37] was used throughout the system. How many tokens an address was allowed to withdraw was stored in the contract, but it was not sent to the receiver unless that user actively requested the transfer to be made. Since this was a separate transaction, the worst thing that could happen was that the transfer failed and the user had to try again. Of course, the transaction had to be reverted if the transfer did not go through so that the amount the user was allowed to withdraw was not updated.

7.13.3 Approve tokens

As explained in 3.2.1, the approve function of the ERC20 standard can be exploited if the allowance was changed. OpenZeppelin [26] extends the standard to allow for an increase or decrease in allowance. By using these functions the problem with an adversary withdrawing tokens before and after the update of the allowance is prevented.

This was slightly more expensive than just updating the approval since it required a few additional computations and comparisons. For this system, these functions did not have to be used since the problem could be circumvented in another way.

There were a few times where the approve function was used in this system. First of all, a potential buyer had to approve tokens to the contract to allow the contract to transfer the payment for the goods. This transfer from the buyer was done when the seller accepted a proposal. Since this could be done only once per agreement, there was no possible way for anyone to transfer the tokens twice from a buyer that updated the allowance.

The insurance provided by the logistics company worked in a similar way. The approval of the tokens had to be made before the transportation of the goods started. Once it was initiated, it was impossible to transfer tokens from the company's account to the contract unless it was actively done by the company itself.

When the agreement was finalized, the tokens were allowed by the contract to be retrieved by the correct actors. This allowance could not be updated since the contract would be in an inactive state. Therefore it did not matter that the users could try to withdraw tokens multiple times.

What makes these scenarios immune to the problem with updating allowance is that the tokens are transferred via the contract. Therefore the rules of the code can be designed to prevent such problems from happening.

7.13.4 Initialization

If the contracts were not correctly initialized, they could be exploited similarly to the Parity multi-signature wallet hack, discussed in Section 3.4. In this project, the contracts held references to the contracts they communicated with. If these could be updated without any restrictions, an adversary could take control of the whole system.

Table 5: Early cost analysis of successful agreement

Action	**Gas**
Create	2423016
Approve tokens	67788
Propose terms	90985
Accept	74484
Transport	40645
Deliver	42687
Success	79785
Withdraw tokens	64461

Table 6: Additional sensor data contract cost

Action	**Gas**
Propose terms	264411
Set 3 sensors	$68008 \cdot 3$
3 sensor data events	$70234 \cdot 3$

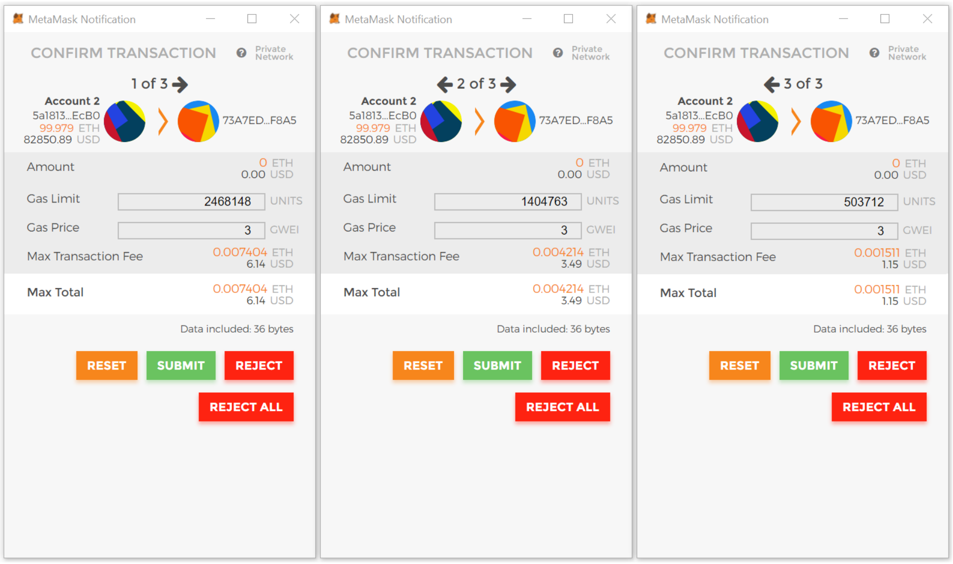
To prevent that, most references to other contracts are initialized in the constructor, without any way of updating it. The contract handling the agreement data needed to know the addresses of the contracts that were allowed to call it, otherwise anyone could update the price of an agreement for example. At the same time, the contracts calling the contract had to know the address of the contract to call. Therefore, a function was needed in these contracts to set the address of the data contract. By only allowing this to happen once, and requiring the data storage contract to call these function when it was created, the problem with anyone updating the addresses was avoided.

7.14 Optimization

7.14.1 Contract creation

When looking at the cost of transactions from the creation of the contract to the point when it is finalized, a scenario with sensors and one without were considered. Table 5 shows an example of the cost of a simple agreement without any sensors using an early prototype of the service.

When adding three sensors to the agreement that sent data one time each, not only the cost for that was added, but proposing terms also became more expensive. This can be seen in Table 6.



