Delft University of Technology

A bachelor end project commissioned by:
Cirsecon

MarketPalace: a Sybil-Resistant and Decentralized Marketplace

Authors:
Dirk van Bokkem
Tat Luat Nguyen
Justin Segond
Naqib Zarin

Supervisors:	*Roles:*
Ir. Bruno Kimura	Client
Dr.ir. Stefanie Roos	Coach
Ir. Otto Visser	Coordinator
Dr.ir. Huijuan Wang	Coordinator

Summary

Fraudulent behavior within online marketplaces is a prominent but unsolved problem. Most marketplace operators try to mitigate this behavior by serving as the central authority. This approach requires user data collection and is not privacy-friendly. In an attempt to build a foundation for solving the fraud concerns and privacy issues, this paper elaborates on the design and implementation of a simple marketplace system using peer-to-peer (P2P) technology in combination with a Self-Sovereign Identity (SSI) solution. The P2P network ensures no single points of control, reduces risks of a big data breach and simply costs less to operate. The SSI solution makes sure that users cannot create multiple accounts to whitewash their dishonest behavior. Ensuring every user has only one identity makes the platform Sybil-resistant. In contrast to other identity verification systems used in marketplaces, such as Facebook Login, SSI aims to put the user in control and to not collect personal data. Users know what data are asked and give explicit consent for each request. This user-centric approach makes them privacy-friendly. Reaching Sybil resistance without having a central authority in a marketplace has not been done before. In the future a reputation system can be built on top of the Sybil-resistant P2P system, ensuring users' behavior can not be whitewashed.
Several methods are used during the design and the implementation process. They include the Scrum framework, MoSCoW prioritization and Class-Responsibility-Collaboration cards. Git was used for version control while code quality was kept high through a custom CI setup. Additionaly, every merge request required at least two approvals to ensure thorough code review. This resulted in an application that is both Sybil-resistant and privacy-friendly.

Preface

This Thesis describes the work we have done in 11 weeks on the Bachelor End Project. It is part of the Bachelor Computer Science and Engineering program at Delft University of Technology. The goal of this project was to create a privacy-friendly marketplace that is also Sybil-resistant. Our client was a planned start-up named Cirsecon and the contact person was ir. Bruno Kimura. Our coach was security expert dr.ir. Stefanie Roos.
At the time of writing, the marketplace is still in development, making it not presentable to end users. However, we are proud of the end product. It was very challenging and it brought out the best in us. None of us had experience in creating P2P networks. We value privacy-friendly applications and hope that this product will be further optimized.
This product could not have been brought to a good end without our coach and client. Stefanie, thank you for always being available to help us. You made us be more critical about our own work, leading to a better product. We could not have wished for a better coach. Bruno, thank you for your patience and positive attitude. You made us feel comfortable knowing we could always walk into your office and ask questions. At the same time, you gave us freedom to make our own decisions. Thank you both for involving us in the privacy movement.

Dirk van Bokkem
Tat Luat Nguyen
Justin Segond
Naqib Zarin

Delft, June 2019

Contents

1 Introduction

Since the arrival of the internet, new ways of connecting supply and demand have emerged in the form of online marketplaces. In a place where buyers and sellers come together, users have different motives and some form of regulation is needed to avoid malicious use of the system. Just as in the real world, online marketplaces are a breeding ground of fraudulent behavior. Sellers can lie about their products, buyers can withhold money. Therefore, an indication about the trustworthiness of users in an online marketplace has become more of a necessity than something that is just nice to have.
Scientists have done a lot of research on online marketplaces. Nevertheless, online marketplaces are still not fraud-proof. Users can, for example, create multiple accounts (Sybils) and initiate fake transactions between the Sybils. This will result in a high reputation value and misleads honest users. The ideal marketplace would therefore not only contain a reliable reputation system, it would also be Sybil-resistant.
Traditional online marketplaces require data collection and this is becoming a sensitive topic. Scandals like the one where Facebook gave personally identifiable information of 83 million users to the data firm Cambridge Analytica without user's consent made people more aware of the fact that they don't have control over their digital identity. Self-Sovereign Identity (SSI) aims to put users back in control over their digital identity. In practice this means that users can choose what type of personal information they want to disclose, as well as how and with whom. A comparative study on currently available solutions has already been done by the group [1]. One of these solutions will be used as a third-party identity platform to prevent users from creating multiple accounts without collecting personal information.
This report describes the attempt to implement the marketplace which we proposed in our research paper (Appendix A). Given the scope of this project our coach advised us to leave out the reputation system and focus on the decentralized marketplace in combination with an SSI solution. So far no marketplace ha integrated an SSI solution and therefore is Sybil-resistant without collecting personal data.
First, we introduce the problem definition and problem analysis in more detail in chapter 2. This results in a list of requirements. After that, the process will be discussed in chapter 3. Here the division of roles, used tools, methodology and planning is explained. Then in chapter 4, we discuss how the architecture of the marketplace and its different components were designed. This is followed by the actual implementation of the design in chapter 5. All the before-mentioned chapters will not include evaluation. We evaluate on the final product and overall process in chapter 6. Here we elaborate on subjects as code quality, testing and performance. This chapter is followed by discussion and future work in chapter 7. Finally, the conclusion can be found in chapter 8.

2 Problem definition and analysis

This chapter will first explain the given problem by our client that we need to solve. This is followed by our analysis of the problem. Finally, this will result in a list of requirements for the final product.

2.1 Problem definition

The project description we received from our client Cirsecon is as follows:

> *Nowadays systems such as Marktplaats.nl are excellent for connecting people who need to buy and sell second hand products. However, there are still risks in digitalization of identities, such as scammers selling non-existing goods. Even after they are discovered and thrown out they can still rejoin the platform with new identities. These risks prevent legitimate users from making the most out of the platform. The goal therefore is to prevent fraudulent users from joining the system as well as misusing the reputation system (transactions between invalid accounts).*
> *The end-goal is to create a Sybil-resistant marketplace with a reliable reputation system, to prevent above-mentioned issues. However, due to time constraints the scope of this project would be focusing only on the underlying architecture and implementation of the verification and reputation system. The creation of the marketplace itself, to be more specific the front end and the wrap around these two components, is out of the scope of this project.*

In short, our client asks us to answer the following question: *How do we build an online marketplace with a reliable reputation system and where users cannot create multiple accounts?*

2.2 Problem analysis

The main question given by the client can be answered by considering the following three sub questions:

1. *How do we prevent users from creating multiple accounts?*
2. *How do we create an online marketplace?*
3. *How do we create a reliable reputation system?*

We have spent the first two weeks on researching these three topics (Appendix A) and found that we can use a Self-Sovereign Identity solution to address the first problem. This way we will not need to store personal information. After talking to our coach we decided to leave out the reputation system. We needed our time to find answers on the first two sub-questions. This had to be discussed with our client as well, since he initially aimed for the reputation system. He agreed with the importance of Sybil-resistance within the system and approved of us laying the foundation of an online marketplace. Besides, in this research we have also explored different marketplace network infrastructures. We have concluded that the most cost-efficient solution which preserves user's privacy is a decentralized one. This founding changes the second sub question to:

How do we create a decentralized marketplace?

Therefore, the goal of this project is to build a decentralized marketplace with a third-party identity platform to prevent users from creating multiple accounts. For the sake of simplicity we will call the platform MarketPalace from now on.

2.3 Requirements

After deciding what we are going to build, we listed down all the requirements using the MoSCoW prioritisation format. This was approved by both our coach and client.

2.3.1 Must have

1. MarketPalace must only store personal data of users for authentication purposes
2. MarketPalace must store user data in a way such that it is not linkable to a person
3. MarketPalace must have a very simple web interface for testing and demo purposes
4. Users must sign up for MarketPalace using a third-party identity platform in a way such that creation of multiple accounts is prevented
5. Users must be able to enter MarketPalace without using a third-party platform if they have signed up before
6. MarketPalace must be affordable and scalable
7. Market listings should be stored solely on user's devices
8. Users must be able to establish a connection with other peers
9. Users must be able to retrieve all market listings from other peers
10. Users must be able to add new listing of offered items
11. Users must be able to add new listings of wanted items
12. Users must be able to delete listings that belong to them
13. Listings must contain the following details:
    - Title
    - Description
    - Price
    - Hashtag
    - Image

2.3.2 Should have

1. Users should be able to search for wanted items with search terms
2. Users should be able to search for wanted items with hashtags
3. Users should be able to indicate interest in a market listing
4. Listing-owner should be the only one seeing the bids on the listing such that creation of multiple accounts is prevented
5. Users should be able to remove a bid whenever they want
6. Users should be able to update the following details of a listing:
    - Title
    - Description
    - Price
    - Hashtag
    - Image

2.3.3 Could have

1. Users could be able to see a reputation (which is an aggregation of ratings) of other users on MarketPalace
2. Listing-owners could be able to open a chat channel with other users that indicated they are interested in the item
3. Listing-owners could indicate that the item is reserved
4. Users that conducted business could indicate that they made a deal
5. Users that conducted business could be able to give each other a rating after they made a deal

6. Users that conducted business could not be able to see the given rating from the counter party
7. Users that conducted business could only increase their reputation if the given rating of both parties was positive

2.3.4 Won't have

1. Users will not be able to pay or receive money through the system

3 Process

In this chapter, the process of the project will be discussed. This includes the product planning, division of roles, workplace, meetings, methodology and ways of communicating.

3.1 Organisation

For the duration of the project, the team worked every day at YES!Delft from half past nine to six. Since the product is made for planned start-up Cirsecon, we managed to get a workplace at the tech incubator next to TU Delft. As mentioned before, the first two weeks were reserved for research on the subject which resulted in a research report (Appendix A). The remaining eight weeks were used to work on the product and to write the final report. Every week, the team had a meeting with the coach on Friday (with the exception of the midterm meeting). In these meetings, the team elaborated on the process and problems that were encountered. These meetings were very helpful in gaining insights and solving these problems. Next to that, bi-weekly meetings were held with the client on Fridays, to keep the client up-to-date with the progress of the product, and to learn about any desired adjustments.

3.1.1 Roles

All the team members were actively involved in the design, implementation and writing process. There were no specific roles required for these tasks. However, we needed to make sure every aspect of the project was taken care of. Therefore, the following distinction of roles was made:

Dirk van Bokkem
Lead Testing / Minutes Secretary

Tat Luat Nguyen
Lead Architecture

Justin Segond
Lead Code Quality

Naqib Zarin
Scrum Master / Lead Communication

These roles included setting up frameworks and making sure every team member finished their tasks for the specified aspect. The roles do not implicate that team these members were solely responsible for finishing tasks within the given aspect.

3.1.2 Responsibilities

- *Lead Testing*
  In order to maintain high code quality, code needs to be tested. Lead testing is in charge of making sure enough tests are written by everyone of the team. Next to that, he is mostly in charge of setting up the testing environments and showing the coverage.
- *Minutes Secretary*
  The Minutes Secretary is responsible for taking notes during meetings and making these readable and available for everyone in the team.
- *Lead Architecture*
  Lead Architecture makes sure the structure of the code is logical and overall coherent. He is mainly responsible for the class diagrams and can intervene when questionable design decisions are made.

- *Lead Code Quality*
  Testing is not the only aspect of keeping the code quality high. Efficiency, naming conventions, commenting; Lead Code Quality is in charge of all of these. He needs to keep an eye on keeping the code bug-free.

- *Scrum Master*
  With the Scrum approach, a Scrum master is needed. He makes sure the team sticks to the plan and follows the Scrum rules. He is in charge of creating and maintaining the backlog and retrospective (apart from filling in, which is done by the whole team). He also keeps track of the Scrum board which needs to be updated throughout the week.

- *Lead communication*
  Lead Communication is in charge of the communication within the team as well as towards our coach and client. Communication within the team needs to run smoothly and not divert to much into unnecessary chit-chat. Also, meetings need to be scheduled and dates communicated to the coach and client. Any messages from them must be shared with the group by the Lead Communication.

3.1.3 Tools

Collaboration tools
As mentioned before, the team worked together at YES!Delft every day, so most communication was directly in person. Outside of these meeting hours, the team communicated using different tools. For simple communication such as sharing meeting places and times, the team communicated via WhatsApp. Next to that, a lot of communication went through GitLab, in addition to it being used as source control. Here team members would comment on each other's work and process, as mentioned in section 3.2. Results from discussions and notes of meetings were stored in a shared Google Drive, as well as diagrams, design decisions, the schedule and Scrum-documents. Reports were written in Overleaf.

Development tools
MarketPalace has two major components; the "door process" ensuring one-time-entrance and the marketplace itself. To ensure users entering one time only, the team decided to make use of the Self-Sovereign Identity solution IRMA[1]. The reason for choosing IRMA is laid out in section 4.1. IRMA was written in Go, which encouraged the team to work in Go as well. However, IRMA also has a JavaScript API that fitted more to our needs, as we needed to make a simple interface. The team chose to write the door process in JavaScript with HTML pages and NodeJS for setting up a server. Testing the door process was done using the Mocha testing framework, accompanied with multiple libraries such as proxyquire for mocking. For the marketplace part of the system, the idea was to create a P2P system using libp2p[2]. The team's preferences were Python and Go, but the Python implementation of libp2p is still under development. Go on the other hand is the most supported language for libp2p, which resulted in us choosing Go with the JetBrains GoLand IDE. Tests were written in Go as well, as this is incorporated in the language. Also, we used Atom as our source code editor for JavaScript, HTML and CSS. In different areas of the program, encryption keys were needed, which were generated with openssl.

[1] I Reveal My Attributes: `https://privacybydesign.foundation/`
[2] libp2p: `https://libp2p.io/`

3.2 Methodology

To bring this project to a successful conclusion, various methods were used to steer the team in the right direction during design and implementation. These methods will be laid out in this section.

3.2.1 Class-Responsibility-Collaboration (CRC)

From the requirements laid out in section 2.3 the team made CRC cards to find class candidates from which we could set up a code structure. The idea is to first highlight all nouns in the requirements document and write these down. By doing this, doubles or synonyms can be found and removed so that you are left with potential class names. The nouns are then divided into conceptual and non-conceptual nouns. The non-conceptual nouns are words like *application*; they will not be actual components of the system. The conceptual nouns however, are potential classes. Next, the verbs are highlighted and written down. These verbs represent the functionality the potential classes from the previous step will have. For each noun, all verbs that belong to it will be listed. Most importantly the responsibilities of these classes will be listed. Here the connection to other nouns will be made. For example, if we have 'hash function' as noun and 'transforms' as verb, the corresponding other noun will be 'user attributes'. By doing this, the potential different classes can be discovered, alongside their responsibilities to other classes.

ListingManager	
Parent class: none	
Subclasses: none	
adds listings	PeerManager
takes down listings	PeerManager
retrieve listings from network	PeerManager
loads listings	memory
save listings	memory

Figure 1: Example CRC card: ListingManager class that has responsibilities with PeerManager.

Lastly, the actual CRC cards are made. An example can be seen in Figure 1. For each potential class a card is made that holds the name of the class, the parent classes, subclasses and lastly the functionality (verbs) with their responsibilities (other nouns). These cards were used to discuss about the structure of the system.

3.2.2 Scrum

The main methodology used in the project was the Scrum-framework. With Scrum, the team worked in weekly sprints. Starting each sprint, a planning was set up known as the backlog and by the end of the week the sprint was reviewed in the retrospective. After the requirements document and CRC cards were made, a general plan for realising the product was made in the form of milestones. There was one milestone for each week. By finishing all milestones, all the must- and should haves of the requirements document will be included in the final product.

Backlog
When creating the backlog each week, the team made a plan to be able to finish that week's milestone. User stories were made to conceptualize what we needed to do and why. Within each *user story* different tasks were formulated, together with their *priority*, *estimated time* to finish the task, *responsible member* for the task and *actual executor* of the task. Making the distinction between *responsible member* and

actual executor was in line with our role division as explained in section 3.1.

GitLab
All tasks created in the backlog were added as tickets to the Scrum-board in GitLab, where the team could keep track of the process. The board had the following four columns: *Sprint Backlog*, *In Progress*, *In Review*, and *Finished*. Naturally, all created tickets were added to *Sprint Backlog*. During the sprint, when team members started working on a task, the corresponding ticket was moved to *In Progress*. When a merge request (MR) was made, the ticket was moved to *In Review*. After merging, it was moved to *Finished*, where it would stay until the end of the week. At the end of the sprint, when the retrospective was made, finished tickets were closed. Tasks that emerged during the sprint were added as new tickets on the board. These tasks were added to the retrospective so that the shift of worked hours in that sprint was insightful.
Reviewing the code was an important aspect of our process and maintaining the code quality. After finishing one or more tasks, an MR was made on GitLab to have the other team members review the code. The following rules applied to our code reviewing:

- Use MR-template:
  - Introduction
  - Details
  - Purpose
  - Logs/Screenshots
  - Potential problems
- Mention corresponding tickets (by using #*corresponding-ticket*)
- A maximum of 3 corresponding tickets
- At least 2 approvals
- Review MR within 24 hours
- MR not merged on Friday 16:00; shift to next week

Retrospective
Every Friday, the team reviewed the sprint and documented this in the retrospective. The goal was get insight in how we could improve the process. All tasks from the backlog were written in the retrospective, with additional columns of *actual time spent*, *actual members worked on* and a column stating if the task was *finished* or not, together with a *note* explaining why.

3.3 Product Planning

In any software project the planning is important and should be addressed before any groundwork can be laid. This section describes the process of creating a product and following through with it until the product is delivered to the client. The product planning serves as a basis for the team to decide on the work that has to be distributed during the working cycles. Since we used Scrum, this working cycle is a sprint of five days. After the research phase of two weeks, the sprints started for which we defined milestones at the end of every week. The aim is to achieve all milestones, each week. Note that writing the report is not a part of the milestones, because we want to keep the report up to date during the process. The milestones have been listed in Table 1.

Milestone	Date	Minimum functionalities
1	10-05-2019	Project setup in Jetbrains GoLand & project conceptualization & IRMA integration & connection between libp2p-nodes.
2	17-05-2019	Basic web-interface "door process" & Hash Database & Add and remove market listing
3	24-05-2019	Wrap-up "door process": hash function and key generator & Retrieve market listings from other peers
4	31-05-2019	Minimal interface for MarketPalace & Connection between "door process" and marketplace & Enter marketplace with key-pair
5	07-06-2019	Add details to market listing (title, price, wanted/offered, etc.) & Indicate interest in market listing (bid) & Seller only one to see bid & Removal of bid
6	14-06-2019	Search function market listing & Update details market listing
7	21-06-2019	Item reserved indication & Deal made indication & Give rating after transaction
8	28-06-2019	Buffer

Table 1: Roadmap of the production process.

4 Design

In this section, three main aspects of the MarketPalace design are laid out: registration, keys, and the market. In registration, the one-time-entrance process with IRMA is explained. Afterwards, we elaborate on the different usages of keys in the system. Lastly, the design of the P2P market system is explained.