1 of 3

Account 2
5a1813...EcB0
99.979 ETH
82850.89 USD
73A7ED...F8A5

Amount	0 ETH	0.00 USD
Gas Limit	2468148	UNITS
Gas Price	3	GWEI
Max Transaction Fee	0.007404 ETH	6.14 USD
Max Total	0.007404 ETH	6.14 USD

Data included: 36 bytes

2 of 3

Account 2
5a1813...EcB0
99.979 ETH
82850.89 USD
73A7ED...F8A5

Amount	0 ETH	0.00 USD
Gas Limit	1404763	UNITS
Gas Price	3	GWEI
Max Transaction Fee	0.004214 ETH	3.49 USD
Max Total	0.004214 ETH	3.49 USD

Data included: 36 bytes

3 of 3

Account 2
5a1813...EcB0
99.979 ETH
82850.89 USD
73A7ED...F8A5

Amount	0 ETH	0.00 USD
Gas Limit	503712	UNITS
Gas Price	3	GWEI
Max Transaction Fee	0.001511 ETH	1.15 USD
Max Total	0.001511 ETH	1.15 USD

Data included: 36 bytes

Figure 11: Cost of creating different versions of the contract.

It is clear that the initial creation of the contract was the most expensive action by far. Two different ways of reducing this cost was evaluated.

By moving most of the code to libraries, the size of the contract became a lot smaller. All the functions were still present in the contract, but they called the corresponding functions in the library.

Another possible solution was to have one contract that kept track of and updated the information about all active purchases. This would be created only once and when initiating a new purchase, a minimal contract handling the payment was created and the terms were stored in the other contract.

Figure 11 shows the cost creating the different contracts. The leftmost one was the original where all code was in the contract which was created. The middle one used a library for the logic and cut the cost of creating the contract almost in half. Finally, creating a contract only for the payment removed most of the creation cost.

This does not necessarily mean that the last one is the best, for that the cost of the function calls need to be analyzed. The second and third alternative showed close to no difference in function call gas while both of them required slightly more than the original. Since the second alternative showed no advantages compared to the third one, the second one was dropped.

Table 7 shows the difference in gas usage when comparing the first (old version) and third alternative (new version). The scenario is the same as before where a successful agreement with three sensors is analyzed (the gas usage for the old version is a combination of Table 5 and 6).

As noted earlier, there was a huge difference in gas required for the contract creation. Most other function calls required around 2000 more gas when using the third alternative. This cost mainly came from the additional parameter and lookup that was required

Table 7: Gas usage comparison

	Gas old version	**Gas new version**	**Gas difference**
Create	2423016	503712	1919304
Approve tokens	67788	67788	0
Propose terms	264411	267405	-2994
Accept	74484	80437	-5953
Set 3 sensors	$68008 \cdot 3$	$69792 \cdot 3$	$-1784 \cdot 3$
Transport	40645	43446	-2206
3 sensor data events	$70234 \cdot 3$	$72948 \cdot 3$	$-2714 \cdot 3$
Deliver	42687	44893	-2206
Buyer satisfied	79785	85792	-6007
Withdraw tokens	64461	64461	0
			1885849

during each call. The parameter was used as a key to find the correct object in the data storage of the contract that kept track of all the purchases.

'Approve tokens' and 'withdraw tokens' had the same gas consumption in both implementations since they only called the token contract which was not changed. 'Accept' and 'buyer satisfied' also involved calls to the token contract. In the old version, the contract that updated information (changed state for example) was also in charge of the payment. It could then approve and transfer tokens directly in the 'accept' and 'buyer satisfied' functions.

In the new version two different functions in two different contracts needed to be called. One that updated the information and one that handled the tokens. A contract can for example not approve tokens of another contract because of obvious security risks.

Based on that information it was pretty clear why 'accept', which transferred tokens to the contract, and 'success', which approved tokens from the contract to the buyer, had a larger difference in gas usage than other functions. These functions of course also needed the additional key parameter.

In the simple scenario in Table 7 the new version of the contract consumed 33 455 more gas than than old version during the function calls. When comparing the total, the new version is still the winner with almost 1 900 000 less gas consumed. Unless something like 700 sensor data events were sent to the contracts, the new version was the better choice.

At the time of this test, the contracts were still in early development. Later when adding more functions, these would most likely follow the same pattern as the old ones; they would cost more to create than the they would save on their calls. The new version was therefore used during the rest of the project.

7.14.2 Splitting large contracts

Each block in Ethereum has a gas limit [34]. This is the total combined gas limit of all transactions in the block. Since it is the miners who decide what transaction should be included in a block, they are also in charge of the final gas limit of the block. The average gas limit of blocks has increased overtime and at the time of writing the main Ethereum network had a limit of around 8 000 000 [38].

The default block gas limit on the test network provided by the Truffle framework [27] that was used in this project was 6 721 975. With the size of the contract handling the agreement data increasing, it was clear that with all the functions planned it would surpass this limit. This would have been the case for the other proposed solutions to reduce contract creation cost in 7.14.1 as well. This meant that it would be impossible to deploy the contract on the network.

The reason for not creating multiple contracts of smaller size from the beginning was because of another gas related aspect. These smaller contracts would have to keep track of the same state, seller, buyer, etc. So to split the larger contract into two smaller ones, a third from where the data could be retrieved was needed as well.

As mentioned earlier, making extra function calls that are part of a transaction increases the required amount of gas. A small test was made to see the increased gas usage. A simple contract with a function similar to the basic functions in the agreement contract was created. The function updated a value if the contract was in the correct state. When the state was stored in the contract, the gas usage was 41 868. If the state had to be retrieved from another contract via a function call, it increased to 42 594.

Since all functions in the agreement contract at minimum had to check the current state, these function calls would result in a significant increase in gas usage. The alternative was to change the gas limit of the blocks in the test network and try to keep the contract size below the current gas limit of the Ethereum blocks. Clearly, this is not a stable solution since the block size constantly changes (even though it rarely fluctuates more than around 10 000 from its current average value).

There are also some other problems with setting the contract size right below the gas limit. Miners choose what transactions to include in a block based on how much money they can earn. If someone has sent a transaction with a very high gas price, miners are very likely to include that in the next block. A transaction (in this case a contract creation) with a gas limit equal to the block limit would not fit in the same block as the transaction with the high gas price. Small transactions with low gas price can fit in the same block as more expensive ones, making them more likely to be included in a block.

Therefore, a rewrite of the large contract had to be made, even though it would increase the cost of most function calls. The solution was three contracts which handled the information about the agreement and the structure of those can be seen in Figure 7. All functions that did not update any initial terms or proposals was moved to the second or third agreement contract. The state had to be updated by all three contracts, which meant that the contract in charge of that information needed to provide a setter for it. It also had to restrict the allowed callers for that function to be only the other two agreement contracts.

8 Result

This section describes how scenarios similar the user stories in Section 5 can be carried out in this system. The cost and security of the system is also examined and discussed.

8.1 Basic agreement

Most users that are trading online do not deal with large amounts of money. These people might still be interested in the transparency of the blockchain and the exclusion of the third party in the agreement. This was the basic use case of this system, where the contracts acted as a state machine and stored the payment for the goods. If this would be used instead of the secure package provided by Blocket.se [1], it would provide a similar service with improvements on the trust, security and transparency aspects discussed in Section 2.

The service could be offered as an alternative payment method and agreement storage on an online marketplace. A seller would then initialize the agreement by specifying the price. A potential buyer had to contact the seller and come to an agreement. This can, and is recommended, to *not* take place on the blockchain. The smart contracts' main purpose was to store information about the agreement and not be used as a chat for the involved parties which would consume large amounts of gas.

The tokens required to pay for the goods were transferred from the buyer to the contract when the seller accepted the proposal. The package would then be handed over to a logistics company that begun the transportation. As most logistics companies take responsibility for damaged goods during transportation of reasonably sized packages, they were required to provide the same amount of tokens to the contract as the buyer. When the package was delivered and if the buyer was satisfied, the seller was allowed to withdraw the payment from the contract and the logistics got back their insurance money.

8.1.1 Cost analysis

As this would be the most common use of the service, it was important that the cost of this was kept low. Exactly how much this would cost depended on the gas price chosen and the current price of ether. In the following cost analysis, the price of ether was set to \$700 and the gas price of choice was 3 GWei, or $3 \cdot 10^{-9}$ ether. This was often enough for a transaction to be included in the next couple of blocks, meaning it would be mined within a few minutes. For larger transactions in terms of gas usage however, the wait might be longer unless the gas price was increased. For example, the agreement creation could potentially suffer from this.

Table 8: Cost analysis basic agreement

Action	**Gas usage**	**Price in USD**	**Actor**
Create	651166	1.36745	Seller
Approve tokens (payment)	67821	0.14242	Buyer
Propose	154482	0.32441	Buyer
Accept	146323	0.30728	Seller
Approve tokens (insurance)	67821	0.14242	Logistics
Transport	134304	0.28204	Logistics
Deliver	82833	0.17395	Logistics
Success	146914	0.30852	Anyone (after a certain time)
Withdraw (seller)	64461	0.13537	Seller
Withdraw (logistics company)	64461	0.13537	Logistics
		3.31923	

Using these prices for gas and ether, the cost for a basic, successful agreement can be seen in Table 8. The gas usage was an estimated value provided by the Ethereum API. In a scenario similar to what was described earlier (or in Section 5.1), the total cost for all actors would be $3.3. It would be reasonable to compare this cost with the cost for cash on delivery service offered by logistic companies since it is similar in the way the payment to the seller is handled.

Parcel transfer in Sweden currently starts at around $11 for smaller packages when using Postnord or DBSchenker. Cash on delivery adds on an extra $14 when using the Postnord service. It is not possible to add cash on delivery to any delivery by DBSchenker, but they do have a deal with Blocket.se when using their secure package [1] service where cash on delivery is included. The price for the cheapest version of this is $12.8, which would suggest that the cash on delivery only increases the price by slightly more than $1.

It is clear that the deal between Blocket and DBSchenker results in quite a large decrease in price for a customer. Still, using the blockchain service would not be that much more expensive while adding transparency and improving trust. It would also be a lot cheaper than adding cash on delivery to a Postnord parcel transfer. There is also a maximum on the value (10 000 SEK) of the goods transferred and the package size when using the Blocket secure package. For packages that are too large or too expensive, the customer has to find an alternative where cash on delivery might not be available or a lot more expensive. In those cases, the blockchain service does not only remove the third party, but provides a very reasonable price as well.

8.2 Agreement with sensors

The service also gave the users the option to include sensors in the agreement. Thresholds for temperature, pressure, humidity and/or acceleration thresholds that the seller and buyer had agreed upon would be stored in the contract. A logistics company transporting the package would have to agree to not violate these thresholds. GPS could also be included to send events to the blockchain with the location of the package. This dealt with the sensor data problem discussed in Section 2.5.

These sensor thresholds were originally set by the seller and could be updated by the buyer when sending a proposal. If the seller accepted the proposal, the buyer's thresholds would overwrite the seller's, while the thresholds not set by the buyer used the ones specified by the seller. This means that the buyer could not remove any sensors that the seller had decided to include, only include additional ones and update thresholds.