4.1 Registration

Authors of the paper 'A Survey of Solutions to the Sybil Attack' have shown that trusted certification is the most popular - some even claim the only - approach that has the potential to completely eliminate Sybil attacks [2]. This requires a centralized authority that must ensure each user is assigned exactly one identity. Preferably, we want to avoid manual or in-person identification processes because they hinder scalability. We want to have a third-party identity platform that can provide us with uniquely identifiable information.

4.1.1 Self-Sovereign Identity

In order to prevent users from creating multiple accounts we need to uniquely identify them. Therefore we need immutable unique attributes. In the Netherlands, the most unique attribute that every person has is the citizens service number (BSN). This is private information and people do not feel comfortable sharing this with companies. Therefore we need a third-party identity platform that is user-centric. This, and the fact that our client wants optimal privacy protection, leads us to the field of Self-Sovereign Identity. Self-Sovereign Identity aims to put users in control over their digital identity. In practice this means that users can choose what type of personal information they want to disclose, as well as how and with whom. We have done a comparative study on SSI solutions before and IRMA turns out to be the most suitable one for MarketPalace [1].

4.1.2 IRMA

The difference between IRMA and non-SSI solutions such as Facebook Login lies in the fact that IRMA offers optimal privacy protection by design because of its decentralized architecture. Non-SSI solutions are organised in a centralised manner because this is commercially interesting. This way they can build up and sell profiles. In contrast to these platforms, IRMA stores the attributes on the users' device. Our research has also shown that IRMA meets ten out of eleven requirements to be a profound SSI solution. Another reason why we choose IRMA is because IRMA users can choose to disclose the attribute we need: the BSN. If this turns out to be a barrier for users to use the marketsystem, we could easily request other attributes from IRMA and find a combination to uniquely identify a user. Other than legal issues, this also influenced our decision to use IRMA instead of integrating DigiD directly in our application. However, IRMA is currently mostly used and supported in The Netherlands. This means international users cannot register for an account using IRMA. This limitation can be overcome by adding more identity verification platforms or verification options in the future.

4.1.3 Database

When users have downloaded the mobile IRMA application they can collect attributes such as date of birth and BSN. Users can import their BSN into the IRMA app with their DigiD. When a fraudster has DigiD credentials of someone else, he or she can still impersonate this person and bypass our authentication mechanism. However, this type of fraud can also happen with a physical passport and will therefore not be investigated further. We still have to make sure that a user cannot create multiple accounts with one BSN number. This indicates that we will need a database where the BSN of users that already entered the marketplace before, are stored. This process needs to be secure, which is why we looked at the well-known phenomenon in security of the balance between confidentiality, integrity and availability

(CIA) [3]. Amazon Web Services are widely known and pursue the CIA balance in their data systems [4]. Therefore we decided to use the Amazon Relational Database Service (RDS)[3].

4.1.4 Hash function

Since the purpose of this database is solely uniqueness verification, we will use a one-way hash function and store this hash. We want to stress why we do not use encryption instead of hashing the BSN. Cirsecon is not interested in and preferably do not want to be able to retrieve the BSN from the hash.
After we hashed the BSN we check whether it exists in the database. If so, it means that the user is trying to register twice. This is exactly what we want to prevent in order to mitigate the Sybil-attack. Figure 2 summarizes the registration process. Note that we call this process the 'door process'. The idea was to enter the marketplace via a single entry point, hence 'door'.

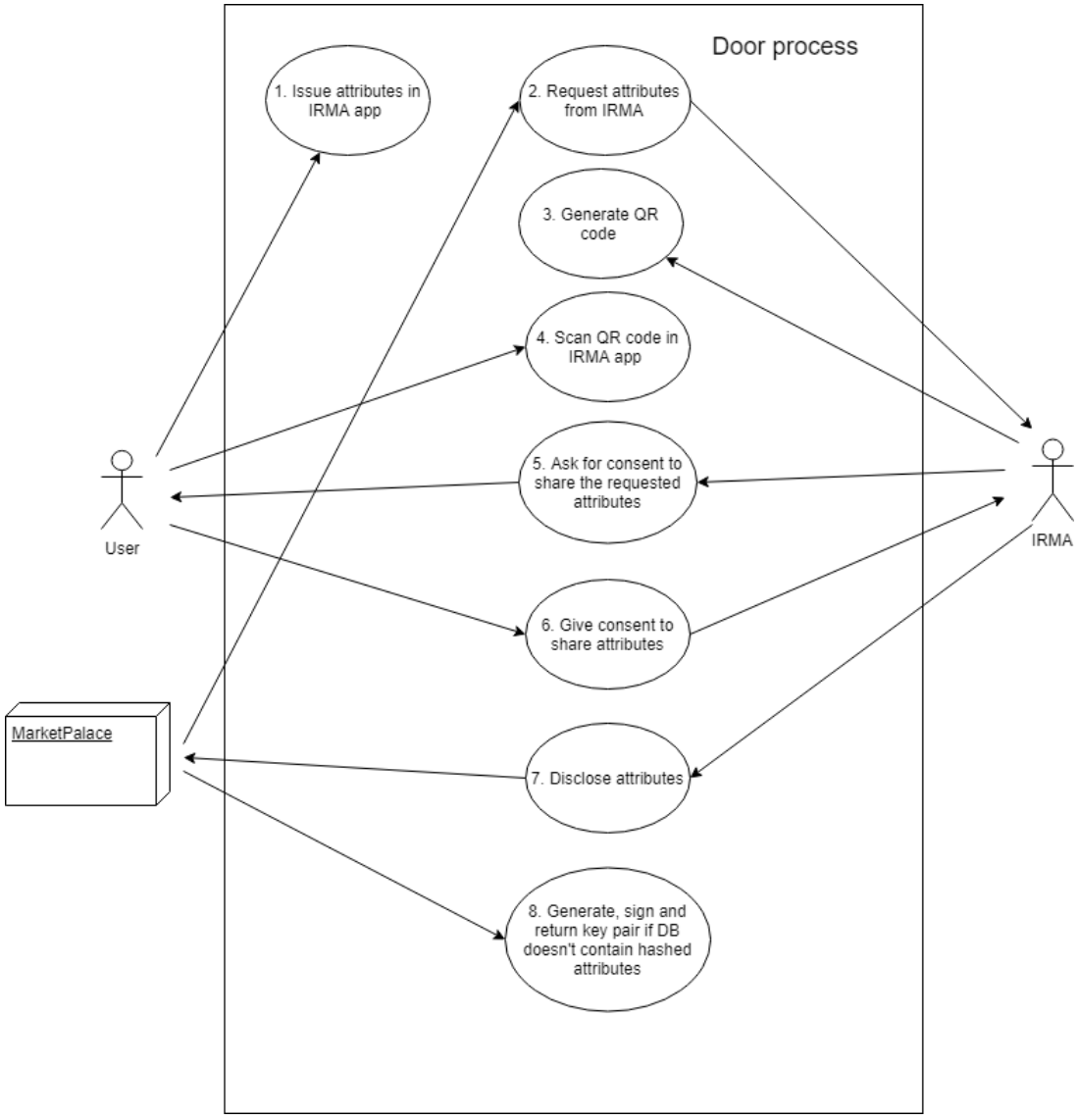

Figure 2: Use case diagram of the door principle.

[3] https://aws.amazon.com/rds/

4.2 Keys

In a marketplace sellers and buyers have to communicate. In the interest of the user, this communication should be secure. In order to make use of the marketplace, it is essential that each participant holds ownership over an object that can be used to prove the authenticity of the sender and receiver. Nowadays we make use of cryptographic keys. An example is the RSA key pair. The RSA private key is used to sign messages that you send to other participants and the RSA public key is used to verify the message's authenticity and integrity. In the same way our marketplace participants also need to make use of RSA keys.
In the next subsections we first explain how we generate RSA keys. Then we will briefly explain how we sign the user's public key with Cirsecon private key in order to verify it's origin later on. This will be followed by a section on why and how we import keys. Finally, we will discuss how users should authenticate after they already imported keys.

4.2.1 Key generation

We still have not discussed when to generate this key-pair. In our proposal, this happens after we ensured that the user's hashed attributes do not exist in our database yet (see Figure 3). If this answer is negative, then we assume this is a new user and generate a key-pair.

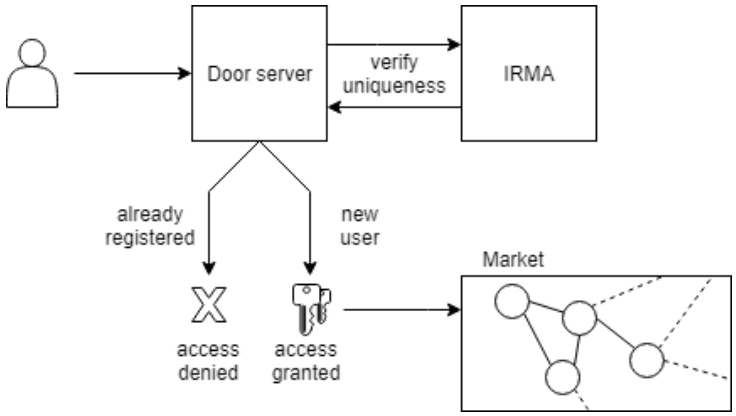

Figure 3: The keys are generated after the user has been identified as a new user.

4.2.2 Key signature

A key can be generated by anyone. A random key defeats the purpose of having a door to verify uniqueness. Therefore, a process is needed to make sure that this key is indeed unique and generated by Cirsecon. This process is called key-signing. The next step for our application is to make sure that each newly verified person can acquire a genuine key that is issued by Cirsecon. In practice this means that the user's public key will be signed by Cirsecon's private key to generate a signature.

4.2.3 Key import and verification

The user will thus receive a private key and signature. With the private key SK_{user} and the signature Sig, a user can now make use of the market. However, there still needs to be a process to securely import SK_{user} and Sig into the user's device. Also, Sig needs to be verified before the application can proceed with importing the key.
Sig is a signed message with the message M being the hash of the public key PK_{user} of the user. To verify Sig, M needs to be sent together with the signature. However, since we are making use of RSA keys, the private key also contains information about the public key. This means PK_{user} can be then retrieved from the given SK_{user} and used for verifying the signature.

During the import, these steps happen subsequently in order to verify Sig:

1. PK_{user} is computed from SK_{user}, which is given by the user.
2. To recreate M, PK_{user} is hashed with the hash function that is used by both the key signing process and the verification process.
3. The Cirsecon public key $PK_{Cirsecon}$, which is the counterpart of the Cirsecon private key $SK_{Cirsecon}$, that is used in the key signing process, is used to verify the signature given by the user. The verification function $Verify(PK_{Cirsecon}, M, Sig)$ is called.
4. If Sig is indeed a signature of Cirsecon for M, the verification function returns true. Otherwise, it returns false to indicate tampering or having an unsigned key.

If the transition through these steps are without problems and step 4 returns true, the SK_{user} and Sig are encrypted with a password chosen by the user and stored securely in the user's device. This encryption makes sure that even when the device is stolen and malicious actors have gained access to the operating system, we still have the application authentication barrier. The signature given earlier is also stored in case the user node has to prove that it belongs to the marketplace in the future.

4.2.4 Password

After the keys are verified and imported to be saved on the user's device, they need to be securely retrieved every time the user makes use of the marketplace. The user is asked for a password which is used to encrypt the private key. This is the password that was created during the key import stage. Without this password, the user cannot enter the market and see listings. The user then has to either create a new key (registration if he/she has never used the marketplace), or import the key and signature in order to make use of the authentication system. The application uses the password to decrypt the private key stored in the device. However, in the scope of this project we decided to modify this system to make development more feasible. In our current system, instead of using a password, we simply ask the user to fill in the first 5 characters of the private key, which we will use to match the private key and if it is correct, we will allow the user to enter the market and make use of the saved private key.

4.3 Market

In this section we explain the design of the market aspect of the system. We elaborate on the P2P network and the features.

4.3.1 Network

The market module uses a P2P network architecture. A P2P network architecture is a good fit for a customer-to-customer marketplace since the communication can go directly between both parties without interference from a third party. Due to this, there is no chance for a company to spy on the private data of the users and sell it. Users are therefore in full control of what they share with others. Another reason why we chose for the P2P architecture is because it is cost-efficient. In a traditional centralized marketplace all of the data is handled on the server which brings along costs for network infrastructure. For our P2P side of the market aspect we only run bootstrap servers. These bootstrap servers are needed in order for peers to initially find other peers. Once they have established a connection with other peers, this server is no longer needed. This means that the bootstrap servers do not store much data and are not very demanding of resources which makes them cheap to run.
Since this is a P2P network, all connections are done between peers and therefore, there are no servers involved. Listings are replicated and stored on multiple nodes so that if one node is not online, there are still other online nodes serving the content. In an example below, node 1 is looking for a listing of a bike. It sends requests to the 20 closest nodes and if these nodes contain a listing of a bike, they will respond (see Figure 4).

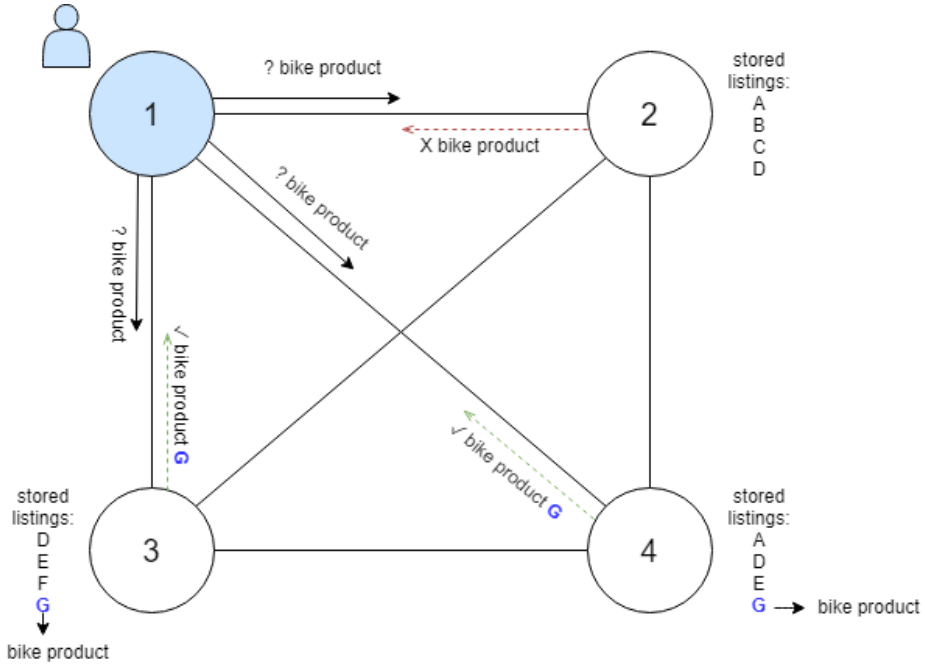

Figure 4: P2P example of finding a bike in the network.

4.3.2 Features

After the IRMA authentication and key verification, users can create listings and see the listings they own. The main focus has been functionality rather than a good user interface. However, a basic interface was needed to demonstrate the functionality of the marketplace. The marketplace has the pages *My listings*, *Other listings*, and *My favorites*.

My listings
Here users can create listings by filling in the following details:

- Title
- Description
- Price
- Tags
- Image
- Offered/Wanted

User must be able to create a listing on the webpage. Also, all the listing the user has created will appear. Here, users are able to edit and remove listings. Next to that, users will see the bids on their items.

Other listings
Self-evidently, listings of other people on the network are shown on the page *Other listings*. Users can use this page to find offered and wanted items. Offered items are contained in a box highlighted in orange. Wanted items have a grey box. In case of offered items, users can place a bid which will be visible to the listing owner only. The user that placed the bid can find the listing in his or her list of favorite listings on the *My favorites page*, together with the bid amount. The listing owner can choose to start a chat with one of the bid placers to complete the transaction. Users also have the ability to search listings by

using the search box. At this time, search terms are matched with the tags of listings. Next to placing bids, users can also choose to add listings directly to their favorites list so that they can review them later.

My favorites
Listings that are added to the favorites list, will be shown on the *My favorites* page. Listings that were added because of a bid will have an additional indication of the bid. Users can also change their bids on the listings on this page. Next to that, the user is able to remove listings from their favorites on this page.

5 Implementation

After the design process of different components of MarketPalace, we had to implement them. This chapter gives an overview of the implementation of these components. We first discuss the door process where we explain the different components including the door server, IRMA server and database. In the second section we explain how we implement the P2P marketplace.

5.1 Door process

5.1.1 Door server

As explained in section 4, the door process handles the registration part with IRMA. This functionality is implemented by *doorserver.js*, to which the client-side web pages are connected. The door server is an HTTPS server implemented in NodeJS. Its only functionality is the communication between the signup webpage and the NodeJS components that handle the door process. The signup webpage opens a socket connection with the door server and sends a message that initiates the disclosure process when the user clicks the *Disclose attributes* button. The door server in turn, sends multiple messages to the client-side during the door process. For example, the changing of the QR-code image is initiated by the door server, but also the generated keys are sent from the door server to the signup webpage; both of which will be discussed below.

5.1.2 IRMA server

To use IRMA, multiple components were needed. First, the IRMA server needed to be set up. The *irmago*[4] implementation by Privacy by Design holds the *irmaserver* library from which you can start the server. As the name states, the *irmago* implementation is written in Go. Next to the *irmaserver*, *irmago* also holds a client library *irmaclient*. Since we decided to implement a web interface, we decided to not use the Go implementation of the *irmaclient*, but rather use the *irmajs*[5] client of Privacy by Design.

The IRMA server makes it possible to perform IRMA sessions, such as disclosing attributes. When the disclosure session is requested client-side, the door server calls

```
doDisclosureSession(websocket)
```

in *irma_script.js*, which starts the IRMA attribute disclosure process. The websocket is used to send messages to the client-side. The disclosure session works by calling the following functions (that interact with the IRMA server) of the *irmajs* library:

```
irma.startSession(irmaserver, request, authmethod, key, requestorname)
```

irmaserver :	URL to IRMA server
request :	the disclosure request; BSN
authmethod :	set to none, see security section 5.4
key :	unused because *authmethod* set to none
requestorname :	unused because *authmethod* set to none

This method returns a Promise, which is a JavaScript object that handles asynchronous tasks. The returned Promise resolves to a QR-package, which holds the data to form a QR-code and the session token for this specific session. The QR-data is sent to the client-side to be displayed and

[4] https://github.com/privacybydesign/irmago
[5] https://github.com/privacybydesign/irmajs

```
irma.waitConnected()
```

is called. It returns a Promise that handles the connection of the user with the IRMA server through their (mobile) IRMA application. The Promise resolves when the user is connected, which is an intermediate state of the disclosure session. The user still needs to give consent to disclosing the attributes. Therefore,

```
irma.waitDone()
```

is called, which again returns a Promise. The Promise is resolved when the user discloses the attributes through the IRMA application. This initiates the call to the IRMA API session result endpoint. The result is retrieved from the IRMA server and will be used to verify the uniqueness of the user in *database_script.js*.

5.1.3 Database

We use an SQL client to operate our remote Amazon Relational Database. Our database is very simple. We have only one table *hash* in which we only have one column *idhash*. Next to that we have a script *database_script.js* where we handle the communication between our application and the database. This includes making connection with the database, looking up the hash and inserting a hash. We make the following function call to lookup a hash:

```
SELECT * FROM doorschema.hash WHERE idhash = hashedAttr
```

This call returns a list of all the hashes that are identical to the variable *hashedAttr*. This variable holds the calculated hash from the user's BSN that we received from IRMA. If this list size is bigger than 0 we tell the user that he or she has already entered the marketplace before and should use the given keys. Otherwise, if this list is of size 0 it means that this user is new. Then we insert the hash using the following call:

```
INSERT INTO doorschema.hash (idhash) VALUES (hashedAttr)
```

When this call is made a key-pair is generated and signed.

5.2 Market

In this section, we will explain the technical implementation of the code in detail. The code discussed here enables the key import and verification, authentication and marketplace features in the Design section (see section 4). An overview of the classes and packages are explained first and the packages and classes themselves subsequently.

5.2.1 Overview

Each package is a group of classes that works together to achieve the same responsibility for the marketplace. A responsibility could be connecting, maintaining and communicating with other P2P nodes. In general, we have 3 levels of packages (See Figure 5).

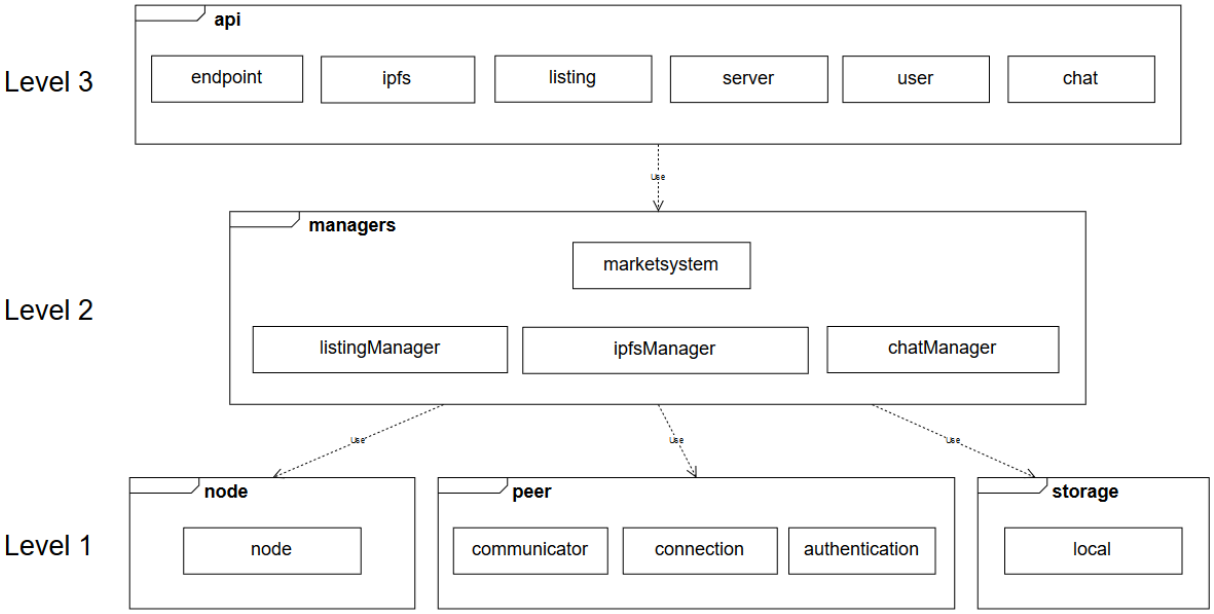

Figure 5: UML diagram of the class overview.

Level 3 is the highest level which is used by the user interface. This level contains the *api* package. This package contains API implementations so that it can be called by the user interface.
Level 2 contains the *managers* package. This package contains object oriented classes, which we call managers. They all have their own responsibilities. The *listingManager* manages listings, *ipfsManager* manages P2P storage and *chatManager* manages chat channels. These classes are accessible and regularly used by the level above.
Level 2 makes use of following lower level classes: *node*, *communicator*, *connection*, *authentication* and *local* in Level 1. *Connection* class takes care of connection establishment and maintainance. *Authentication* class takes care of validation, signing and encrypting.
In the next subsections, these classes functionalities will be further explained.

5.2.2 IPFS and libp2p

In this section we will first go over what InterPlanetary FileSystem (IPFS) is, why we chose it, and then discuss how it is implemented in our application.

IPFS
IPFS is a distributed file storage system. Anyone can upload files to IPFS because the files are stored solely on your own device after uploading. When you upload a file to IPFS, the file or directory you uploaded gets saved by the hash of its contents (Content ID). Any user that is part of the IPFS network can then access that file or directory by requesting that Content ID. This can only happen if at least one user in the network is actually storing the requested content. If you shut down your IPFS node after uploading, no one has access to the data. This is not ideal. Therefore, when another user accesses that content a copy of the content gets stored on their local IPFS node. Other users can now also fetch the content from this other node, creating a distributed file system.
We currently use IPFS only for the storage of images. When a user uploads a listing with an image, the image gets added to their local IPFS node. We then store the hashes of the uploaded images in the listings. Other users can then retrieve these images from the IPFS network. IPFS is great for a marketsystem, because when someone is browsing the listings, the images get copied on the users device. This user will now be able to send these images to other users. This process will create a stable marketplace when there are a lot of active users. This is why we chose IPFS.
When the application starts up, a new IPFS node is created in the *node* package. Its responsibility is therefore creating the IPFS node for our application. To interact with this node we chose to use the IPFS core API, as we embed the IPFS node into our own application instead of running one separately. It is possible to run an IPFS node separately and interact with it by its API, but we are then limited by what the IPFS API offers us. By embedding the IPFS node into our own application we gained access to it's underlying networking library which we will discuss in the next section. It is however noteworthy to say that the core API that we use is an unfinished feature and is likely to change in the future. For our project we also chose to use a private IPFS network since that way we only connect to other nodes that actually can have listing images.

libp2p
We have also access to IPFS's internal networking library libp2p because of the way we have set it up. The *peer* package is responsible for directly connecting and communicating with other peers with the use of the libp2p library. The package also manages open connections and incoming connection requests. We try to reuse past streams before opening a new one.
The libp2p library provides us with a lot of functionality that is important to our marketplace. One of the most important features that we use is connecting to other peers by their peer id. Libp2p uses a Kademlia routing table to lookup peers. Using this lookup table, nodes can be found in O(log n). This is great because it lowers the amount of requests and data that needs to be sent with every request.
Another responsibility of the peer package is authentication. Communicating with other peers requires authentication, which is done in the *authentication* class in the peer package.

Authentication class

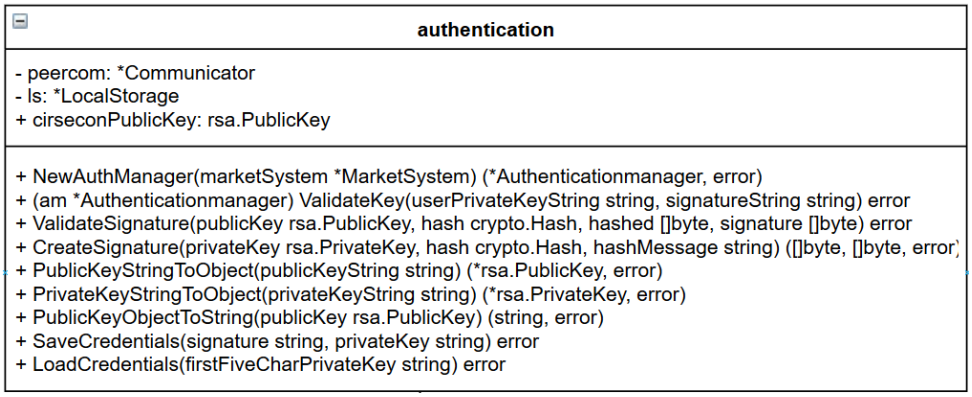

Figure 6: UML diagram of the class authentication.

The authentication class (see Figure 6) handles the authentication for communication between peers. The class also handles the authentication requests from the *api* package. In order to be able to import keys as mentioned in section 4.2.3 and to verify other users' keys, the signature must be verified. This is done via the method *ValidateKey*. This method takes in the input string of the private key, the signature and will do the following procedures:

1. Compute the public key from the given private key
2. Hash this public key with the agreed upon hash protocol
3. Call the method *ValidateSignature* which takes in the Cirsecon public key, the agreed upon hash protocol, the hash from step 2 and the signature given by the user.

ValidateSignature will return an error if the verification is incorrect. It is incorrect when the hash from step 2 is not the same as the hash signed by the Cirsecon private key.
After validating the signatures without problems the method *SaveCredentials* can be called in order to save the private key and signature onto the device. In the future we would like to encrypt this information with a passphrase as well.
Except from validating signatures, the authentication class also has to make sure that when a user wants to make use of the saved private key and signature, they have to give the application the correct password. This is done via the method *LoadCredentials* which for now will take in the first five characters of the key as a simulation for the password. These five characters are checked against the real key to verify if they match. If this method returns without errors, the user has authenticated himself and can start making use of the marketplace (See Figure 7).

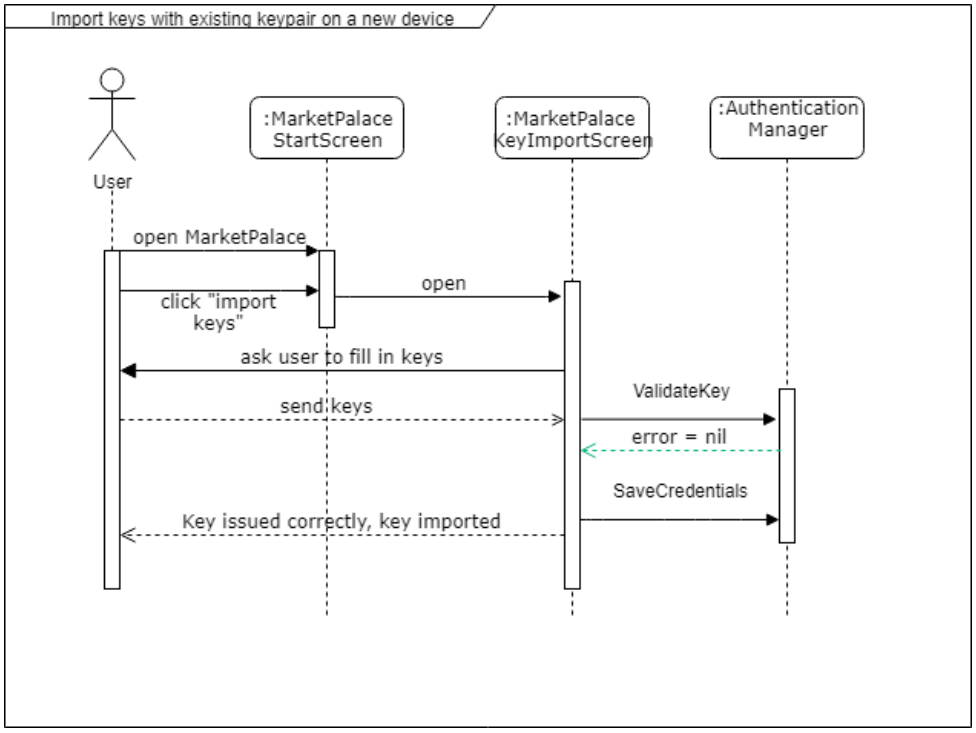

Figure 7: Sequence diagram of the key import process.

The authentication class also contains various methods with different functionalities. The method *CreateSignature* takes a private key, a hash protocol and a message to be signed. It returns the hashed version of the message and the signature which is created from the hashed message and the given private key. The other methods take care of converting the string representation of keys to actual key objects or vice versa, such as *PrivateKeyStringToObject*.

5.2.3 Packets

The *pb* package deals with serializing data to be sent over the network. The package consists of .proto files and go files. First we will explain what Protocol Buffers are, and then we will look into what packets we have created.
For serializing data over the network we have chosen to use Protocol Buffers by Google. Protocol buffers allow us to declare how a packet should look. They provide their own language in which these specification of packets are written (.proto files). A Protocol Buffers compiler then compiles it into the language we use (Golang). We chose Protocol Buffers because of the low overhead compared to other methods such as json. In practice this results in lower bandwidth costs for users. It also supports extending the protocol later on without breaking backward compatibility, which means it can be extended later on. In our testing it also came out as slightly faster than json serialization.

The packets
In this section we will go over the packets we have created for communication between peers. The general packet structure we have created for every packet is as follows:

Packet		
PacketType	enum	the type of packet
Authentication	Authentication	the authorization of this packet
Data	any	any serializable packet defined below

Where Authentication is the following:

Authentication

PublicKey	string	the verified public key given by Cirsecon
PublicKeySigned	string	the signed public key given by Cirsecon
Signature	bytes	the signed contents signed by the private key

As can be seen we send the public key and signed public key along with every packet. This means that any user communicating with this user can check whether the person they are talking to is actually verified by Cirsecon to use the marketplace. We also send a signature of the sent data so users can make sure the verified Cirsecon user actually sent the packet. Next we will go over the packets that will be sent along with the header.

Listing

Owner	string	the peer id of the owner (to be found in the libp2p network)
Id	int32	the id of the listing
Time	int64	the unix timestamp
Title	string	the title of the listing
Price	float	the price of the listing
Description	string	the description of the listing
Tags	repeated string	the tags of the listing
Image	repeated string	the Content IDs of the images on IPFS
Reserved	bool	the reserved status of the item
Wanted	bool	whether the listing is for a wanted item or an offered item
Version	int	the version of the listing
Authentication	Authentication	the authorization of this listing

The packet specifies sending along the authentication again. We do that because in the next packet we also send other peoples listings along with our packet. This way receiving nodes can verify whether the listings that were sent along of other peers are actually of a verified Cirsecon user. Users also sign their listing with their Cirsecon verified private key so no other user can edit their listings.

ListingUpdate

Listings	repeated NetworkListing	The listings to be sent

In this packet we send along any listings we know of from any peer or from ourselves.

ListingsRequest

Id	int32	The listings id to look for (or empty for all).
Tags	repeated string	The listings with given tags (or empty for all)

This packet request listings from a peer. When starting up the application this packet is sent to peers in order to ask them for the listings they know. This packet should be responded to with a ListingUpdate packet.

AddBidRequest

BidOwner	string	the id of the placer of the bid
ListingId	int32	the listing id on which the bid is on
Bid	float	the amount of money that was bid

This packet request to add a bid to the listing of another peer. It sends the message directly to the other peer, so no one else knows about the bid.

Message

PeerIdFrom	string	the peer id who sent the message
PeerIdTo	string	the peer id who the message was sent to
ListingId	int32	the listing id
Content	string	the contents of the message
Time	int64	the unix timestamp the message was sent
Delivered	bool	whether the message has been successfully delivered
Me	bool	whether the message originated from the current user

This packet is used for sending a direct message to another peer over the network.

5.2.4 API

The market module contains an API which is located in the *api package*. It is a simple Go http server that listens for post and get requests. Most of the API calls return json objects. When there are errors with the API call, an array of errors is returned also in json format. The following API calls are supported.

POST /api/listing/create

title	string	the title of the listing
description	string	the description of the listing
price	float	the price of the item
tags	string	comma separated tags
image	file	the image of the item
wantedOroffered	bool	the type of listing

This API call tries to create a listing and add it to the users own listings. It uploads the image to IPFS and stores the hash of the image in the listing.

POST /api/listing/remove

listingid	int32	the id of the listing to remove

This API call tries to remove a listing of the user based on the given listing id.

POST /api/listing/update

listingid	int32	the id of the listing to update
title	string	the new title of the listing
description	string	the new description of the listing
price	float	the new price of the item
tags	string	the new comma separated tags
wantedOroffered	bool	the new type of the listing

This API call tries to create a listing and add it to the users own listings. It uploads the image to IPFS and stores the hash of the image in the listing.

POST /api/listing/get

This API call retrieves the users own listings in json format.

POST /api/listing/getpeer

This API call retrieves the known listings from other peers in json format.

POST /api/listing/gethashtag

searchterm	string	the tags to search for

This API call retrieves the known listings from other peers with the given tag in json format.

POST /api/wishlist/get

This API call retrieves the listings in the users wishlist in json format.

POST /api/wishlist/add

owner	string	the peer id of the owner of the listing
listingid	int32	the id of the listing

This API call tries to add the given listing to the wishlist of the user.

POST /api/wishlist/remove

owner	string	the peer id of the owner of the listing
listingid	int32	the id of the listing

This API call tries to remove the given listing from the wishlist of the user.

POST /api/user/savekeys

privatekey	string	the base 64 encoded private rsa key generated by Cirsecon
signature	string	the signed public rsa key generated by Cirsecon

This API call imports the keys generated by Cirsecon into our market system.

POST /api/user/loadkeys

password	string	the password (first 5 chars of private key)

This API call loads the keys if the given password matches the first 5 characters of the key.

GET /api/ipfs/getimage

hash	string	the hash of the ipfs image to retrieve

This API call tries to retrieve the image given by the hash from the IPFS network.

POST /api/chat/send

peerid	string	the id of the peer to send to
listingid	int32	the id of the listing
message	string	the contents of the message

This API call tries to send a message to a peer about a listing.

POST /api/chat/getchat

peerid	string	the id of the peer
listingid	int32	the id of the listing

This API call retrieves all messages from a chat with a given peer id and listing id.

POST /api/chat/getstates

This API call retrieves the states of all open chats with other peers.

GET /api/chat/poll

timeout *(optional)*	int32	the time to poll for
sinceTime *(optional)*	int64	the time since to retrieve events from
category	string	the category to listen for events

This API call starts a long polling request and returns data when an event in the given category has occurred.

5.2.5 Managers

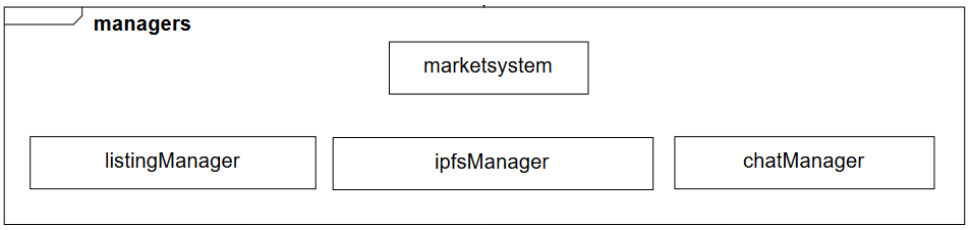

Figure 8: UML diagram of the manager classes.

The Managers package (see Figure 8) manages multiple different responsibilities. The main classes in the managers currently are listingManager, ipfsManager and chatmanager.