An additional cost for the hardware had to be payed by the seller. Either by providing the sensors themselves or by paying the logistics company to include them in the package. If the seller wanted the buyer to contribute to this cost, they had to update the price of the goods accordingly. This was not something that was included in this project (except for the option of updating the price of course), since it focused on the software. But it is a rather important aspect of a fully working system.

Each sensor had to be connected to the blockchain to send any data that violated a threshold. This was then stored as a warning in the contract and could be seen by anyone. An event also fired with the value that caused the warning. Since the sensors were connected to the blockchain, they could also listen to events that were sent by the users of the system. This could be used to retrieve the location of the package or any other information from the included sensors.

Since this introduces quite a lot of additional costs, it should be seen as an optional inclusion when something valuable is being transported. If the price of the goods are large compared to the cost of these sensors, it might be worth adding this extra form of safety for the seller and buyer.

8.2.1 Cost analysis

Depending on how many sensors were included, the cost for the service was different. The cost for including one sensor in the service can be seen in Table 9.

Table 9: Cost analysis one sensor

Action	**Gas usage**	**Gas increase compared to no sensors**	**Price in USD**	**Actor**
Create	731457	80291	1,53606	Seller
Approve tokens (payment)	67821	0	0,14242	Buyer
Propose	212272	57790	0,445771	Buyer
Accept	179539	33216	0,377032	Seller
Approve tokens (insurance)	67821	0	0,1424241	Logistics
Transport	134304	0	0,2820384	Logistics
Set sensor	29235	29235	0,061394	Sensor
Sensor data	32249	32249	0,067723	Sensor
Deliver	82833	0	0,1739493	Logistics
Success	146914	0	0,3085194	Anyone (after a certain time)
Withdraw (seller)	64461	0	0,1353681	Seller
Withdraw (logistics company)	64461	0	0,1353681	Logistics
			3,808071	

The table suggests that a sensor only added $0.5 to the service, but in a real scenario the hardware would increase the total price by a lot more. DBSchenker offers special services with sensors for their transfers, but does not display any pricing [39].

As explained earlier in Section 7.8.2, the system is limited in the amount of data that can reasonably be stored. However, it is still possible for someone to provide a service with a database for storing the history of the sensor data. This way, the blockchain system would provide an extra level of security on top of the traditional service. Depending on who sets up this database, this would probably still come at some extra cost, even though it would be nothing compared to storing it on the blockchain.

8.3 Disagreements

A dissatisfied buyer had 24 hours to send the package back before anyone could withdraw the tokens from the contract. The return process worked similarly to the first transportation, with the original agreement of potential sensors still valid. A returned package started a new 24 hour period where seller could examine the goods. It could then be accepted which returned the tokens to the buyer and logistics company. If the seller found that the goods were damaged, they could ask for a compensation from the logistics company. Since it is not reasonable to provide evidence such as images on the blockchain, this had to be done by some other method.

As long as the involved parties followed the original agreement, the system could handle the payment and finalize the contract by itself. However, if some party was not happy with the outcome, the system had to get help from some authority solve the dispute. This could for example be if the logistics refused to compensate the seller for damaged goods. At any point, a clerk could be called by an actor in the agreement which automatically locked the contract. The clerks were voted for by the users and once they had enough votes, these trusted people or organizations could try and solve any disagreements.

What disagreements to solve was up to each individual clerk and therefore a reward had to be given to the one who came up with a solution. The reward could be increased by anyone who wanted the problem to be solved. A user that did not agree with the decision made by a clerk could appeal against it and wait for another solution. After yet another 24 hour wait the contract viewed the solution as final. The actors could then either retrieve their allowed tokens or the agreement would continue, depending on if the clerk decided to return it to the previous state or not.

8.3.1 Cost analysis

The clerk inclusion could be seen as a last resort for complicated cases and should not be seen as a common use of the system. This of course also introduced more cost for the user choosing this option to solve a dispute.

Table 10: Cost analysis return

Action	**Gas usage**	**Price in USD**	**Actor**
Dissatisfied	56650	0,118965	Buyer
Return	55233	0,115989	Logistics
Deliver	85123	0,178758	Logistics
Goods damaged	55698	0,116966	Seller
Compensate	158166	0,332149	Logistics
		0,862827	

Returning the package meant that the agreement had to go through a few more states, resulting in higher total gas usage. Table 10 can replace the 'success' action in Table 8 or Table 9 for a full agreement, which would increase the cost of those by around $0.5.

8.4 Security

A lot of time was spent investigating possible security issues in the contracts. Solidity-coverage (discussed in Section 7.4.4) was used to ensure that 100% of the code was tested and worked as expected. This of course did not mean that there could be no bugs or possible points of attack for a hacker but it reduced the risk.

The Oyente tool (Section 7.4.5) provided an analysis on some common and critical bugs such as transaction-ordering dependency, reentrancy vulnerability and the second Parity

Table 11: Cost analysis clerk

Action	**Gas usage**	**Price in USD**	**Actor**
Approve tokens (clerk reward)	67821	0,142424	Anyone
Call clerk	137145	0,288005	Anyone
Solve	144937 - 190129	0,304368 - 0,399271	Clerk
Accept	180369	0,3787749	Anyone (after a certain time)
		1,113571 - 1,208474	

multisig bug. These were explained earlier in chapter 3. None of these vulnerabilities were found by Oyente in the contracts. Oyente also did not found any timestamp dependency or callstack depth attack vulnerability. An explanation of these can be found in the Oyente paper [16] for example.