Listing manager

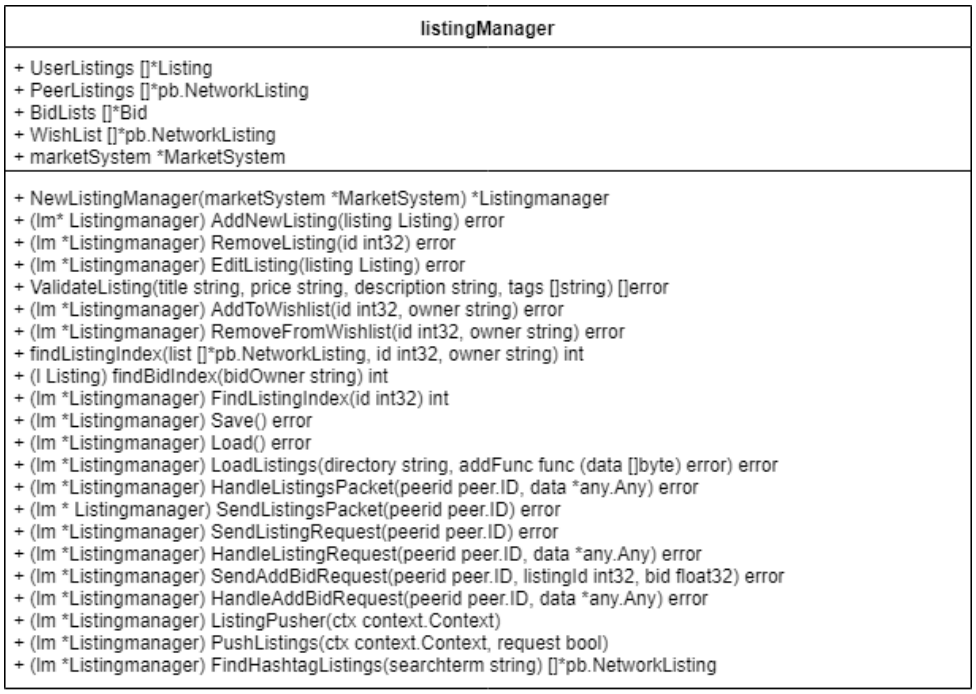

Figure 9: UML diagram of the class listingManager.

The listingManager (see Figure 9) handles all of the requests that have to do with market listings. A newly self-created listing can simply be added into the network by calling the *AddNewListing* method. This method is used to add your own listing. It takes a new listing, calculates the hash and adds a signature to the listing and saves it locally.
RemoveListing enables deletion of your own listings.
EditListing enables the editing of your own listings. When a detail in a listing changes, the listing hash will be recalculated and the signature will be updated as well. The listing version is incremented by one. This way other nodes know that they got the listing with the newest version number and that it is indeed coming from the owner.
AddToWishlist and *RemoveFromWishlist* are called when a user wants to add and remove a listing from his wish list. This just saves the listings locally.
Next, there are two versions of the function *FindListingIndex*. The first method is used to find the index of listings that were not created by this node. This method is often used when a new version of a listing arrives and we need to find the location of the old version to do the comparison and replacement. The second method is used to find the position of the node's own listing. When there is a bid coming to this node, it can look up it's own listing given the id.
The *Save*, *Load* and *LoadListings* methods are used to save and load listing information from the user's device.
The *SendListingsPacket* function sends all listings contained by a node as a packet to another node, given the node id. The *HandleListingsPacket* function handles receiving this listing packet on the other side. The former function is called when there is a need to send all listings one has to a particular node, for example when the method *HandleListingRequest* is called. *HandleListingRequest* is called only when this node has received a request for a listing (initiated by calling *SendListingsRequest* on the other side) from another node.
SendAddBidRequest is a function to be called when you want to send a bid to a particular listing owner, indicating you are interested and did a bid on the listing. The method *HandleAddBidRequest* is used to handle all the add bid requests sent from other nodes.
ListPusher and *PushListings* are used to push all the listings a node knows to all the nodes it knows. This is periodically done to ensure that all the listings are available for every node.

Chat manager

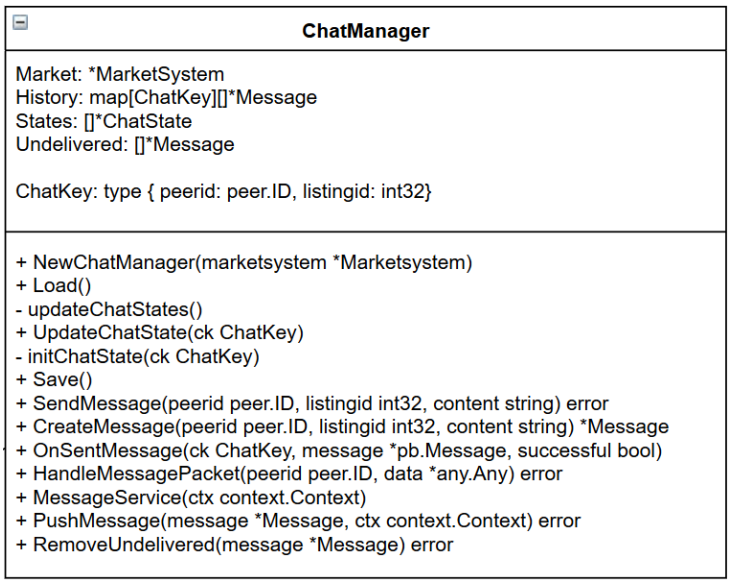

Figure 10: UML diagram of the class ChatManager.

The *ChatManager* class handles all chat history and states for the user. Chats are based on a peer id and a listing id. That means for every different listing of the same peer you have a different chat. And for every different peer you have a different chat on the same listing id. The history of all chats is therefore stored in a map by peer id and listing id combined.
To create a ChatManager object, the function *NewChatManager* is called. This method correctly initializes the chatmanager by instantiating the correct empty maps and Marketsystem object. This function also calls the *Load* function to load the chat history from the disk. After the chat history is loaded from the disk, the *updateChatStates* function is called. This function goes through the history and calls the *UpdateChatState* function. This function first makes sure the chatstate is correctly initialized by calling the *initChatState* function. After that, *UpdateChatState* will compute some statistics. An example of these statistics is whether there are unread messages, but a lot more can be computed from the history. These chat states are used to give an easy overview of the state of a chat without looping over the messages every time the user requests the state. After this, the ChatManager is done initializing.
After initialization is done, the ChatManager can be used. The *SendMessage* function tries to send a message to another peer. To do this, it first creates a message object with the *CreateMessage* function. It then calls a function in the peer package to actually send the packet to another peer. *OnSentMessage* is called after that to handle what happens after sending the message (e.g. add to undelivered if unsuccessful delivery).
When receiving a Message packet from another peer, the data is received in the *HandleMessagePacket* function. This function parses the received data to a Message object. It adds it to the history and then updates the states.
When a message was not delivered successfully, the message gets added to an undelivered list. The *MessageService* function handles the pushing of undelivered messages through the use of the *PushMessage* function. It attempts to push undelivered messages every 90 seconds. This limit is there because libp2p prevents us from trying to connect to the same peer more often. When the message is now successfully delivered, the message gets removed from the undelivered list by calling the *RemoveUndelivered* function.
When the application is shutdown, the *Save* function is called to save the chat history to the disk. This method saves the chat history in json format.

IPFS manager

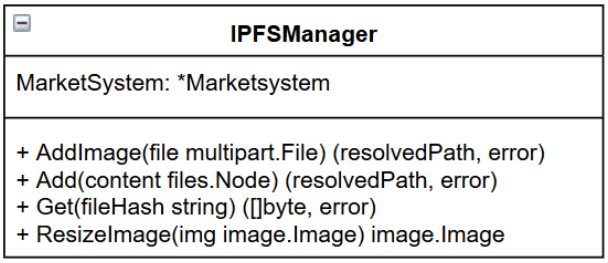

Figure 11: UML diagram of the class IPFSManager.

The IPFSManager (see Figure 11) is a class that handles all the interactions with the running IPFS node. The *Add* function adds a file to your local IPFS node. It returns the hash of the contents, which is in turn the location of the stored content. The *AddImage* function adds an image to your IPFS node. It tries to resize it to an acceptable size by calling the *ResizeImage* function. After adding a file to IPFS or when receiving an hash from another user, the *Get* method tries to retrieve the requested content.

MarketSystem

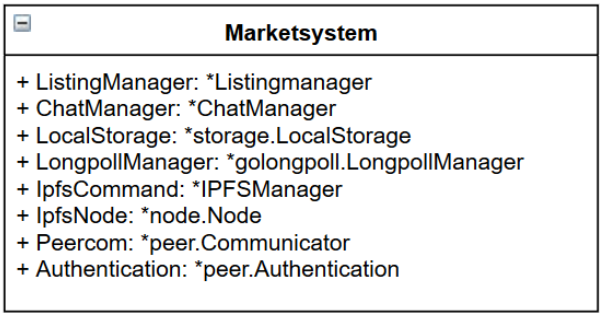

Figure 12: UML diagram of the class marketsystem.

The marketsystem (see Figure 12) is a simple class with pointers as variables, used as an object for references. This object is initiated once in the beginning and being passed to the constructors of all managers. This way, every managers have access to the same instance of all the managers and the other classes of the program.

5.2.6 Storage of data

The storage package has a single class that handles reading from and writing to files on the user's hard drive. It provides a simple way of accessing files within the directory that we use to save the data. The data is currently stored in the user's home directory, within a folder called marketpalace.

5.3 Web interface

As stated before we do not focus on the front-end of our application. However, we decided to create a web interface for demo and testing purposes. For this reason we will not provide screenshots. We have in total 6 different pages:

- *Sign in*: Users can use their password to enter the marketplace. If they have not registered before, they can click on *Sign up* or *Import keys*. When entering the correct password the user is redirected to *My listings* page.
- *Sign up*: Users are instructed to install the IRMA app and to disclose their attributes. When they scan the QR code, we hash this attribute and check whether this user already entered the marketplace before. If so, they are requested surf to the *Sign in* or *Import keys* page. Otherwise the generated keys are displayed and the user is instructed to save these keys offline. After this, he is redirected to the *Sign in* page.
- *Import keys*: When a user has a new device and wants to save its keys on this device, he or she can import the keys. They have to paste the keys into the two fields. When the keys are incorrect, an error message is shown. Otherwise he redirected to the *Sign in* page.
- *My listings*: This page displays all the listings that this user created. The user can add, remove or edit a listing. The user also sees the bids on this item. Every listing border in MarketPalace is either orange or blue. Orange indicates that this item is offered and blue means it is wanted.
- *Other listings*: This page displays all the listings that other users created. The user can do the following things on a listing: make a bid, add to favorites and open a chat with the listing owner. Next to that there is a search bar that matches the search term on the hashtags of listings.
- *Favorites*: User can remove the bid from his favorites and bid on favorite items he has not bid on. The user will also get an overview of all the listings he placed a bid on.

Besides these pages we have an overlay for the chat on the *favorites*, *other listings* and *my listings* pages. Users can open a chat by clicking on a button on a listing. A chat window appears at the side of the screen where the user can type their message and view their chat history.

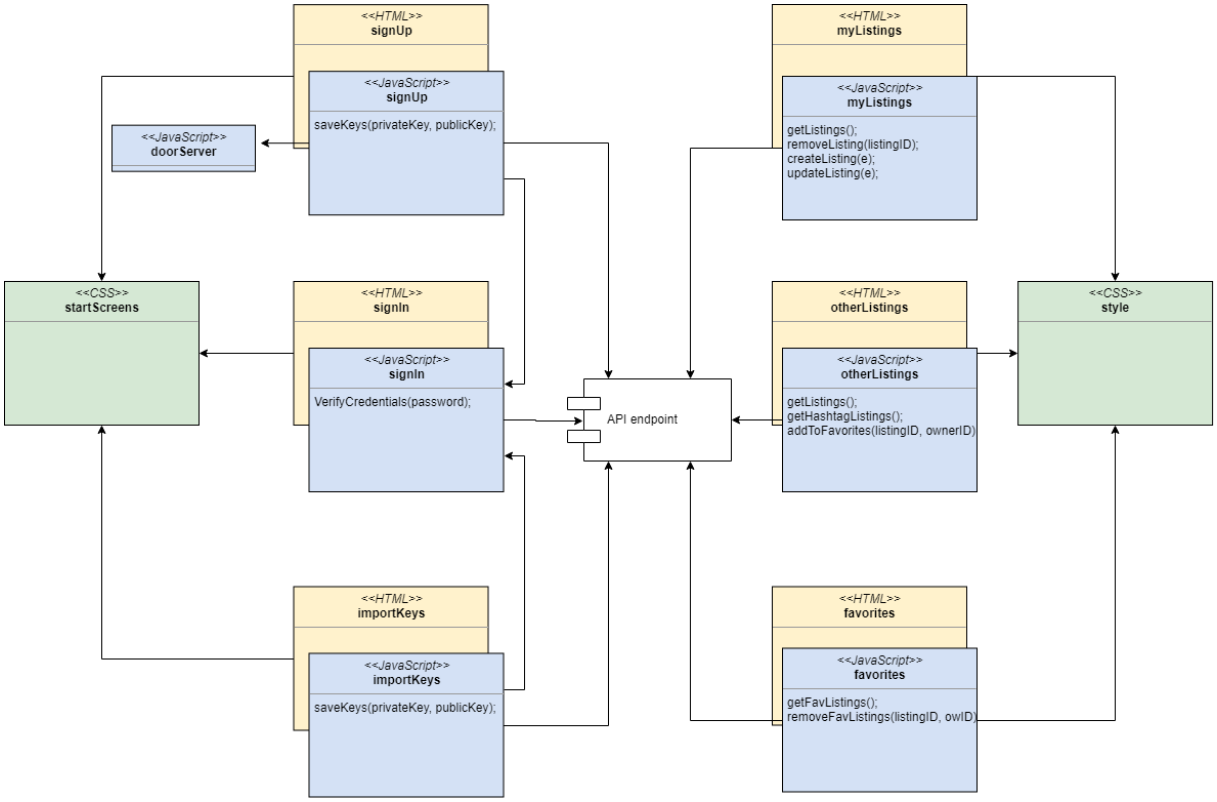

Figure 13: Overview of the web interface.

Figure 13 shows how the source code is structured. Note that doorserver is the back-end of the door process as explained in section 5.1. We did not find it necessary to add all function in the diagram. Most of them were handling the front-end. Only functions that make API calls are listed. The chat implementation is not shown in this diagram because it exists on at least three of the pages as a separate module.

5.4 Security

Since we are building a fraud-resistant marketplace, we need to address the security of our system. In various parts of the system, different security decisions were made, which will be laid out in this section.

5.4.1 IRMA server options

How secure the IRMA server is, depends on the configuration and where the server is hosted. When the IRMA server sends attributes over the internet, some form of security is needed to ensure that the private information of users is not intercepted by malicious persons. For this reason we decided to handle the disclosure session results on a server that is hosted on the same machine as the IRMA server. This functionality is implemented with the door server, as explained in section 5.1. IRMA allows for the usage of an API token to only have authorized requestors, but also signing session requests with a key is possible [5]. However, because the session requestor is a trusted entity, we chose *none* as *authmethod*. The session results can be handled securely on the backend.

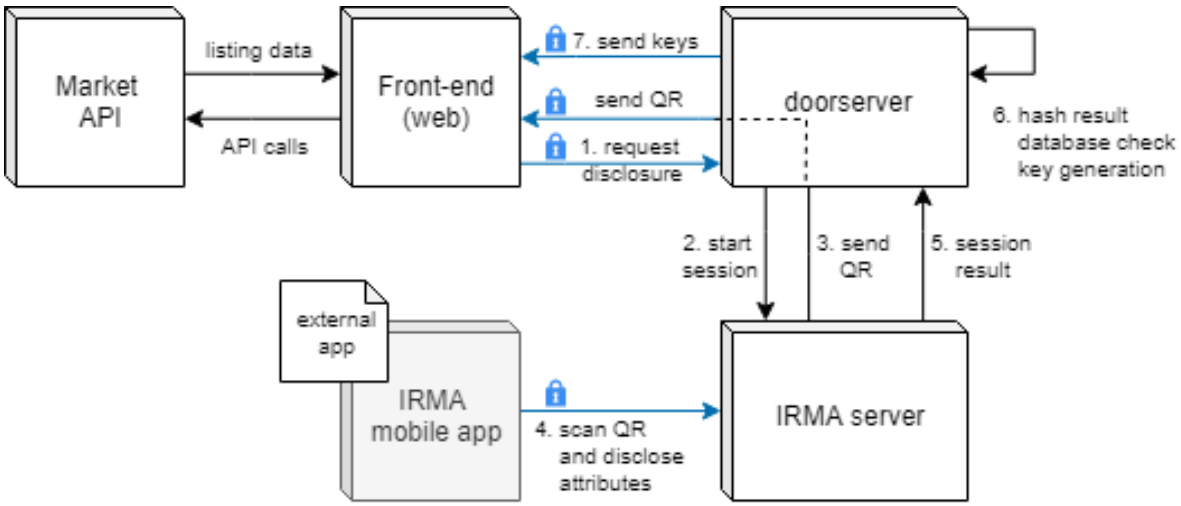

Figure 14: Schematical representation of the different modules of MarketPalace and the protected connections in the IRMA disclosure process (blue).

As can be seen in Figure 14, the session request is initiated on the door server in step 2 after the user has clicked a button in step 1. The QR code created in step 3 is sent to our front end where the user can scan it. In step 4, the personal attributes are sent from the user's IRMA application to our IRMA server. This rises a new security issue. IRMA has implemented a TLS-connection between the server and mobile application which can be enabled by adding a key and certificate to the configuration of the IRMA server. By having a TLS-connection between the frond-end and door server, and the IRMA mobile application and IRMA server; there is a secure disclosure of personal attributes in the system.
The API calls to the market system and the listings that are returned are not encrypted, as the listing data is not as privacy-sensitive as keys and personal attributes. In the future however, listings could be encrypted as well with the public keys of users to realize a complete secure application.

5.4.2 Hash attributes

Hashing the BSN is an important part of our system security-wise, as we do not want to store any personally identifiable information of users. There are different hash functions that we can use to hash the BSN that we retrieved from IRMA. We considered the following functions:

- MD4
- MD5
- SHA-1
- SHA-256
- SHA-3
- TIGER

SHA-1 and MD5 (and its predecessor MD4) were popular hashing algorithms, but they are shown to be vulnerable for collision attacks [6]. SHA-256 has a better security level than former hash functions against collision attacks [7] and has great potential for our application. Tiger is newer and is a bit longer. Both SHA-256 and Tiger are strong hashing functions, which means they take longer to compute but are harder to decipher. SHA-3 is new but is not as widely used as SHA-256. SHA-3 is meant as a backup of SHA-2 [8]. Therefore we decided to use SHA-256, which is also what our coach, a security expert, recommended.

5.4.3 Key encryption

We distinguish two main approaches to encryption: asymmetric-key encryption and symmetric-key encryption. We chose to use asymmetric-key encryption because we can not ensure a secure channel to exchange keys. The most used asymmetric-key encryption algorithm is RSA. It relies on the difficulty of factorizing large numbers [9]. We are aware of the fact that quantum computers can do factorization easily. The other drawback of RSA is the fact that the keys need to be long in comparison with using symmetric-key encryption. Nevertheless, with current technologies the RSA algorithm will do for our system.

5.4.4 Signed listings

The other part of RSA that was used is signing data. As explained in section 5.2.3, listings are signed by users before they are sent over the network. The listings are first hashed to speed up the signing process and avoid collisions. Afterwards they are signed with the private key of the user. The receiver can verify that the user that claims to be the owner of the listing is actually the one that sent the packet [9].

6 Evaluation

6.1 Final product

In this section we are going to evaluate the product we implemented. First we will look back at the requirements and discuss what has been implemented and what not. Then we will look at the code quality. This will be followed up by evaluating to which extend we tested our code. After evaluating this aspect we move on and evaluate the performance of our network. Finally we list the limitations that our final product has.

6.1.1 Requirements

In this section, we evaluate to which extent the requirements from section 2.3 are satisfied. All must haves are implemented, as well as the should haves. Additionally, one could have was implemented.
The only personal data of users that is stored is the hash of the attributes. This is used to uniquely identify users. We used as little personal data as possible and we have paid extra attention to preserving the privacy of users, by making it nearly impossible for malicious persons to retrieve this personal data (usage of hashing).
The application has a simple web interface that demonstrates how the system works.
The creation of multiple accounts for a person is made impossible, by using the third-party identity platform IRMA for registration. If users have signed up once, they can enter the marketplace with their key-pair that was generated during registration.
The aim was to have an affordable and scalable system. In our research (Appendix A), we came to the conclusion that scaling was the most affordable in a P2P system, as no extra server capacity is needed. A peer can join the network at any time with their own resources. On a small scale, the P2P network indeed offers affordable scalability. However, when the system grows large, it would run into some issues. MarketPalace is designed in such a way that every peer sends and receives all listings over the network. On a large scale this is not maintainable and this would lower the performance.
The P2P network was not only chosen for affordable scalability, but also to keep users in control of their own data. Market listings are stored on user's devices and not on some central server. The listings contain the necessary details which can be updated. Users can add listings for offered and wanted items and are able to retrieve these from other peers. The listings can also be removed.
Next to these main requirements, the search functionality was also implemented. The system matches the search terms with the tags of the listings.
Users can also indicate their interest for an item by placing a bid. The listing owner can see all bids belonging to that listing. The user can remove the bid if needed.
Finally, the chat channel was implemented so users can conduct business within the system.

6.1.2 Code quality

To maintain high code quality, several tools were used throughout the project. Next to that, SIG provided us with valuable feedback that we could use to improve the code quality even more. These different aspects of code quality will be discussed in this section.

Continuous Integration
Continuous Integration (CI) was set up early in the process, to make sure some standard of code quality was maintained. GitLab offers an easy way to set up CI, with the possibility of running parallel jobs. We expected some issues in running a CI for a multi-language project, but in GitLab CI, different Docker images can be used for each job. The JavaScript and Go code analysis tools were run in parallel, as well as the test jobs for both languages. Lastly, a compile job was run for the Go part of the system.

Code analysis tools
The code analysis tool used for the JavaScript part was JSHint[6]. JSHint finds simple syntax errors and typo's, but also bugs. It has helped us in keeping the code clean and consistent. Warnings of wrong scoping were also given, resulting in us better understanding the workings of the JavaScript language. Since we were fairly new to writing in JavaScript, we used the default ruleset as this filters the most common mistakes.
A similar tool was used for the Go part of our code; GolangCI-Lint[7]. Various linting tools exist for Go, but this one seemed widely used and fitted our needs. We needed an analysis tool that is easy to integrate in the CI. Again the default ruleset was used.

SIG
In week 6, we sent our code to Software Improvement Group[8] (SIG) to be evaluated. We were satisfied to find that our code resulted in a score of 3.7 stars in their maintainability model. "Unit Size" and "Unit Interfacing" were seen as potential points of improvement due to their lower partial scores.
Unit Size refers to long functions that contain too much functionality. A lot of comments within these functions mostly indicate that the functions have too much responsibilities. The advice is to split up these functions into smaller ones, each one having a clear and specific functional scope. One of the example functions that had an above average length was NewNode() in node.go. Among others, we placed the initialization of the repository in a new function. We managed to go from 68 lines of code to 42 in NewNode().
Unit Interfacing looks at how much functions have an above average amount of parameters. Too much parameters normally indicate a lack of abstraction. This can be solved by replacing a group of parameters with a new object. An example in our project was AddListing in ListingManager.go, which we have improved by replacing the parameters with a Listing object. SIG suggested splitting the parameters into multiple new objects, but we think this does not make sense here, as the parameters of Listing can not be abstracted further.
Lastly, the presence of tests looked promising, but the amount of test code lagged the amount of production code a bit. Testing the web code was hard and sometimes unnecessary in our opinion, as we will elaborate upon in section 6.1.3. Therefore we focused on improving the tests for our Go code as a result from the feedback we received from SIG. The improvement in coverage for the different packages in our Go code can be seen in Figure 15.

```
/api       [no test files]
/classes   0.955s  coverage: 13.4% of statements
/ipfs      [no test files]
/node      1.568s  coverage: 74.2% of statements
/pb        [no test files]
/peer      [no test files]
/storage   0.290s  coverage: 73.1% of statements
```

(a) Coverage before feedback

```
/api       6.209s  coverage: 50.2% of statements
/managers  0.964s  coverage: 51.4% of statements
/node      2.134s  coverage: 78.7% of statements
/pb        [no test files]
/peer      0.540s  coverage: 29.2% of statements
/storage   1.347s  coverage: 73.1% of statements
```

(b) Coverage after feedback

Figure 15: Test coverage of the different packages of the Go part of the system before and after receiving SIG feedback. Note: code restructuring caused some packages to be removed.

[6] https://jshint.com/
[7] https://github.com/golangci/golangci-lint
[8] https://www.softwareimprovementgroup.com/

6.1.3 Testing

To be able to say something about how well the system works, testing is needed. Next to that, it also gives valuable feedback to where the system fails when new functionality breaks the existing code. In this section, we clarify how we tested our code and the challenges we encountered.

Testing web JavaScript
As explained in section 5, the web functionality of MarketPalace can be divided into the 'door server', the 'IRMA server', and the client that contains the HTML pages to navigate through the system. We decided to not have our focus on testing the front-end, as the front-end is not our main goal in general for this project. That's why the HTML pages are not tested.
Also, we do not test the IRMA server as this is a dependency that is already well-tested.
What remains is the door server, which interacts with the IRMA server and the client-side. The handling of messages between the client-side and the door server is not tested, as this is mainly testing if the server library works instead of our own code. Next to that, it is hard to mock server-socket behavior, so we decided to use our time and effort elsewhere.
Some behavior that was tested was the signing of generated public keys of users. This functionality also depends on a library, but to make sure this vital part of our program works and that we create valid keys, we decided to test this part of the door server. Next to that, some minor key parsing functionality was tested.
To verify the database queries such as insertion and deletion work as intended, we made use of error messages instead of tests. For example, one of the functions is connectToDatabase(), which either establishes the connection or returns an error message. There is no need for an extra testing layer on top of this.

Testing market Go
Most of the marketsystem was easily testable. A problem arises however when there are functions that require multiple peers connecting to each other. We could not get multiple authenticated market nodes connected to each other in the tests. This made it impossible for us to test these features of our application. This is therefore the reason why we only got around 30% coverage for the peer package. Most of the code in that package however, is already tested at a basic level by the libp2p and IPFS team. This is also the reason why the coverage in the managers and API package is only around 50%, as they contain code that points to the peer package that tries to connect to other peers.
A package that we did not test is the pb package. The pb package contains proto files which are not testable. These proto files are compiled to go files. Since these files are automatically generated we found no need to test them.

6.1.4 Performance

In this section, we will take a look at various actions in the network and their performance. The conclusion will be formed upon reviewing the results from experimenting with these actions. We considered answering the following two questions:

- How long does it takes to add a listing?
- How long does it takes to retrieve a listing?

The time required for performing these operations depends on the system. This is affected by Distributed Object Location and routing as well as content caching, replication and migration [10].
Before the questions can be answered, it is essential to understand what exactly is an experiment and how is it conducted. The time to add a listings is the time between the moment of a new listing added until it is received by an arbitrary peer in the network. The time to retrieve a listing is the time between the moment of a peer asking for listings from another peer and the moment of receiving it.
The first and second questions have an overlap. The overlap is the time it takes for a listing to reach a receiver. This is why we decided to merge these two questions in one: *How long does it takes to add and receive a listing?* In order to answer this question, we have set up an experiment where we will

measure the time between these two moments: The moment node 1 initiated adding a listing into the network and the moment node 2 has received the same listing. To do so we have set up two nodes and manually carried out the steps and measured the time automatically. We have run this test 100 times. However, first we need to understand the factors that can potentially influence the measurement time. Note that, when a new listing is created it does not necessarily mean it will be pushed into the network right away. There is a timer within every node regulating the frequency of pushing listings (including the newly created listing) into the network. We call the sender node 'node 1' and the receiver node 'node 2'. We have concluded that there are theoretically three factors in adding and receiving a listing on the P2P network:

1. The time remaining on the timer of node 1 until the listings are pushed into the network.
2. The cumulative time remaining on the timers of other nodes on the route from node 1 to node 2 before they push the listings to the next node excluding the time traveling through the physical network medium.
3. The total time it takes for the network packets containing the listing to travel through the physical network medium.

The first factor depends on the time remaining on the timer within node 1. Only when this timer ends, the new listing is pushed to the network where other nodes can retrieve it. This timer, in this experiment, is set at an interval of 90 seconds. If we were to do many experiments with only this factor accounted, the distribution would be uniform since every amount of time of 1, 2, ..., 90 seconds has the same probability of happening. This is a big factor in our experimentation since we will be sending from node 1, thus always making use of the timer.
The second factor depends on the cumulative time remaining in the timers of the nodes that lay on the propagation route from node 1 to node 2. This time excludes the time it takes for the signal to travel over the physical medium. For example, the second factor for a network with a route through two nodes before it reaches node 2, with the remaining timer of the first node being 20 seconds and the remaining timer of the second node being 15 seconds, would be 35 seconds. However, this second factor is excluded from our experiment since we only send directly from node 1 to node 2. In other words, node 2 is in the list of the 20 closest peers of node 1.
The third factor depends on the cumulative time that it takes for a signal to travel on the physical medium on the propagation route from node 1 to node 2. Nowadays, the speed of signals traveling through the physical medium is extremely fast and can be considered a negligible factor in our experimentation.
The 100 measurements are put below in a table together with the median and sample standard deviation (See Figure 16).

Experiment time				
	52	12	10	3
	11	23	36	4
	63	45	8	12
	64	62	9	35
	1	28	10	32
	37	13	32	42
	21	1	77	14
	34	3	81	1
	35	89	60	1
	53	67	41	32
	1	35	82	25
	7	75	33	46
	55	23	62	67
	60	19	48	43
	23	23	36	29
	55	1	16	18
	66	34	32	9
	9	88	78	20
	47	3	17	11
	76	19	32	45
	84	71	49	83
	16	58	7	2
	12	60	88	89
	70	5	22	7
	70	50	25	80

Mean	36,7
Median	32,5
Mode	1
Standard Dev	26,64866817

Figure 16: The experiment measurements, their median and standard deviation.

Assuming a normal distribution, the mean at 36,7 means that if we were to add a listing to the market and it is received at another node in the network, most of the time it will take an average of 36,7 seconds. The standard deviation of 26,6 shows that the chance of a listing added and received in the network cost more than 89,9 seconds is less than 5 percent. This number could useful in the sense that, if after 89.9 seconds there is no acknowledgement from other nodes, the add listing operation can be considered lost and can be recreated. This might potentially save waiting time for acknowledgement of new listings in case we use acknowledgement in the future.
However, putting the measurements into a histogram shows us that the distribution is not normal (See Figure 17).

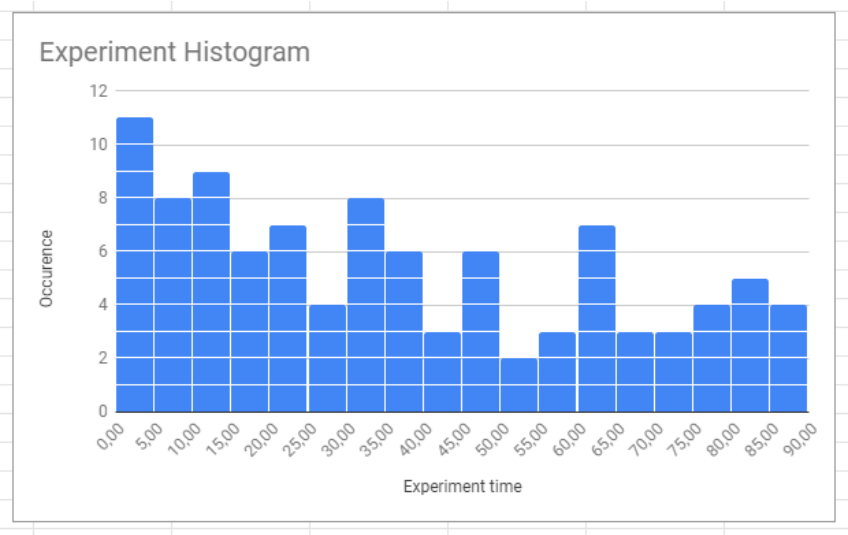

Figure 17: The experiment measurements in histogram.

Every blue block in Figure 17 represents an experiment result. It can be seen here that it could have the form of a Pareto distribution. The Pareto distribution is a skewed distribution with heavy "decaying" tails. Note that this is an extremely small amount of experimentation result, therefor the above mentioned

observations can be highly inaccurate.
Since the experiment is hypothesized to have the biggest influence to be factor 1, the expected histogram should have a uniform distribution (See factor 1 for explanation). However as shown in the previous paragraph, the distribution is skewed to the left. This could have been caused by an extra time overhead of approximately 5 to 15 seconds when carrying out the experiments.

6.1.5 Limitations

Even through we strive to build a perfect marketplace, due to time and resource constrains there are several limitations on our software. Most of these limitations are from the marketplace aspect (See section 5.2).

- When a user imports his or her key on a previously used device which still contains the listings from the previous user (if not deleted from the device manually). This is caused when a user has successfully authenticated, the program just fetch the default file on local storage for the owner listings. This issue can be fixed by associating the listing stored locally to the key which is currently in use
- If the device is lost or the owner goes offline permanently, his or her listings will remain forever in the network due to not being deleted by the owner. This issue can be fixed by integrating a timer or an expiration time stamp on the listings.
- When user switch device, his or her listing are not transferred to the new device. This limitation can be fixed by having an import mechanism via Bluetooth or via cloud servers. However this transfer module is not in the scope of this project and can be implemented later on.
- Since a node is designed to push all listings it has to all the other nodes it knows every 90 seconds, there will be a bandwidth issue once the data involved becomes bigger. This can be fixed by for example redesign the node in such a way that each node checks their listings completeness and request for changes periodically. The other alternative is to store own listings and replications of others' listings and just send out requests for specific listings when needed.
- If the user lose his or her key and signature given after the registration process, he/she can never the marketplace again. It is essential that the key and signature are securely stored. This can be fixed by linking the registration information with the key and signature. However, this reduces the user's anonymity if our methodology of hashing registration information is compromised.
- If a owner of a listing is offline, then the users which are interested in that listing cannot contact or send a bid to the owner. This limitation can be overcome by implementing a server which temporary holds the messages and make the software run in the device in idle mode, ready for push notifications.
- Currently, the platform makes use of IRMA solely. This leads to dependency of IRMA. The registration process makes use of the IRMA identity platform which does not have the ability to scan QR code from image file. This means that if the registration process were to happen on a mobile device, there is no way to scan the the QR code of the registration page with the same mobile device. This limitation can be fixed by adding alternative verification options.IRMA is mostly used and supported in The Netherlands. This means international users cannot register for an account using IRMA. This limitation can be overcome by adding more identity verification platforms or verification options during the registration process.
- The door server makes use of a config file that holds passwords and keys for running the servers and accesssing the database. Normally you would not push this on the repository. The config file should only be visible to the machine that runs the door server and IRMA server. For simplicity and in the scope of this project we don't host the door- and IRMA server

- Currently we are the one who generates the key and signature for the users. This means that if our server which generates the key and signature is compromised, then there is a possibility that the key can reach non-intented recipients. It is best that the key pair is generated by the user, or by a process in the user's device and then give the public key to a process in Cirsecon to sign. This is the best way since the private key has never traveled any distance online.

6.2 Process

In this section we are going to evaluate the process. It is not only important to learn from designing and implementing a product, we should also learn from the process.

6.2.1 Organisation

Roles and responsibilities
When we found that we had to answer two sub questions, we decided to split the team. Two member would start focusing on the door process and the other two on the P2P network. After sprint 5, the basic functionalities of the door process were roughly finished. This was the moment these two team member joined in on working on the network part.
During the design and implementation phase we found out that we needed a new role. This person, *Chameleon*, was assigned less hours in the sprint backlog. In return he had the responsibility to jump in when other tasks required help. Justin took this role because he had the most experience working with the used tools. Other than that, all members were happy with the team work. Each member took his role seriously and was responsible.

Tools
Installing the dependencies of Go in the CI was most of the times not an issue. Sometimes it caused build failure. The next day it would work again. This unpredictable behavior was annoying but did not cause damage. There were no complains about the other tools we used.

6.2.2 Methodology

The different methods we used during this project have proved to help us in realising our product during design and implementation. We have used CRC cards to design the product and used the Scrum-methodology to guide our implementation process.

CRC cards
As the name suggests, Class-Responsibility-Collaboration cards are a tool to find the responsibilities of potential classes. In previous projects, the team found the usage of CRC cards very beneficial to the design process. However, these were mostly object-oriented projects, where code is divided in classes. Since our system is not completely object-oriented, we did not use the results from the CRC cards as classes in our implementation, but we did get a lot out of the card creation process. With the team we discussed the requirements and were able to get a clear overview of what elements were needed in the system.

Scrum
Creating the weekly backlog and retrospective documents were crucial in our successful team process. We had manageable weekly plannings that improved every week, by using the points of improvement we found in the retrospective of the previous week. Splitting the responsibility and execution of tasks helped team members in being involved in other's work and focusing on one part of the process.
The GitLab board was very convenient for managing our tickets created in the backlog and helped us in reaching the milestones at the end of every sprint. in this way revealed certain problems and served as a tool to improve each sprint.

6.2.3 Product Planning

Our initial roadmap can be found in Table 1. Working on a 1200+ hours project can be very stressful. It is not only important to learn from designing and implementing a product, but also from the process. In this chapter we will first look back at the team work. Then we will evaluate whether the chosen methodology was useful. We will close this chapter by evaluating our initial product planning. We will discuss every milestone separately.