What it did find was possible integer over- and underflows. These could for example occur if someone set a ridiculously high price or an enormous amount of proposals was sent to an agreement. That being said, neither of those were a serious problem for the contracts. A price so high that it caused an overflow would not be included in any serious agreement. It could also not be used by an adversary to attack the system. An overflow in the number of proposals would be a problem, but that was so expensive that it could never happen. The other integer over- and underflows were of the same nature, therefore no effort was made to fix them. Besides, this would increase the gas cost for the affected functions, which seemed like a waste when no real issues was discovered.

Unfortunately, Oyente had a bit of a problem with covering all the functions. For the return contract, the coverage was as low as 35%. This can be explained by the fact that the functions in that contract was only available in the later stages of the agreement. Oyente probably had problems calling the functions in the other contracts in the correct order to be able to run all the code in the return contract.

Coverage issues aside, the analysis showed no obvious flaws in the contracts. This reduced the risk of a successful attack on the system even further.

9 Discussion

Overall, the main goal of increasing the trust of a trade agreement has been reached. However, a blockchain solution and the technology in general is not without flaws. This section describes some of these and discusses the different results of the project.

9.1 Environmental concerns

The main criticism of large blockchains such as Ethereum and Bitcoin is the energy usage. The consensus algorithm that the miners have to run results in constant consumption of electricity. Currently, the estimated annual consumption of electricity is around 18 TWh [40]. This leads to an average energy cost for transactions of around 72 KWh, which is equal to the electricity consumed by a couple of households during one day.

These numbers are obviously huge and a common argument against any blockchain based system. As the usage of blockchain technology grows and attracts more miners, the energy consumption will continue to increase. The change of consensus algorithm to a **proof-of-stake** [4] solution proposed for a future update of Ethereum can greatly reduce this waste of electricity. This is already used by some cryptocurrencies such as Nxt [41].

Switching to proof-of-stake is not trivial, and Ethereum plans to slowly transition to their implementation of the algorithm called Casper [4]. How much of an improvement on the energy usage the change will bring is not certain. But since a system without proof-of-work does not have any miners running expensive algorithms, it is safe to assume that the consumption will be a lot smaller. The miners are replaced by so called validators, who instead of buying hardware that run computations to reach consensus uses their money directly. Money can be put at stake, locking it in the system, and validators place votes on blocks they think should be included. How much a validator earns on average for a successful block addition that they voted for depends on how much money they have invested. These are only payed by transaction fees, meaning that the creation of coins that was given as payment for miners does not exist in a proof-of-stake system.

Whether or not proof-of-stake can solve the environmental problems for large blockchains such as Ethereum remains to be seen, and it does not come without issues of its own. Until this is added to the Ethereum network, the developers of DApps can only try to make their gas consumption of their contracts small, as this is closely related to how much energy the miners spend on the transactions.

9.2 Choice of network

The development of this project was made on a test rpc that mimics the behavior of the main Ethereum network. This does not mean that it has to be deployed on the main network. A new public blockchain can for example be created that is tailored to better support this system.

How much better performance a customized blockchain can provide is uncertain, but changes could potentially be made to allow for a cheaper storage of sensor data. As such, one of the largest drawbacks of this system could be improved. Some form of payment would of course still have to be made to the nodes running the code of these data writes. Implementing a good optimization of this is a rather complex task, if possible at all.

A simpler option would be to deploy it on a new blockchain following the same rules as the main network. This would result in less traffic, potentially reducing the delay of transactions.

Both these ideas have the same big issue. The network needs to attract miners to prove the correctness of the contracts. A small number of nodes is enough to run the system, but it would not be well protected against attacks.

On large blockchains such as bitcoin and the main Ethereum network, a so called *51% attack* is practically impossible due to the number of nodes in the network. This attack involves an adversary controlling more than 50% of the blockchain and is then able to manipulate the ledger. In smaller networks however, such an attack is not unheard of. Recently, the blockchain of the cryptocurrency Verge suffered from this which resulted in a hard fork to mitigate the damage [42].

If the cost and time of transactions on the main network can be tolerated, it is probably the best deployment option. As is the case with all systems transferring money, security is the most important aspect.

9.3 Competitiveness

As described in section 2, this system tried to improve a centralized system by providing a decentralized alternative. To be competitive, the system should not only show the strengths of a blockchain solution, but also try to reduce the effects caused by its weaknesses.

One of the main selling points of the system is the improved trust for the involved parties. If this is not something that is interesting to a potential user, it would be hard to convince that person to use this service. As such, a seller could for example miss out on a buyer that doesn't want to use the blockchain to transfer the money. But since this is an alternative secure payment method that can be used by existing, traditional services and not a standalone marketplace, a seller can reach out to more people.

Large data writes such as images are not suitable for the blockchain. This is another weakness and argument against using it to create an entire marketplace. To reduce the number of data writes, users are encouraged to come to an agreement before sending it to the smart contract. Even though the system supports multiple proposals, an aspiring buyer should not send 10 offers to the contract before contacting the seller first. Deciding what has to take place on the blockchain is not only up to the developers of a decentralized application but the users as well.

Trying to coexist with traditional online marketplaces rather than compete against them can also attract more investors. Even if someone was interested in the service, they might be concerned that other people would not be, resulting in them not wanting to support the project. An established, online marketplace allowing this system as an alternative payment method (which would not cost them anything) would greatly increase the exposure and interest in the project. This could also provide a good opportunity for providing a database solution of the sensor data history, as briefly discussed in Section 8.2.

9.3.1 Attacks

If the system is deployed on the main network, the 51% attack should not be an issue. Even though the smart contracts have not been proven safe (which hypothetically can be done by providing a mathematical proof for all of them), they follow the recommendations for secure contracts and have been tested thoroughly. The main remaining point of attack possible is the attack on personal wallets. Let's compare that to an attack on a database in a centralized system.