Milestone 1
Our first goal was to set up the project and conceptualize it. This includes making CRC cards and diagrams. We did not finish all the diagrams before the deadline. This was too ambitious. Next to that we wanted to get familiar with IRMA and their code. We experimented with it and we understood by the end of the week how we could integrate it in our application. The third goal was to create a connection between Libp2p nodes. This was a lot of work. We were able to establish connection but not with the desired effects.

Milestone 2
By the end of sprint 4 we wanted to have a basic web interface with a sign up page. This was goal was achieved. We also wanted to have set up a database. Amazon RDS was set up without the functionality of adding a hashed BSN to it. The third goal was to be able to add and remove market listings. This goal was shifted to a newer milestone because we had to optimize the connection establishment in the network and the diagrams.

Milestone 3
During this sprint one crucial team member had to leave the country for family matters. This slowed down the process. It was our goal to wrap up the very basic door process. This means being able to retrieve attributes and add the hashes to the database if the user is new. Also, we wanted to generate keys when user was given green light to enter the marketplace. This was all done successfully by the end of the sprint. The final goal was to be able to retrieve market listings from other peers. This workload of this task was underestimated. We managed to save market listings to the disk and were able to send it over the network, but not via API calls.

Milestone 4
The team had agreed upon having a web interface for the complete marketplace. This means having different listing pages. By the end of the sprint we were able to have the following pages: Sign up, My listings and Other listings. We also wanted to connect the door process to the marketplace via the Sign up page. This required creating an API. This third goal was also achieved by the end of the sprint.

Milestone 5
Again, we had three goals for this milestone. Our first goal was to add details to market listing. This includes Title, Price, Description, Image and Type. This goal was already achieved in milestone 3. The second goal was to make it possible for users to bid on items. Because of other priorities and unfinished tasks we decided to shift this feature to another sprint. Our third goal was to be able to remove bids. Since this task could only be done after the previous goal it was also shifted. We implemented other features like finding items based on their hashtags.

Milestone 6
Our initial goal was to have a search function and be able to update a market listing by the end of sprint 8. However, we already had a search function in the previous sprint. We created a pop up to update a market listing. A lot of time was spent on writing the report and testing during this sprint.

Milestone 7
This was the final milestone where we could work on the application. We wanted to be able to indicate that an item is reserved. This was a very low priority task and therefore was obsolete. The next goal was to have a deal made indication. We preferred a chat channel over this so that users can actually get in touch with each other. The final goal was a could have. Namely, a very basic reputation system where users could give each other a rating. This is useless when you cannot ensure that there has been a handshake between the parties. In other words, rating each other without having a indication that a deal is made is useless. We decided to spend our time on finishing and refactoring. Next to that we worked on the report.

Milestone 8
This week was a buffer week. We would finish the report and the product if this was not already done. For the product we did not need this sprint. We used Monday and Tuesday to finish the report.

7 Discussion and future work

7.1 Ethical implications

Peer-to-peer networks can be very controversial. The main reason for this is sharing copyright material is illegal [11]. Content industry representatives such as Recording Industry Association of America (RIAA) have sued file-sharing platforms and it's users. We want to stress that we are not a file-sharing platform but a decentralized marketplace. These marketplaces can be controversial too. The US has sued a decentralized marketplace before, Silk Road. Silk Road operated anonymously on the 'Deep Web' and was taken down in 2013 for selling illegal products like drugs and weapons [12].
In the current version of MarketPalace it is possible for users to sell or buy illegal products. Since we have decentralized control, we will not be able to ban this person. In the future we could introduce blacklisting system. Users would be able to report listings as inappropriate or illegal. Cirsecon will review the listing and put the user's public key on a blacklist. This blacklist will be fetched by the users. Listings that correspond with a blacklisted public key will not be displayed. Another way of solving this is to have a decentralized ban system. Both approaches have some drawbacks. Having Cirsecon as the judge by introducing a centralized blacklist will be time consuming and can be costly. This goes also against the idea of decentralized power. The second approach solves the latter, but is not fault tolerant. How do we reach consensus? If the network consist of a significant amount of users who sell or buy illegal products they can take over the network. Also, users might not want spend their time reviewing listings.
We do not collect personally identifiable information. In this version, we request attributes from a Self-Sovereign Identity solution which is privacy-friendly. Users have insight in what is requested and give explicit consent. The decentralized infrastructure ensures that Cirsecon cannot build profiles and sell it to make profit. In essence, MarketPalace strives to be ethical with the focus on privacy and user consent. However, we are aware of the fact that we facilitate immoral behavior which leaves points of improvement.

7.2 Recommendations

MarketPalace is in its current state far from finished. The limitations in section 6.1.5 point towards topics that has to addressed in the future. Next to these points we have the following list of recommendations:

- MarketPalace is not user-friendly. The front-end was neglected and deserves attention in the future.
- When users have successfully authenticated they are given a key-pair. When a user loses this pair, they will not be able to enter the marketplace on a new device. Future research should look into this problem and find a solution that is in line with the philosophy of MarketPalace.
- IRMA does not only collect BSN. They can also collect attributes like home address and bank account number. These attributes can be used to for example disclose the address of the buyer or seller and the bank account number of the seller. That way users can know for sure these attributes are verified.
- The marketplace network should be divided in many sub groups, defined by peers that are closer to each other geographically. This is to avoid having to push unrelated listings to peers that are located too far geographically. Thus saving a lot of bandwidths and network traffic.
- There should be a reputation system implemented to make use of the full effect of the door system. Currently the door prevents multiple or unverified accounts to be created. The reputation system implemented would have been ballot stuffing resistant. This also gives the users of the marketplace an idea on how trust worthy a seller or buyer is.

7.3 Challenges

This subsection is dedicated to all the struggles we have as a group during the development process. The first challenge to us is that the description of the project was not really clear to us so it took us an appointment with the client and a while to figure out the concepts and what exactly the client want. Even after we know what the client wanted, we did not know what to build. What kind of network do we have to build? What tools and where do we have to research into?
Only after we went to our coach we were able to get a pointer in the right direction. We were asked to take a look into P2P networks, which was not familiar to any of us. It is a relatively difficult subject and we have not had any lectures on it at the university. However it also intrigued us to do decide to research and explore possibilities in this unknown field.
The client asked for many thing at the same time, in particular a whole marketplace with a reputation system. We have done research on both topic and decides to only implement the marketplace without a reputation system for now. This is due to implementing both will be very difficult in the time span of 10 weeks.
We also have language restrictions. Initially we decided to use Python since it is easy and is well supported. However as we do the research we realized that the libraries that we need, in particular, the Libp2p library is only well implemented in Go. This means we will all have to learn a new language from scratch. This is doable, however for the door process, we needed to use JavaScript due to it's compatibility with user web interface.
This leads us to cross-language compatibility when it comes to creating and verifying signature. Since we create a signature at the door process which is written in JavaScript and verify it in the marketplace import process which is written in Go, we were making use of two different libraries. This resulted in the mismatch of implementation and we could not get the signature verified correctly. This was our first time working with signatures and keys so hard becomes difficult.
Apart from that, the process of research and development, in our retrospective, is surprisingly well done. There were no major restructure of the architecture. There were no major delays or compatibility issues.

8 Conclusion

Our goal was to create a Sybil-resistant and decentralized marketplace for the planned start up Cirsecon. In the past 11 weeks we have designed and implemented MarketPalace. MarketPalace is not end-to-end operational yet, however the foundation has been laid. All the *must haves* and *should haves* in section 2.3 are implemented. One of the *could haves*, the chat function, is integrated and operatable aswell.
In order to create MarketPalace two sub questions had to be answered. The first sub question was:
How do we prevent users from creating multiple accounts?
We can prevent users from creating multiple accounts by having a database containing uniquely identifiable information from every user that has given access to the marketplace. In order to preserve privacy we decided to use IRMA as an SSI solution and request the BSN. This BSN is hashed with a hash-function that takes into account collisions and pre-image attacks. Every time a new user signs up, we hash the attribute we received from IRMA and check whether the database contains it. If so, the user is not given access. Otherwise, the user has proofed that he or she has not entered the market system before.
How do we build marketplace network which preserve user privacy and is cost-effective?
In order to preserve user privacy and enable (pseudo-)anonymity, we have chosen to implement a peer-to-peer network with Libp2p. Such a decentralized network is cost-effective when it comes to setting up, maintaining and scaling. We made sure that by using signatures, only users that have registered with Cirsecon can make use of the marketplace. As for the anonymity aspect, there is no link between the identity verification process and the issuance of the credentials. This makes sure that there is no way to trace back the original identity of a user in the marketplace.
In the future a reputation system can be build on top of MarketPalace. Also the P2P network can be optimized by dividing the network more geographically. This will save a lot of bandwidth and network traffic. Finally, IRMA is not only useful to prevent users from creating multiple accounts, it can also be further integrated within the marketplace. Buyers and sellers can disclose addresses and bank account numbers by using the IRMA app.

References

[1] D. van Bokkem, R. Hageman, G. Koning, L. Nguyen, and N. Zarin, "Self-sovereign identity solutions: The necessity of blockchain technology," *CoRR*, vol. abs/1904.12816, 2019. [Online]. Available: http://arxiv.org/abs/1904.12816

[2] B. Neil Levine, C. Shields, and N. Boris Margolin, "A survey of solutions to the sybil attack," *Technical Report of Univ of Massachussets Amherst*, vol. 2006–052, 11 2005.

[3] M. S. Olivier, "Database privacy: Balancing confidentiality, integrity and availability," *SIGKDD Explor. Newsl.*, vol. 4, no. 2, pp. 20–27, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/772862.772866

[4] A. W. Services, "White paper: Amazon web services: Overview of security processes," Tech. Rep., May 2017. [Online]. Available: https://d0.awsstatic.com/whitepapers/aws-security-whitepaper.pdf

[5] "irma server," Privacy by Design. [Online]. Available: https://irma.app/docs/irma-server/

[6] K. Bhargavan and G. Leurent, "Transcript collision attacks: Breaking authentication in tls, ike, and ssh," 01 2016.

[7] H. Gilbert and H. Handschuh, "Security analysis of sha-256 and sisters," in *Selected Areas in Cryptography*, M. Matsui and R. J. Zuccherato, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 175–193.

[8] C. Boutin. (2015) Nist releases sha-3 cryptographic hash standard. [Online]. Available: https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard

[9] E. Milanov, "The rsa algorithm," June 2009.

[10] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 335–371, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1041680.1041681

[11] M. Cenite, M. W. Wang, C. Peiwen, and G. S. Chan, "More than just free content: Motivations of peer-to-peer file sharers," *Journal of Communication Inquiry*, vol. 33, no. 3, pp. 206–221, 2009. [Online]. Available: https://doi.org/10.1177/0196859909333697

[12] M. C. V. Hout and T. Bingham, "'surfing the silk road': A study of users' experiences," *International Journal of Drug Policy*, vol. 24, no. 6, pp. 524 – 529, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0955395913001369

A Research paper

A Fraud-resistant Online Marketplace With a Reliable Reputation System

Dirk van Bokkem*, Tat Luat Nguyen†, Justin Segond von Banchet‡ and Naqib Zarin §
Computer Science and Engineering
Delft University of Technology
Email: *d.vanbokkem@student.tudelft.nl, †t.l.nguyen@student.tudelft.nl, ‡j.v.segondvonbanchet@student.tudelft.nl, §n.zarin@student.tudelft.nl

***Abstract*—Fraudulent behavior within online marketplaces and reputation systems is a prominent but unresolved problem. Many advancements have come by, as well as new forms of cheating the system. In this paper, multiple forms of fraud within these types of systems are looked into accompanied with available solutions. Reputation systems are often used in online marketplaces, as a means to mitigate fraudulent behavior. With it, however, new forms of deceit come into play. Furthermore, the underlying networks bring their own opportunities and threat, which require further investigation. With the combination of existing technologies such as global and local reputation, different forms of aggregation and using feedback similarity, this paper finds new ways of tackling the problem. The field of online marketplaces and reputation systems is explored and a suitable solution is proposed.**

***Index Terms*—trust, reputation, marketplace, fraud-resistant, sybil-resistant, anonymity, distributed systems, centralized, peer-to-peer**

I. Introduction

SINCE the arrival of the internet, new ways of connecting supply and demand have emerged in the form of online marketplaces. Market research company eMarketer has recently estimated that Amazon Marketplace is responsible for 31.3% of the total e-commerce sales in US in 2018 [1]. In a place where buyers and sellers come together, users have different motives and some form of regulation is needed to avoid malicious use of the system. Just as in the real world, online marketplaces are a breeding ground of fraudulent behavior. Sellers can lie about their products, buyers can withhold money.

Reputation systems have played a big role in online marketplaces. Research about the value of reputation in marketplaces has shown that buyers put a significant weight on the seller's reputation [2]. Switching off this reputation system decreased the revenue of the investigated marketplace by 11.1%. Different strategies have been developed to manipulate reputation systems with self-beneficial intent. This has a negative effect on online marketplaces where users are motivated to conduct business with trustworthy counter-parties for mutual benefit.

Before 2010 academic research has mainly focused on defense mechanisms against reputation systems. More recent researches have shown the potential of distributed reputation systems. The emergence of Bitcoin has played a role in this because of its underlying Distributed Ledger Technology (DLT). However, most of the applications that use distributed reputation system still suffer from Sybil-attacks because users can easily create multiple accounts (Sybils). Strong-identity mechanisms like Self-Sovereign Identity (SSI) can help hinder a user to create Sybils without having a need for a central authority [3].

In this paper we propose a solution that mainly focuses on combining a strong-identity mechanism with a reputation system. The main question we strive to answer is: *How do you build a fraud-resistant online marketplace with a reliable reputation system?*

To answer this question the following questions have to be answered first:

1) *What are the possible ways to address frauds in an online marketplace?*
2) *What are the possible methods to compute reputation of users in an online marketplace?*
3) *What are the possible system architectures for an online marketplace?*

First, we introduce the most common frauds that occur in online marketplaces and reputation systems in section II. Current solutions to these fraudulent behaviors are briefly mentioned as well. Next, different existing reputation systems are explored in section III, together with various concepts in creating a user's reputation. After this, the possible underlying network architectures that can facilitate in these different solutions are examined in section IV. In the before-mentioned sections, the field of study has been laid out. Many design decisions in these fields are dependent on one another. In section V, a proposal of a fraud-resistant online marketplace with a reliable reputation system is given, accompanied with a clarification of the made decisions. Various additional points of consideration regarding the system, such as GDPR-compliance and anonymity, are mentioned in section VI. Finally, the main question will be answered in section VII.

II. Fraudulent behavior

The increasing interest of honest users in online marketplaces go hand in hand with increasing interest of dishonest users. We will categorize the types of fraud that can take place. First we will discuss the most common network attacks in an online marketplace. Then we will discuss the type of fraud that dishonest users can commit in order to benefit from the reputation system. Finally we will discuss what we call the

transaction fraud where users make claims about (dis)honest payments and (dis)honest items.

A. Network fraud

- *Sybil attack*: Forging multiple digital identities within a system to have an advantage over other genuine identities [4].
- *Replay attack*: A malicious user intercepts a message and resends it. In blockchain technology it refers to reusing a pointer to a record of the other party to claim that he has performed a transaction that is beneficial for him twice [5].
- *Man-in-the-Middle (MitM) attack*: A third party with malicious intentions intercepts and forwards the communication between two or more endpoints in a network [6].
- *Eclipse attack*: The attacker overtakes all incoming and outgoing connections of a node, thus isolating it from the network. The attacker can use this to give an altered view of the network to the node [7].
- *Denial of Service*: Attackers make the system unavailable to the users, for example by creating a buffer overflow. In a reputation system this attack can translate to preventing the calculation and dissemination of reputation values [8].

Sybil attack: Sybil attacks mostly occur in peer-to-peer reputation systems to artificially increase the influence of a node. Attackers have different objectives from the honest nodes and use their fabricated power to gain information or prevent other nodes from carrying out actions within the system [9]. Uniquely identifying users is an often used strategy to counter this, in which case an attacker can not create multiple identities. Douceur mentions various systems that use some mechanism for identifying users to tackle Sybil attacks, such as the CFS cooperative storage system, that hashes the IP address of a node [10]. However, they all rely on a central authority to verify the identity [4]. Douceur presents convincing proofs that without a central authority, entities in the system are severely limited in determining (unique) identities.

Replay attack: Communication that is intercepted and sent again might be encrypted, but only the resending is of importance to the malicious user, not so much the content of the message. In the case of a reputation system, the communication refers to the user rating someone or a transaction. A negative value given in a transaction to a competitor could be sent again, resulting in a form of slandering (See subsection II-B). There are multiple ways to avoid replay attack, such as using timestamps on the messages. By only allowing messages within a certain time range, the window of opportunity for attackers to resend the message is reduced [11]. Another way would be to use passwords for each message and directly discard them. Finally, random session keys could be used, being only valid for one transaction and not usable afterwards.

Man-in-the-Middle attack: There is a subtle difference between replay attack and MitM attack. Just as in replay attack, MitM attack intercepts the communication between nodes in the network. However, where replay attack only addresses the resending of communication, the MitM can eavesdrop, manipulate, craft, and drop traffic on the network. To mitigate the effects of MitM attack, often encryption is used [6]. The attacker can not eavesdrop or alter the content of the message anymore, but still is able to drop traffic. In the case of an online marketplace, some form of verification between seller and buyer is needed; a signature letting the seller know that the buyer genuinely wants to buy the product and is willing to pay for it. Similarly, the buyer wants verification that the seller is in fact the user of the system that holds the product. By cryptographically signing the communication between the two parties, a man-in-the-middle could be avoided.

Eclipse attack: Eclipse attacks occur on the blockchain, where the victim's view of the blockchain is filtered or their compute power is redirected to be wasted or used in the benefit of the attacker [7]. Two of the mentioned solutions to this problem are (1) disabling incoming connections, and (2) using whitelists, where some of the outgoing connection-nodes are fixed to well-known connected peers. But this counters the open and decentralized nature of the blockchain. One of the ways to lower the chances of an Eclipse attack, is to make sure there are only unique entities on the system. For an eclipse attack, one would need multiple nodes, but if the creation of multiple nodes would be impossible, the only way to perform the eclipse attack would be to seize the power of other existing nodes. Heilman, Kendler, Zohar and Goldberg explain a number of other counter measures against the Eclipse attack, some of which are implemented in Bitcoin [7]. They include the selection of connections within the blockchain network, but the inner workings are blockchain-specific and out of the scope of this research report.

Denial of service: Denial of service in an online marketplace using a reputation mechanism can be hurtful in several ways. Especially centralized systems that have no type of redundancy are vulnerable to these types of attacks [8]. Attackers may try to overload the system to prevent the marketplace from operating. This is an attack to the whole system, rather than against a certain node or for personal benefit within the system. Next to that, they may want to corrupt the reputation system as well. If the reputation system is automated and decisions need to be made in a short time span, subverting the reputation system may give the window of opportunity to use the system without their negative reputations being known. This may apply to peer-to-peer systems as well, as parts of the reputation system are inoperable, the data is routed along different paths that don't hold these negative ratings of the attackers. The often used solution against DoS attacks is using redundancy.

B. Reputation fraud

In [8] the authors survey attack and defense techniques for reputation systems. In this often-cited article five different types of attacks are classified by identifying which system components and design choices are the target of attacks. Assumption is made that every attacker is already authorized to access the system and can participate in the marketplace. Moreover, the attacker's behavior is based on self-interest.

Note that we already discussed DoSand therefore leaving this type of attack out. The following four remaining types of malicious manipulation of reputation systems can take place:

- *Self-Promoting*: Attackers manipulate their own reputation by falsely increasing it.
- *Whitewashing*: Attackers escape the consequence of abusing the system by using some system vulnerability to repair their reputation. Once they restore their reputation, the attackers can continue the malicious behavior. Often attackers will attempt to re-enter the system with a new identity and a fresh reputation.
- *Slandering*: Attackers manipulate the reputation of other nodes by reporting false data to lower their reputation.
- *Orchestrated*: Attackers orchestrate their own efforts and employ several of the above strategies.

Each of these attacks accompanied with their defense mechanism will be explained.

Self-promotion: One form of this attack is when an attacker fabricates fake positive feedback about itself or modifies its own reputation during the dissemination. Authentication of the source data doesn't prevent this type of attacks. A group of identities - potentially from one entity - can still collude to promote each other. This collective feedback of dishonest users enables them to improve their reputation at a faster rate than honest participants and therefore also counter the effects of negative feedback.

Self-promotion can be mitigated by preventing users from creating multiple accounts on the platform and by requiring a proof of successful transaction before enabling to give feedback on the counter-party. When a group of users collude with the intent of self-promotion, the network will have to detect this group of dishonest users that primarily interact with one another. A method to mitigate the effect of a group of colludersis to introduce relative trust instead of a global value. This results in the group of colludersonly increasing their trust within this group, without having an effect on users outside of this group. A more natural way to demotivate this attack is by introducing transaction fees such that it becomes costly for colludersto create fake transactions with one another.

Whitewashing: One form of this attack is when an attacker re-enters the system with a new identity. Especially reputation systems based on only negative feedback are vulnerable for this type of attack. Dishonest users can scam and get away with it by re-entering the platform with a clean reputation. Systems that use both positive and negative feedback can also be vulnerable if they do not discriminate between old and recent actions. For example, if an attacker generates positive reputation for a while, it can perform malicious attacks for a short duration with little risk as the previous history will outweigh the current actions.

Ways to mitigate whitewashing include taking limited history into account and - again - preventing users from re-entering the system by the means of creating multiple accounts. Also, the reputation system should distinguish long-term good behaving users from newcomers. This can be achieved by allowing both positive and negative feedback.

Slandering: What this type of attack distinguishes from the other attacks is that this can be done by a single identity. However, the effect of this will be relatively small depending of the sensitivity of the influence of negative feedback on the reputation of a user. If negative feedback has a large effect on this reputation, slandering attacks will be facilitated. But if the sensitivity is low, dishonest users are enabled to exhibit bad behavior for a longer period of time. The same logic holds for groups of attackers. When the sensitivity is lower, it is more robust against malicious groups but it allows the users in that group to continue their dishonest behavior.

Slandering attack can be mitigated by making sure that feedback can only be given after a valid transaction took place. This way malicious groups can not slander a user without conducting business. But users from a group can still approach the target separately. If they have a high reputation and are trusted by the target user, then each of them can still slander the target. Therefore preventing users from creating multiple accounts might mitigate this further. Finally, having a local trust system limits the effect of this attack in reputation systems with dense interactions. Having a global reputation will allow the sybils or colludersto decrease the reputation value of a peer for everyone in the system.

Orchestrated: Orchestrated attacks can only be performed in groups. In contrast to the above-mentioned reputation attacks where one strategy is applied, in orchestrated attacks various coordinated attacks are combined. One form of this attack is the oscillation attack where a group of dishonest users are divided in multiple teams with each team playing their own role at some point in time. Some teams perform a certain type of attack while other teams play the role of honest users. The honest teams serve the dishonest teams by limiting the speed of decline of the reputation of the dishonest teams until the reputation of those dishonest teams is too low to benefit from the system. In the mean time the honest team will have built a positive reputation. At this point the teams swap roles so that dishonest teams can rebuild their reputation and the previously honest teams can perform attacks.

There is not a single strategy to mitigate orchestrated attacks as they depend on the role of the teams. However, the system should try to require a lot of participants to form a group in order to achieve the desired effect. Therefore, preventing users from creating multiple accounts will help against orchestrated attacks. Moreover, trust is more likely to be an issue of concern to developers working in large distributed teams [12].

C. Transaction fraud

Fraudulent behaviour in the transactions between two users should also be taken into account. Transaction fraud is all of the frauds that can happen with the transaction between two users [13]. Transaction fraud is one of the most common types of fraud as there is not a lot of knowledge involved in executing these scams. The following scams fall under transaction fraud:

- *Payment fraud*: When the seller falsely claims the buyer has not paid or refunded the money. Also when the buyer does not pay or when the buyer refunds the money after the item was shipped.

- *Package delivery fraud*: When the seller does not deliver the item or the buyer falsely claims the item was not delivered.
- *Package content fraud*: When the seller delivers the wrong item or an empty box. Also when the buyer falsely claims the received item was not as advertised or that it was shipped an empty box.

Each of these frauds and their possible solutions will be explained.

Payment fraud: With payment fraud buyers will seem to have paid for an item, so the seller ships the item. However, in reality, the buyer either refunds the money or has provided a false proof of payment. In this case the seller never receives the money for the products that were shipped. The seller could also falsely claim the buyer has committed payment fraud. A good online marketplace should deal with this fraud so it can not happen so easily.
A possible solution to this problem is to use a non refundable and trackable payment system. Using this, the exchange of money can be verified by another party. This would make committing payment fraud impossible, as the claims can be verified.

Package delivery fraud: Package delivery fraud is also a common fraud on an online marketplace. The seller can possibly not ship an item that the buyer has actually paid for. The buyer could also claim that a package was never actually sent. Being able to confirm these claims is vital in any functioning marketplace. For long it was not possible to verify these claims, only recently it became possible to track packages.
A possible solution to this fraud could therefore be escrow combined with an integration with a package delivery company. The money of the transaction could be held in escrow by a trusted third party or in a smart contract, while the seller ships the item. The package delivery company can then verify whether a package was actually sent and delivered. When the package was not sent, the money can be returned to the buyer. If the package was actually sent, the money will go to the seller. This would solve the package delivery fraud problem.

Package content fraud: To circumvent the package delivery fraud solution, sellers or buyers can commit package content fraud. The seller could ship an item that is broken or a different less valuable item. Sometimes even an empty box is sold. The buyer could also falsely claim that he was sent an item that was not as advertised. These can both be valid claims and should ideally not be possible in a marketplace.
There are possibilities to track packages and whether they have arrived, it is however not possible to know whether the seller has actually shipped the advertised items. Solving this problem would require a human judge that judges on these cases with more information from both parties. This extra information however, can easily be forged. It is therefore difficult to actually find out who was scammed. The best solution there is, is preventing transactions with these scammers. That is achieved by having a solid reputation system that can not be fooled by the scammers.

III. Computing reputation

A. Different types of reputation systems

In the case of a system where multiple peers interact with each other, some form of trust or reputation is needed to avoid malicious use. The reputation of peers can be computed in multiple ways, but one clear distinction is the following, given by Ziegler and Lausen: "Trust metrics may basically be subdivided into ones with *global*, and ones with *local* scope." [14]. According to Ziegler and Lausen, reputation can be seen as global, while trust focuses more on the local scope. But there is an overlap. Regardless of which scope is used, different properties of users and their interactions with other peers can be used to compute their reputation. First these different properties will be discussed and how they can be aggregated to get a reputation value. Finally, the difference between *global* and *local* scope will be explained and to what extent these different reputation values will be used in our system.

B. Reputation metrics

1) Feedback other users: The feedback of other users often forms the basis of someone's reputation in the existing systems. Users can give feedback on transactions in the form of a rating. With eBay for example, a user has three possibilities of rating a transaction; +1 for a positive experience, 0 for a neutral experience, -1 for a negative experience [15]. Multiple of these ratings are then aggregated into a global reputation value. PeerTrust also takes into account the feedback of other users as a basis of their reputation system. The initial reputation is calculated by taking the weighted average of the amount of satisfaction a peer receives for each transaction [16]. However, more metrics are taken into account, as we will discuss later.

2) Time: The age of given feedback can be taken into consideration when calculating the effect of this feedback on the reputation [17]. For instance, a user has been selling products for years and has built a good reputation. When this user becomes malicious and tries to scam other users, it takes a long time to lower the years of good reputation that was built. It is therefore necessary to implement some form of weighting to the age of the feedback, where newer feedback is weighted more heavily. There are two main ways to accomplish this behaviour. The first way is to decrease the effect old feedback has on the reputation through the use of decay. This is for instance used in Yelp to lower the impact of older reviews on the reputation. Another way is to ignore feedback older than a certain age. This is for instance done by eBay that ignores ratings older than 12 months.

3) Amount of transactions: The amount of transactions is also important. A new user with one positive transaction seems less reputable than a user with thousands of positive transactions, even if they have the same reputation percentage. That is why it is important to state the amount of transactions the feedback is based on or to incorporate it into the reputation rating of the user.

4) Amount of transactions with same user: In order to stop users from raising each other's reputation repeatedly, the amount of transactions between two users can be taken into account. When a person has had multiple transactions with the same person, the influence of the new feedback could be lower. If done correctly, this will compensate for the spamming of feedback between two users. However, this will only work if new accounts can not easily be created. Else, one of the users can repeatedly create new accounts and raise the feedback of the other user.

5) Value of a transaction: When implementing fees as a percentage of the price, the value of a transaction can also be taken into consideration [17]. Higher value transactions will cost more in fees and therefore increases the value of reputation put into the account. Reputation in this case, costs money to generate. This will help prevent Sybil attacks [18]. If the money cost to create a reputable account is greater than the reward for the scammers, it is no longer profitable to scam other users.

C. Quality of the reputation

1) Credibility of users that give feedback: "One of the major issues with online rating systems is the credibility of the quality ranks that they produce ... For various reasons, users sometimes might also have vested interest to post unfair feedback, either individually or as an organized colluding group." [19]. The credibility of users and their feedback is thus important to be able to rely on the reputation system. PeerTrust gives a higher weight to feedback of users with a higher credibility [16]. They first mention an extra feedback-reputation value next to the transaction-reputation value, but argue that this would make the system more complex. Instead, they provide two ways in which the credibility of users in their system is inferred. One way is to use the transaction-reputation value as a measure for credibility. Users with a low transaction-reputation most likely also will provide false or misleading feedback, to hide their own malicious behavior. However, users with a good transaction-reputation might also give false feedback to other users, to have an advantage over their competitors. Therefore another form of credibility-measurement is needed. The second credibility measure PeerTrust mentions, is that of feedback similarity with other peers. The feedback you and another peer have provided on multiple users is compared. The more similar your feedback ratings are to the other peer's feedback ratings, the more credible these ratings are for you. This does not always work however, as similar rating data between two peers may be sparse in a large marketplace system.

2) Bias: A users bias can be taken into account for determining the value of the feedback this user gives. If the user gives a very high rating on average, this user is probably not very critical in their reviews. The impact of this users positive feedback can then be lowered to remove the positive bias. This is done in many reputation systems including eBay's reputation system [20]. This bias removal can also be done the other way around to remove negative bias. Users can give varying average scores and therefore removing bias is essential to give a correct reputation score.

D. Aggregation methods

Computing the actual reputation value can be done in several different ways [17].

1) Counting: "Reputation is computed by either summing positive and negative ratings, or averaging ratings." [17]. These ratings are easy to calculate and are therefore used by most online marketplaces.

2) Discrete: A discrete measure can also be considered. The actual rating is produced by looking up the values in a table. This is not used very often probably because of non optimal computation aspects.

3) Flow: Flow algorithms can also be used. A well known flow algorithm is that of PageRank [21]. In the PageRank algorithm the trust of the feedback givers is also taken into account into calculating the trust of the user. Although PageRank is created for centralized systems, distributed implementations also exist [22].

E. Representation

There are several different ways to describe and interpret reputation information [17].

1) Binary: When reputation is interpreted as binary, a user can be either trustworthy or not. This is a very limited way to describe reputation, and is usually combined with the other methods to describe reputation more extensively.

2) Discrete: There are several ways to use a discrete interpretation. For instance, summing all positive feedback (as +1) and all negative feedback (as -1). An alternative way is to use a feedback scoring system with discrete values. Then rounding of the value while interpreting to a round number.

3) Continuous: Reputation can also be interpreted on a continuous scale. For instance, the percentage of successful transactions is a continuous scale. Another way of using a continuous scale, would be to place the reputation value in the interval $[-1, +1)$. Having a value of 0 would mean no trust, while a negative value implies distrust. No trust is the absence of trust information, which is different from actual distrust. A total trust of +1 is excluded from the interval, to let the user think about the situation and avoid 'blind trust' [23].

4) Text: Another way to describe reputation is by the use of text. Users in this system can give textual feedback to other users. This is usually combined with a numerical description as it is hard to combine feedback text to a single understandable overview.

F. Reputation scope

1) Global reputation: A global reputation system is a reputation system where the feedback of all the users that have given feedback to another user is used to calculate the reputation [17]. The reputation system aggregates all the feedback of a user and displays it in a single understandable format. This leads to a consistent, global view of every user in the system. Some examples of marketplaces that use global reputation systems are eBay and Amazon. Global reputation systems scale well, they are however vulnerable to several attacks. Global reputation systems are especially dominant in

centralized networks, as there needs to be a central authority that maintains and distributes this global reputation information [24].

2) Local reputation: As opposed to global reputation, local reputation takes into account the different levels of trustworthiness between sets of peers. Persons that peer *a* trusts, may be completely different from the persons peer *b* trusts [14]. A single global value for each peer does not encompass these differences in trust, so a local trust value is used. Also, if a peer already has an experience with a certain user, the global reputation might be less relevant. A local trust-value is thus either a user's personal experience with a peer, or the peer's reputation for trustees of the user, or a combination of the two. An example of a local reputation system is Appleseed, which makes use of local group trust [14].

3) Combined: Local reputation systems are more personalized and give a more relevant reputation of a peer for a user, but they are not always effective. In a big system, there may not be enough relevant information as the user does not have enough trustees that have interacted with the peer. Therefore, the system may benefit from a global reputation system, as this takes into account all the available information of a peer in the system [25]. As Gohari, Aliee and Haghighi mention, multiple studies demonstrate the positive effects of both global and local reputation systems, but omit the combination of the two. "When there is not enough information about a user, it is reasonable to put more emphasis on global trust than on local trust and vice versa. Thus, it seems more appropriate to combine local and global models into a single model instead of having a fixed choice for the target model. [25]".

IV. Network architecture

In this section, we will take a look at three main types of network architectures which are client-server architecture, peer-to-peer architecture and hybrid architecture which is a combination of the former two architectures. These systems are used for the same purpose of establishing a network infrastructure between different market participants. However their implementations differ significantly and require a thorough review on different aspects. These aspects are handled respectively: Scalability, Persistency, Consistency, Affordability, Security and Anonymity.

A. Client-Server Architecture

Client-server is a network architecture where each node in the network is either: a client or a server. The server is the central node where multiple clients can connect to and send their requests. The server then replies with information to each individual client [26]. In the context of a marketplace, buyer clients will request listings from the servers which are posted by the seller clients.

If a buyer client decides to purchase/rent from a seller client, he can enter a digital contract which is stored in the server. The server serves as a validator and in some cases, the enforcer of the contract. In systems such as Amazon or Alibaba, the server is the middleman who oversees transactions, enforces policies and keeps track of shipments. Other information such as client information, authorization credentials, listing information are also saved on the servers. Server hardware is needed next to the hardware of the service consumers (the client), but most of the processing work is performed on the server side.

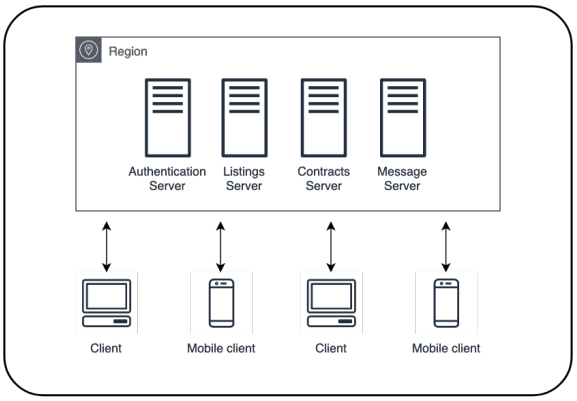

Fig. 1. An example of a client-server architecture

B. Peer-To-Peer Architecture

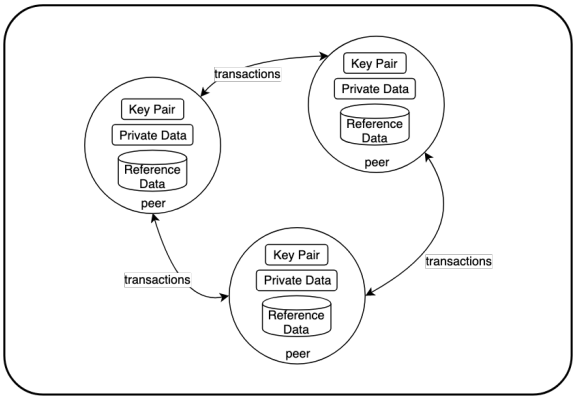

Fig. 2. An example of a peer-to-peer architecture

Peer-to-peer is, in contrast to client-server, an architecture where each node in the network plays the same role and has the same responsibility as others. To be more specific, there is no central server [27]. In a peer-to-peer network, nodes connect together to serve each other on content depending on their needs. In the context of a marketplace, the buyer clients will request listings stored from other clients nodes which are posted by seller clients.

This also means the listing data and other information such as client information, authorization credentials, contracts and ratings are distributively stored on the nodes across the whole network. Since we do not have a central point of authority to confirm and enforce the contracts and policies as in the case of the server in the server-client architecture, these are confirmed and enforced by a combinations of nodes. There should be no other hardware required except hardware from the service consumers.

One possible implementation of a peer-to-peer marketplace is to build the peer-to-peer marketplace on top of different network layers which already exist, in order to take advantage of the services they already provided. For the storage system, the marketplace can make use of a decentralized storage system such as IPFS [28]. For the payment system, the marketplace can make use of a decentralized cryptocurrency platform such as Bitcoin[29]. Finally to be able to make use of persistent contracts the marketplace can make use of a smart contract [30] such as Ethereum.