In these two cases, getting access to either an account or a database is certainly possible. The difference is the damage it can cause to the systems. If the database of the centralized system is hacked, all users are affected. On the other hand, in the case of a hacked Ethereum wallet, an attacker can only steal the money stored on that address. It can potentially also affect the active trades involving that user. But since any attempt to withdraw money from the contract that does not follow the original agreement has to go through a clerk, it can only cause limited damage.

If an account contained a large amount of money, or many private keys for various accounts were stored in the same place (a database for example), a hacker that got access to these could still steal a lot of money. The system has no control over the amount of money stored in the involved accounts, that is up the users to decide for themselves. What the system is controlling however is the amount of money handled by its contracts. For each new agreement, a new contract (a new address, comparable to a new account) is created which only holds the money for that purchase. Of course, this could still be a lot of money, but it limits the balance controlled by a single address.

A large scale attack on this system is therefore both harder due to its decentralized nature and has a smaller payoff. The main concern is still the security of the smart contracts. Unfortunately, it is easier for an adversary to find one exploit than it is for a developer to prove that there exist none. In the case of an attack, the clerk system can be used to lock the contracts. This might not be a perfect solution, but at least it helps if someone found a way to extract money from the contracts.

9.3.2 Sensor data

While the system can make sure that the provided data is not tampered with, it is limited in the amount of data that can reasonably be stored. Here it suffers compared to a centralized alternative, and it is up to each user to decide what they think is more important: large amounts of data or more trustworthy data.

The ability to request data does improve this issue quite a bit, and as long as a user is not interested in the history of each sensor, it accomplishes the task equally well (albeit a bit slower). It also allows for a gps sensor, which would be rather tricky to include otherwise. A user wanting more data could get it by simply paying for more transactions. The data writes by the sensors are payed for by their respective accounts. Anyone could provide the required ether to these transactions, the involved parties would have to agree on this beforehand.

9.3.3 Cost and time

The price for this service is of course closely connected to the size of the agreement and the number of transactions made to the contracts. In the case of an agreement where nothing goes wrong, the cost is small for everyone involved. The cost analysis in Section 8 showed that it was only outdone by one or two dollars when the logistics company had a deal with the marketplace to cover extra expenses. Without that, the blockchain service was cheaper.

Overall, the cost was probably not such a big difference that it would be the deciding factor when deciding on what service to use. The trust issues contra large data writes is probably more important for example.

The time required for data writes was another drawback compared to a database system. Depending on how the system is used, these delays might be a bigger or smaller problem. I a user wants to have a real time surveillance of the parcel transfer, it is clearly an issue. However, a seller of buyer might be satisfied with checking if a violation has occurred when the package arrives. In this case, how fast the transaction is does not really matter.

If a standard gas price is used, a transaction currently has to wait on average two blocks, or 30 seconds, before it is mined [43]. Note that a block requires a number of blocks to be added after it to finalize all transactions in the block, but when a block is mined the data from the transactions can already be read.

Whether the standard gas price results in a reasonable price for the transaction depends on the price of ether and the current traffic on the blockchain. Larger transactions are also slower since it is harder for them to fit in a block, this mainly affects the initial creation of larger contracts in this system.

9.4 New technology

The implementation of the system proved a bit more difficult than initially thought. Not only did the gas and security requirements complicate the development, but the tools and frameworks are quite recently released. Therefore they are far from perfect and compatibility between the softwares is a bit shaky, especially when updating to new versions.

Solidity [36] for example, which is the most commonly used language for writing smart contracts, is currently in version 0.4.23. As such, it lacks in some functionality and while some limitations are because of how Ethereum works, other might be because they have not been implemented yet. For example, doubles and floats do not exist and strings can not be passed between contracts. These are not serious flaws, but they limit what can be done in the smart contracts.

Information about the development of decentralized applications are also a bit sparse. There exists a number of tutorials on how to get started and write basic contracts, but for more complicated issues, thorough reading of the documentation and trial and error is usually the way to go. This is of course because even though blockchain has attracted a lot of interest in recent years, there are not a huge amount of developers working on applications for it.

9.5 Final thoughts

The project produced varied results on the initial goals. It does offer a secure merchandise trading service without a third party, but how competitive it is is debatable. The most important issues regarding trust can be considered a success, as the secure transfer that traditional services such as Blocket.se provides can be migrated to a blockchain solution.

Attacks of the sort where the database is hacked were also nonexistent, since the assets of the system was spread out among all its users. It did however introduce uncertainty about how secure the contracts are. Even if all recommendations were followed, it was close to impossible to be sure that there existed no flaws in the system that could be exploited.

The reusability was also good since the same safe storage of payment and agreement information could be used for different online marketplaces. For successful and/or small agreements, the service was pretty cheap, making it a good alternative for a user. However, as the contracts became more complicated and disagreements occurred between the involved parties, the cost could increase quite a bit.

Delay caused by the mining process were unavoidable but contributed to the negative aspects of the system. The improvements on the problems discussed in Section 2 might be enough for some to overlook the drawbacks, while some might prefer a traditional system with a third party.