C. Hybrid Architecture

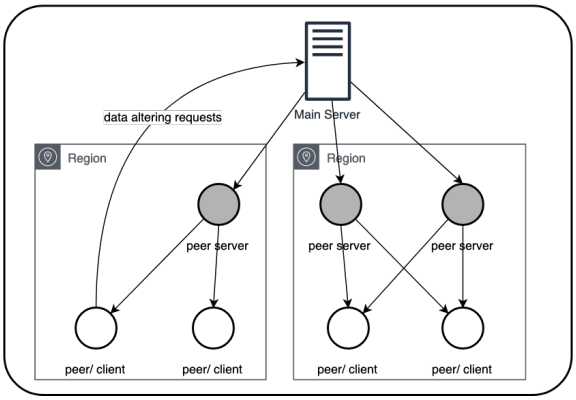

Fig. 3. An example of a hybrid architecture

Hybrid architecture tries to combine the advantages of client-server and peer-to-peer architectures by adopting practices from both. Like the client-server architecture, there will still be main servers which will serve the clients, but their work is reduced by making use of other client nodes as servers [31]. In the context of a marketplace, a group of nodes can be dynamically chosen by the server based on their bandwidth performance, up time and storage capacity, to serve requests from other client nodes. We will call this group of nodes peer servers. This group can be selected based on the region that the node is in, inspired by the concept of regional servers.

Data which are long lived such as market listing data which are regularly requested by the clients can be stored on the peer servers. This way the main server does not have to handle all the requests for these types of data and can instead, use all resources on handling data-changing requests. Data-changing requests are for example requests that modify the listing data, create contracts and update ratings. The reason that these requests should be handled by the main server is to achieve persistency which the peer-to-peer architecture lacks and address the development costs of the client-server architecture.

D. Comparison

In this section, each aspect of a marketplace network architecture is visited where we will make a systematic comparison between the three types of architectures.

1) Scalability: Scalability is one of the biggest challenges for an online marketplace [32]. Scalability is the ability for the system to stay operational efficiently while new users join the marketplace without running out of resources. Resources usually mean extra network bandwidth, processing hardware and database.

In client-server network, scalability is an issue [33]. One primary way to support a growing number of users is to add more hardware such as increasing servers and storage. The system also needs to be re-balanced in order to address imbalance of connections in different regions. This is usually costly. There is a risk of architecture hardware being underutilized. Finally, there is the risk of having a single point of failure: if a server fails, clients will get a reduction in performance or even lose access.

In a peer-to-peer network, as more and more users join the marketplace, more resources are also available. When a user joins the network, his or her hardware is made available for processing and storage of the marketplace data. It seems as if scalability in peer-to-peer market is highly achievable and is free. However, without an efficient implementation of how data is processed and distributed across the network, a potential throughput bottleneck can become an issue [34] since peers' hardware are household machines which have lower processing capacity than enterprise machines.

In a hybrid network, it is easier to scale compared to a client-server network since extra resources are replaced by capable peer nodes. This reduces the amount of requests that the main server has to handle significantly and thus is a step up compared to the client-server model. However, hybrid networks are still not as scalable as peer-to-peer networks since a big part of work still needs to be handled by the main server. Also, not all the peer nodes have to contribute to the network.

2) Persistency: When transactions are recorded in the system, they need to be store in a place such that a user can request it whenever it is needed. The data needs to "stay alive" even if it is not being used. One way of achieving persistency is to store the data in a database which is always running.

In a client-server network, information is stored on the server and therefore is always available upon request [26]. Every time a user loads an advertisement, the information is looked up and returned. Persistency of data in the client-server model is high.

In a peer-to-peer network, the persistency of the data depends on the online time of the peer nodes and how the data is distributed across the network. This means that if a traditional peer-to-peer network, where all nodes contain a specific subset of global data, goes offline, that data is then not available to be read or edited. Another point is that since data is stored in a distributed manner, an efficient mechanism must be implemented in order to retrieve data quickly and with minimal resource. However, there currently are decentralized database solutions to address this problem.

A Hybrid architecture makes use of server, so it can achieve persistency by storing the data on the main server. This data is then replicated on the peer servers.

3) Consistency: Users of a marketplace should be able to see the same information when the same listing is requested.

This also means that when the listing data is changed it should be changed across all the copies of the listing data.

In the server-client architecture, consistency is high since the server is the only entity that has to make change to the database. All clients get their listings from the server so they will all have the same copy as soon as the data is modified. In short, there is only one version of data which is stored on the server.

In the peer-to-peer architecture, consistency is more difficult to achieve since data is by default distributed and there might be multiple copies of the data that need to be updated. To efficiently update all of these copies in nodes spread across the network is geographically challenging. This mean that when a contract is entered for a listing, which in turn marks the listing unavailable, changes to the data should be made in such a way that there can be never a second contract entered for the same listing. Again, there are several proposed solutions for a consistent peer-to-peer storage that we will have to take a look into.

With the hybrid architecture, consistency is achieved easier than in the peer-to-peer architecture, since it makes use of the main server. Any changes of the data will be handled by the main server and passed down to the peer servers. However, the consistency of listings data must still be achieved between the peer servers and there should not be any bottleneck to these peer servers. Overall consistency of the hybrid architecture is higher than the peer-to-peer architecture but is lower than the server-client architecture.

4) Affordability: In order to get an online marketplace up and running, the developers need to spend capital to purchase or rent infrastructure hardware needed for the operation. The costs of this varies greatly per implementation.

With the client-server architecture, it is necessary to purchase or rent server and network equipment. A server requires a lot of processing power, at least for a global marketplace, large hard disks and a lot of memory meaning they can be very expensive. Another one time cost is the set up cost. Next to that there is also maintenance costs, energy and floor space costs. Whether the equipment's are purchase or rented, these costs are unavoidable. This could negatively affect the costs of transactions between market participants.

In contrast, peer-to-peer architecture makes use of node's computational power and resources. These resources are provided by the users that join the network and therefore the developers need not to purchase any servers or network equipment. There is no setup costs, no maintenance costs, energy and floor space costs. Therefor the affordability of peer-to-peer model is high: the costs to set up and keep the system running for the developers is little or even none.

Hybrid architecture stands in the middle. Since it makes use of a central server, there is costs associated with it. However, due to the support of peer servers (clients that contribute to the network), the central server does not require as much as processing power and storage as the server in client-server architecture. The hybrid model can deal with a large user base but is more affordable than client-server architecture.

5) Security: When an online marketplace is up and running, there will be attempts from malicious users to exploit the system, in a technical scope, in order to gain monetary or reputation benefits. This could be unfair for other users and in the worst case, render the marketplace unusable. The architecture should be designed to have ways to prevent or detect such attempts. Security is one of the most difficult aspect of network architecture, therefore it can only be briefly discussed in this section.

In client-server model, all requests have to go through the central server in order to be processed. The server acts as a "guard door" where every requests is analyzed and validated. This therefore gives client-server architecture an easy way to detect and prevent frauds. One other concern of security of this model is, since the data are mainly stored on the servers, there is a security risk of data being leaked if the database is compromised.

In peer-to-peer model, the peer nodes all manage a part of the resources. Security is harder to achieve than in client-server since the analysis and validation of requests happens on a distributed level. Since requests no longer go through a verified "guard door", there needs to be a mechanism such as network consensus to verify them. Another concern for this model is data leakage and tampered network consensus which can happen when a malicious party has gained control over a large part of nodes. Security is harder to achieve in peer-to-peer architecture, but it does not necessary mean that peer-to-peer model is less secure than client-server model.

Hybrid model makes use of a central server thus all the data altering requests must go through the "guard door", just like in the client-server model. It is easy to detect and prevent any fraudulent requests. However since this model makes uses of trusted peers as servers and give these peer servers the privilege to process and store data, it is extremely important to secure these nodes.

6) Anonymity: Anonymity is an important aspect in an online marketplace. Users should have the ability to conceal private information, stay anonymous in a transaction and not being prosecuted by authorities up to a certain degree.

In client-server and hybrid architectures, since the main server is the only one party that process all the request from the clients, it is easy to know where they comes from and intercept the behavior of targeted clients, revealing their identities. Since the servers are centralized points, they must always comply with the regional authorities or risk being prosecuted. In the worst case, the data could also be used by (authoritative) parties for malicious purposes.

In contrast, in peer-to-peer network with routing overlay networks is harder to take control over and tracking down a peer's identity. There is no single point of control since the nature of peer-to-peer is decentralized and is not a single organization. If well implemented, fully anonymous network can be achieved.

A summary of the comparison of the discussed aspects is presented in Table 1.

TABLE I
TABLE CONTAINING THE REVIEWED NETWORK ARCHITECTURE AND THEIR ASPECTS

	Client-Server	**Peer-To-Peer**	**Hybrid**
Scalability	costly	no cost	moderate cost
Persistency	easy to achieve	difficult to achieve	moderately difficult to achieve
Consistency	easy to achieve	difficult to achieve	moderately difficult to achieve
Affordability	difficult to achieve	easy to achieve	moderately difficult to achieve
Security	easy to achieve	moderately difficult to achieve	easy to achieve
Anonymity	difficult to achieve	easy to achieve	difficult to achieve

V. POSSIBLE SOLUTION

To find a good solution the researched factors have to be taken into account. For the network architecture, a decentralized architecture seems a good solution. This is because anonymity between users can be more easily enforced, as users are in control of what information they disclose. A decentralized architecture has also no costs when it comes to scalability, it is however more difficult to achieve. One of the biggest problem of marketplaces as researched, are Sybil attacks. Sybil attacks are hard to mitigate, especially in a decentralized marketplace. To prevent Sybil attacks, there needs to be a good reputation system that can work on a decentralized marketplace. Luckily, research has shown that several good reputation systems exists that are proven to work on a decentralized architecture. In order to prevent users from creating multiple accounts and fooling the reputation system, a solution that makes sure that every user can only have one account is necessary. For this self-sovereign identities can be used in combination with a decentralized marketplace. Our suggested solution is thus a decentralized marketplace with self-sovereign identities and a reputation system.

VI. DISCUSSION

In this section, we will discuss various important aspect in regarding to the proposed possible solution. These aspects are GDPR-compliance and anonymity.

A. GDPR-compliance

GDPR - short for General Data Protection Regulation is a set of regulations which requires business operating with data to enforce customer data protection practices. Non-compliance could cost businesses and projects dearly. This is why it is essential to have a discussion regarding where the possible solution stands with GDPR-compliance. So far, the possible solution given by us does not have enough in depth implementation details in order to concludes that they are GDPR compliance or not. However, as stated we would make use of self-sovereign identity solutions, which means we will be dealing with highly sensitive personal data. That is one point to keep in mind to design a system in such a way that this data is securely when it is being transferred or processed within our system. Since we will be making use of a decentralized architecture, users personal data is not clustered as in a client-server architecture and thus reduces risk of major data breaches. Personal data should be kept at the owner's device. Keep in mind that we should value users privacy rights by allowing them to modify and remove their personal information from the network anytime. Also all data we have of the users, such as their purchase history and rating history should be completely transparent to them.

B. Anonymity

Users should have the right to stay (pseudo-)anonymous if required. Even though we make use of self-sovereign identity solutions, we need to make sure that it is solely used for the purpose of preventing the creation of fraudulent accounts and that the identity from within the market place cannot be in anyway linked to the identity used for creating an account. This way we prevents the user within the market place being traced back to their origin identity. User information that can be used to trace back identity should be kept on the user's device as much as possible to preserve anonymity.

VII. CONCLUSION

In this paper we discussed many of the different attacks and frauds that can happen in online marketplaces. This research formed the basis of the following chapters in which we tried to find solutions to these attacks.

Research was conducted on many of the different types reputation systems and metrics that exist. We found that there are many different solutions and ways other reputation systems deal with the attacks defined above. However, Sybil attacks in which multiple identities are forged usually remained unsolved. We then researched the different network architectures that can be used in online marketplaces. We found that the most commonly used architecture was a centralized architecture. While this is a good architecture, we concluded that it is most vulnerable to data breaches and costly to scale.

To conclude, this paper aimed to research the possible solutions of creating an online marketplace to solve the current issues with online marketplaces. We found that no solution covers all of the problems, and therefore trade offs have to be made. To answer the main question, we came up with a solution that combines self-sovereign identities with a strong reputation system. We found that this is a good solution that prevents Sybil attacks while being as anonymous as possible.

REFERENCES

[1] J. Dzieza, "Dirty dealing in the 175 billion dollar amazon marketplace," Dec 2018. [Online]. Available: https://www.theverge.com/2018/12/19/18140799/amazon-marketplace-scams-seller-court-appeal-reinstatement

[2] H. Yoganarasimhan, "The value of reputation in an online freelance marketplace," *Marketing Science*, vol. 32, no. 6, pp. 860–891, 2013. [Online]. Available: https://doi.org/10.1287/mksc.2013.0809

[3] D. van Bokkem, R. Hageman, G. Koning, L. Nguyen, and N. Zarin, "Self-sovereign identity solutions: The necessity of blockchain technology," 2019.

[4] J. R. Douceur, "The sybil attack," in *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek, and A. Rowstron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 251–260.

[5] P. Otte, M. de Vos, and J. Pouwelse, "Trustchain: A sybil-resistant scalable blockchain," *Future Generation Computer Systems*, 09 2017.

[6] Y. Mirsky, N. Kalbo, Y. Elovici, and A. Shabtai, "Vesper: Using echo analysis to detect man-in-the-middle attacks in lans," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1638–1653, June 2019.

[7] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 129–144. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831152

[8] K. Hoffman, D. Zage, and C. Nita-Rotaru, "A survey of attack and defense techniques for reputation systems," *ACM Comput. Surv.*, vol. 42, no. 1, pp. 1:1–1:31, Dec. 2009. [Online]. Available: http://doi.acm.org/10.1145/1592451.1592452

[9] G. Danezis and S. Schiffner, "On network formation, (sybil attacks and reputation systems)," 05 2019.

[10] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," *ACM SIGOPS Operating Systems Review*, vol. 35, 09 2001.

[11] K. Lab. What is a replay attack? [Online]. Available: https://www.kaspersky.com/resource-center/definitions/replay-attack

[12] L. E. Espedalen, "The effect of team size on management team performance: The mediating role of relationship conflict and team cohesion," 2016.

[13] D. Houser and J. Wooders, "Reputation in auctions: Theory, and evidence from ebay," *Journal of Economics Management Strategy*, vol. 15, no. 2, pp. 353–369, 1 2006. [Online]. Available: https://doi.org/10.1111/j.1530-9134.2006.00103.x

[14] C.-N. Ziegler and G. Lausen, "Spreading activation models for trust propagation," in *IEEE International Conference on e-Technology, e-Commerce and e-Service, 2004. EEE '04. 2004*, March 2004, pp. 83–97.

[15] H. Chaokai and W. Meng, "Comparison and analysis of different reputation systems for peer-to-peer networks," in *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, vol. 3, Aug 2010, pp. V3–20–V3–23.

[16] L. Xiong and L. Liu, "Peertrust: supporting reputation-based trust for peer-to-peer electronic communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 843–857, July 2004.

[17] F. Hendrikx, K. Bubendorfer, and R. Chard, "Reputation systems: A survey and taxonomy," *Journal of Parallel and Distributed Computing*, vol. 75, 08 2014.

[18] N. Balachandran and S. Sanyal, "A review of techniques to mitigate sybil attacks," *International Journal of Advanced Networking Applications*, vol. 4, 07 2012.

[19] M. Rezvani, A. Ignjatovic, and E. Bertino, "A provenance-aware multi-dimensional reputation system for online rating systems," *ACM Trans. Internet Technol.*, vol. 18, no. 4, pp. 55:1–55:20, Sep. 2018. [Online]. Available: http://doi.acm.org.tudelft.idm.oclc.org/10.1145/3183323

[20] H. Xie and J. C. Lui, "Modeling ebay-like reputation systems: Analysis, characterization and insurance mechanism design," *Performance Evaluation*, vol. 91, pp. 132 – 149, 2015, special Issue: Performance 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166531615000577

[21] L. Page, "The pagerank citation ranking: Bringing order to the web," 01 1998.

[22] A. Yamamoto, D. Asahara, T. Itao, S. Tanaka, and T. Suda, "Distributed pagerank: a distributed reputation model for open peer-to-peer network," in *2004 International Symposium on Applications and the Internet Workshops. 2004 Workshops.*, Jan 2004, pp. 389–394.

[23] S. Marsh, "Optimism and pessimism in trust," 07 1999.

[24] H. Zhao and X. Li, "H-trust: A robust and lightweight group reputation system for peer-to-peer desktop grid," 07 2008, pp. 235 – 240.

[25] F. S. Gohari, F. S. Aliee, and H. Haghighi, "A dynamic localglobal trust-aware recommendation approach," *Electronic Commerce Research and Applications*, vol. 34, p. 100838, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567422319300158

[26] M. R. Civanlar and B. G. Haskell, "Client-server architecture using internet and public switched networks," Nov. 30 1999, uS Patent 5,995,606.

[27] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proceedings First International Conference on Peer-to-Peer Computing*, Aug 2001, pp. 101–102.

[28] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[29] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and cryptocurrency technologies: A comprehensive introduction*. Princeton University Press, 2016.

[30] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[31] B. Yang and H. Garcia-Molina, "Comparing hybrid peer-to-peer systems," in *Proceedings of the 27th Intl. Conf. on Very Large Data Bases*, 2001.

[32] M. U. Ahmed, "ebay-ecommerce platform, a case study in scalability," *McGill University*, pp. 1–13.

[33] L. Ricci and E. Carlini, "Distributed virtual environments: From client server to cloud and p2p architectures," in *2012 International Conference on High Performance Computing Simulation (HPCS)*, July 2012, pp. 8–17.

[34] E. d. S. e Silva, R. M. Leao, D. S. Menasché, and A. d. A. Antonio, "On the interplay between content popularity and performance in p2p systems," in *International Conference on Quantitative Evaluation of Systems*. Springer, 2013, pp. 3–21.

B SIG Feedback

[Feedback]

De code van het systeem scoort 3.7 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Size en Unit Interfacing vanwege de lagere deelscores als mogelijke verbeterpunten.

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:

- NewNode(context context.Context, repoPath string) in node.go
- main() in main.go

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Dit kan worden opgelost door parameter-objecten te introduceren, waarbij een aantal logischerwijs bij elkaar horende parameters in een nieuw object wordt ondergebracht. Dit geldt ook voor constructors met een groot aantal parameters, dit kan een reden zijn om de datastructuur op te splitsen in een aantal datastructuren. Als een constructor bijvoorbeeld acht parameters heeft die logischerwijs in twee groepen van vier parameters bestaan, is het logisch om twee nieuwe objecten te introduceren.

Voorbeelden in jullie project:

- Listingmanager.AddListing in listingmanager.go

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid tests blijft nog wel wat achter bij de hoeveelheid productiecode, hopelijk lukt het nog om dat tijdens het vervolg van het project te laten stijgen. Op lange termijn maakt de aanwezigheid van unit tests je code flexibeler, omdat aanpassingen kunnen worden doorgevoerd zonder de stabiliteit in gevaar te brengen.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.