Blockchain is an interesting technology with a large potential, but it has suffered from a number of setbacks and will probably encounter more problems in the future. The fact that many people uses trading of cryptocurrencies as a stock market has increased the exposure and interest in the technology and also increased funding for a lot of blockchain projects. But it has also caused issues with the technology to have dire consequences. First of all, the instable currencies is a problem which makes people feel uncertain about using the technology. Secondly, the mistakes made by developers are disastrous and gains a lot of publicity when a lot of money is involved. The DAO hack [10] and attack on Parity [21] [22] would not have been such huge failures if not for the amount of stolen money. In the future, this volatility will hopefully decrease. Blockchain systems would benefit a lot if the price for cryptocurrencies represented their usability and not how much money speculators hope to earn.

10 Future work

Since the development of a new system was a large part of this thesis, there exist a lot of ways to improve and continue the work on it.

10.1 Additional states

More possible states could be added to the agreement to increase the functionality or security of the system. The number of states in the contracts was barely touched since the early stages of development. The main reason for that was because of the thesis work done in parallel by Maxim Khamrakulov. The comparison and analysis done in that thesis would be greatly affected by rewrites of the design. It could also force similar updates to be done in the system developed in that thesis. Therefore the basic design was kept as consistent as possible throughout the project.

One update to the states that could be done is to add states that allows for handshakes with the logistics company. The seller could for example write the address of the logistics company to the contract to only allow that user to move the agreement through the transit states. An additional confirmation on the choice of logistics company from the buyer could also be added. When the package was delivered another handshake between logistics company and buyer could take place.

10.2 Sensors

While the contracts handled the data and events well, there exists some issues with the practical inclusion of sensors. Having a sensor that is connected to a blockchain during the transport is quite demanding on the network infrastructure. While the rest of the system tried to improve trust and security, these might be heavily reduced if the communication with and from the sensors is unreliable. Additionally, a defect sensor, either because someone tampered with it or because it is faulty, would send the wrong data to the contract without anyone noticing. Such problems can only be discovered by examining the sensor.

A lot of work would have to be put into the hardware part of the system in the future since it has not been examined by this project. The conclusion of that might be that some parts of the service are less useful than initially thought.

The sensors also had to be programmed to be able to communicate with the blockchain. This would also require an Ethereum wallet for the sensor and the ability to sign its transactions. To get a working solution would probably not be that complicated, but to reach high performance and availability might prove tricky.

Who adds the sensors to the package was not set in stone. The main reason someone would want to include sensors in a package was because they did not trust the logistics company. Having that company include the sensors would then introduce the same trust issues. The most logic choice from the service's perspective would therefore be to let the seller provide the sensors. This might not be accepted by the logistics company however, who would probably want to examine the sensors. If the sensors instead were provided by them, they could also easily be reused in future transports.

10.3 Clerk voting system

As explained in Section 7.10, a token-based voting system was created to decide if a user can become a clerk. The addition of a required number of votes from existing clerks before a candidate has enough votes made the system a bit more robust, but it was by no means perfect.

The biggest problems to solve is how to decide who is allowed to vote, and how many votes is required before a candidate can become a clerk. As the system did not keep track of its users, some way of registering voters is probably necessary to improve the process further. This introduces problems with how to prevent people from having multiple Ethereum addresses and possibly controlling multiple voting accounts.

10.4 Price volatility

Any tokens deployed on Ethereum are exposed to the price volatility of cryptocurrencies. In this system, the tokens could easily spend a couple of days in the contract. That means that compared to fiat currencies, the tokens received by the seller in the end might be worth less than the initial price set on the goods.

For this system, most users would probably want to exchange their tokens for fiat money once the agreement is complete. While some sellers might end up with more money and be happy, dissatisfied customers due to a drop in value is a problem.

A solution similar to the one used in Canya [24] could work quite well. They use an investment technique called **hedging**.

Hedging can be seen as a kind of insurance, i.e. someone is willing to pay for your potential loss. If the loss does not happen, the one providing the insurance is the one making money.

In more detail, hedging in this system could work as follows:

1. The seller sets a price of some goods to 100 SEK.
2. The buyer and seller come to an agreement and the buyer sends 100 SEK worth of tokens (let's say 100 SEK is worth 100 tokens) to the contract.
3. The tokens are sent to investors, who promise to give tokens back when the agreement is finished equal to the initial value.
4. If the value of the tokens have dropped 50%, the seller will receive 200 tokens from the investors when the buyer accepts the goods. On the other hand, if the value increases by 100%, the seller will only receive 50 tokens. In both cases, the seller has received tokens equal to 100 SEK, but the investors have either earned or lost money.

So where does these investors come from? In most DApps that involve some kind of token payment, an **initial coin offering** is made. These are meant to fund the project by selling the tokens (coins) to people that believe in the idea.

These could then serve as the investors, hopefully increasing their number of tokens. This is an even higher risk than just buying tokens hoping that their value will increase. If the value of the tokens drop, the investors will not only see the price of their assets go down, they will also have to give some of their tokens to the users of the system as compensation.

10.5 Privacy

Since the data on the Ethereum blockchain is public, it causes some privacy concerns. By encrypting sensitive data, it can be hidden for certain actors in the system. Running encryption algorithms on the blockchain can be costly. Therefore, it is better to do it on the frontend and send the encrypted data to the network. Besides, if some data was sent to the contract, it would be visible to anyone even if it was encrypted afterwards.

10.6 Load balancer

The current system with one contract that handles all data saves a huge amount of gas during the creation of each agreement. A problem can occur if there are a lot of active agreements at the same time. This can lead to many transactions arriving at the same time, causing congestions.

If that happens, the system will slow down leading to higher gas prices for users who want their transactions to be mined relatively fast. A solution to this problem needs to be found for a system with many users.

By creating multiple copies of the data handling contract, each newly created agreement can be designated to the contract with the least amount of active agreements. This way, it is less likely that too many transactions are sent to the same contract at the same time.

A load balancer will keep track of the number of agreements each contract currently handles. When an agreement becomes inactive, the contract will tell the load balancer to reduce their number of agreements by one. While the information about that agreement is still available, no transactions can be made.

There is no way of deciding how many contracts are enough to handle all possible transactions in the system. Therefore, someone needs to maintain the system, adding more contracts to the pool to deal with increased load.

This could be quite costly, depending on how often it had to be done. A library that provide the functions for these contracts could reduce this cost. But it will probably slightly increase the cost of each function call as can be seen in 7.14.1.

References

[1] (2018). Blocket.se secure package, [Online]. Available: `https://www.blocket.se/blocketpaketet/blocketpaketet.htm`.

[2] V. Buterin *et al.*, “Ethereum white paper,” *GitHub repository*, 2013.

[3] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[4] (2018). Proof of stake faq, [Online]. Available: `https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ`.

[5] J. Choi. (2018). Ethereum casper 101, [Online]. Available: `https://medium.com/@jonchoi/ethereum-casper-101-7a851a4f1eb0`.

[6] R. C. Merkle, "Protocols for public key cryptosystems," in *Security and Privacy, 1980 IEEE Symposium on*, IEEE, 1980, pp. 122–122.

[7] K. Wüst and A. Gervais, "Do you need a blockchain?" *IACR Cryptology ePrint Archive*, vol. 2017, p. 375, 2017.

[8] (2016). Hyperledger whitepaper, [Online]. Available: `https://wiki.hyperledger.org/groups/whitepaper/whitepaper-wg`.

[9] (2018). The dao, [Online]. Available: `https://github.com/slockit/DAO`.

[10] S. Falkon. (2017). History of the dao, [Online]. Available: `https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee`.

[11] P. Daian. (2016). Analysis of the dao exploit, [Online]. Available: `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`.

[12] D. Siegel. (2016). Understanding the dao hack, [Online]. Available: `https://medium.com/@pullnews/understanding-the-dao-hack-for-journalists-2312dd43e993`.

[13] A. Quentson. (2016). Dao attacker interview, [Online]. Available: `https://www.ccn.com/exclusive-full-interview-transcript-alleged-dao-attacker/`.

[14] (2018). Smart contract best practices, [Online]. Available: `https://consensys.github.io/smart-contract-best-practices/`.

[15] (2018). Consensys, [Online]. Available: `https://new.consensys.net/`.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 254–269.

[17] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," NDSS, 2018.

[18] M. Vladimirov and D. Khovratovich. (2016). Vulnerability of approve function, [Online]. Available: `https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4`
`jp-RLM/edit#heading=h.m9fhqynw2xvt`.

[19] (2016). King of the ether throne, [Online]. Available: `http://www.kingoftheether.com/postmortem.html`.

[20] (2018). Parity github, [Online]. Available: `https://github.com/paritytech/parity`.

[21] H. Qureshi. (2017). Parity multi-signature 1st hack, [Online]. Available: `https://medium.freecodecamp.org/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce`.

[22] A. Akentiev. (2017). Parity multi-signature 2nd hack, [Online]. Available: `https://medium.com/chain-cloud-company-blog/parity-multisig-hack-again-b46771eaa838`.

[23] (2018). Parity contract killed, [Online]. Available: https://github.com/paritytech/parity/issues/6995.

[24] (2018). Canya, [Online]. Available: https://canya.io/assets/docs/WhitePaper.pdf.

[25] (2017). Shipchain, [Online]. Available: https://shipchain.io/shipchain-whitepaper.pdf.

[26] (2018). Zeppelin solidity, [Online]. Available: https://github.com/OpenZeppelin/zeppelin-solidity.

[27] (2018). Truffle framework, [Online]. Available: http://truffleframework.com/.

[28] (2018). Mist, [Online]. Available: https://github.com/ethereum/mist.

[29] (2018). Metamask, [Online]. Available: https://metamask.io/.

[30] (2018). Solidity coverage, [Online]. Available: https://github.com/sc-forks/solidity-coverage.

[31] (2018). Oyente github, [Online]. Available: https://github.com/melonproject/oyente.

[32] (2018). Erc20 token, [Online]. Available: https://theethereum.wiki/w/index.php/ERC20.

[33] (2018). Postnord insurance, [Online]. Available: https://www.postnord.se/skicka-brev-och-paket/inrikes/standard/postpaket.

[34] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[35] (2018). Metamask/web3 1.0 event bug, [Online]. Available: https://github.com/MetaMask/metamask-extension/issues/2350.

[36] (2018). Solidity, [Online]. Available: https://solidity.readthedocs.io/en/develop/.

[37] (2018). Solidity common patterns, [Online]. Available: http://solidity.readthedocs.io/en/v0.4.21/common-patterns.html.

[38] (2018). Ethereum stats, [Online]. Available: https://ethstats.net/.

[39] (2018). Dbschenker smartbox, [Online]. Available: https://www.dbschenker.com/dk-en/products/special-products/sensortechnology-smartbox.

[40] (2018). Ethereum energy consumption, [Online]. Available: https://digiconomist.net/ethereum-energy-consumption.

[41] Nxt community, "Nxt whitepaper," 2014.

[42] K. Sedgwick. (2018). 51% attack on verge, [Online]. Available: https://news.bitcoin.com/verge-is-forced-to-fork-after-suffering-a-51-attack/.

[43] (2018). Ethereum gas station, [Online]. Available: https://ethgasstation.info/.