Bulls, Bears and Wolves Contextproject Final Report

Naqib Zarin, Luat Nguyen, Jasper Hu, Ashay Somay, Ymte Jan Broekhuizen

June 22, 2017

Contents

1 Introduction

The problem of proving your identity to strangers has arisen as a necessary consequence of larger societies. From early banking systems in the ancient Babylonian society to government institutions and companies nowadays, they all had so solve the same problem: how to verify that someone really is who he says that he is while minimizing the risk of identity theft.

The classical solutions used range from identity tokens to identification cards and online accounts, but essentially work the same way: a large institution keeps track of a large number of people associated with information such as a bank balance or property ownership, and its customers provide a small piece of information - either physical or in the form of login data - that prove to the institue who they are. Once identified you may use the provided service: transfer money to someone, ask for a new identity card, etcetera.

The weakness of these classical identity schemes is that there is a third party that holds all the power. They may deny services to you or freeze your account (PayPal) without you being able to do something about it [2]. When they have poorly implemented security, such as the credit card system in the United States, you are powerless as a customer. Password dumps of major websites are another example of failure in which you as a customer are powerless against the institute responsible for the leak.

Recent advancements in technology have provided alternative solutions to the classical authority-based identity systems. Decentralized applications such as Bitcoin and SSH have been developed that rely solely on cryptography for identity proof; there is no authority necessary thereby implicitly avoiding malicious authorities.

While these applications have identity proof as side-effect, their main purpose is something else (text transmission, banking). We strive to create a library that provides decentralized identity proof for any two strangers. This is done by keeping track of what users think of each other stored in an immutable Blockchain, where trusted contacts are exchanged when two people meet and trust each other in real life. This results in increasingly larger amounts of trusted contacts in your local database. The more trusted people you meet, enlarging the opportunity for safe transactions such as direct messaging, exchanging goods, or even collaborating in online projects. Since our application is not directly usable for end-users, our final product will be an API that other apps can use to manage contacts, where our library performs the task of verifying the data and securely and efficiently retrieving it.

Following the requirements, our final API should be able:

- To retrieve contacts of the current user along with a trust amount
- To add or remove contacts
- To verify if another user is who he says he is
- To submit failed or succeeding transactions with a contact to increase or decrease your trust in them

2 Overview of the Developed Product

The main goal of our library is to remove the need for an authority from the process of proving your identity, hereby liberating users from possible mistakes from this same authority. The library is meant to be part of other applications that want decentralized identity verification without going through the hassle of creating this functionality from scratch.

The library works by letting you put contact information and revokal into 'blocks': undeniable proof that you have created certain information. This means that once another user D gets from you the information that B has C as a contact, he can verify this without ever meeting B or C.

2.1 Possible Applications

Having a proof of identity system is useful in a wide range of applications.

Some examples are:

- Messaging application. By creating a decentralized messaging application, one removes the need for an authority that has all power over the messages and is able to deny service -

accidentally or on purpose. With the help of our library the application could enable you to look up users by user name without adversaries pretending to be your contact.

- Market place application that allows you to sell and buy commodities. By only allowing transactions between contacts fraud becomes much harder as any criminal needs to gain enough trust from its contacts before one fraudulent action. Furthermore, once a criminal has committed fraud, he loses a large amount of trust from his contacts so that he must again gain trust to perform another transaction.

- Large-scale collaboration tool. When working on large open-source projects it can be useful to only allow trusted contacts to contribute to a project to reduce malicious contributions and only allow high-quality code originating from trusted programmers.

2.2 Encryption

In our library the identity of a person is reduced to his private key. This is useful for our library since these private keys are small and easy to generate and manipulate. This private key represents identity by enabling others to verify that you have the private key without you giving away the private key.

While this is easy for computation, once the private key is stolen your entire identity is compromised. This library always encrypts your private key before storing and requires a password to decrypt it, however malicious applications may steal the private key once it is decoded in memory. Once this happens your identity is completely lost; an impersonator can perform any action that you also can.

3 Reflection on the Product and Process

As a group we encountered severe problems during the course of the context project. Unacceptable behavior, irresponsible group members, and unclear requirements are some of the obstacles we encountered. Despite these problems we have managed to create a final product.

The first problem we encountered happened in the first two weeks. We did not have a strong sense of what to do since the text describing our assignment was brief and left many details to the imagination. We did not take appropriate initiative to figure out what what we should do and instead made tasks that were too large (implementing the TUDelft blockchain) or outside our problem domain.

When we finally managed to produce something it was not very pretty. It was the result of all group members trying to add all functionality to only two classes, DatabaseHandler and BlockController, which were the classes that had anything to do with the blockchain logic. A third class, Block, acted as a universal data class on which all functions operated. There were unittests but they existed mainly to keep the code coverage high and did not truly test the functionality. For example, some parts of the programs allowed multiple public keys per user while other parts did not; this inconsistency was not picked up by the unit tests.

Communication was the major reason for all these problems. We had a difficult subject to tackle and we did not resolve all implementation details in the beginning; partly because it was hard to understand, partly because the communication went not so well because some group members did not like each other. In retrospection we should have made a clear interal design document describing all ambiguities such as what where to connect each block in the multichain, which fields to include in the calculation of the block hash, etcetera, then letting everyone review, understand and approve it.

We also lacked standardized UML documentation in earlier weeks. A simple class diagram would help in making sure that our code is not accumulated in god classes and also facilitate cooperation without the mountain of merge conflicts that we went through, since everyone would in theory be able to write their classes at the same time as long as the class diagram is followed.

Another issue was the continuous integration. We did not manage to get it working despite putting in significant work. The combination of Travis and Android was hard to get working and Travis timed out many times while starting the Android emulator; we were thus forced to merge branches with no Travis confirmation that all unit-tests and style checks passed.

In week 6 we decided to completely revise all our code, starting with making a proper class diagram. Using an online tool (Draw.io), we could collaborate in real-time, and after that we

planned some dedicated time to review the scheme and identify possible errors. The resulting work was split into parts that someone should develop; this went surprisingly well and although some minor issues still arose (too large tasks, pull requests not fully completed on Thursday evening) we had the idea that we mastered the group process.

4 Description of Developed Functionalities

As we are writing a library, the functionality we have written is not exposed inside an application - except our tiny demo application - but instead contained in a series of functions available through a static API class. The main overview of functionality our library provides is described in the following list.

- Storing and retrieving contacts with their information
- Verification of incoming contact data
- Trust management of your contacts
- Revoking bad contact information
- Encrypted storage of data on the disk

4.1 Contact Storage and Retrieval

The core functionality our app provides is the storage and retrieval of information about your contacts and the contacts of your contacts. Before using the API you have to initialize it with a certain name, IBAN, and public key. The API then assumes that this represents you, the owner of the device the app is running on. After this you can add contacts along with their contacts and viewing the contact information of a certain user: such as their name, IBAN, and public key.

4.2 Verification of Incoming Data

When you retrieve blocks from other users, you receive their entire blockchain and the blockchain of their contacts, enabling you to verify each block's content with the provided hash. Things become interesting when you receive new blocks in which one of your existing contacts is already involved: you can trust the received information 'a little bit' since an existing trusted contact also has trust in the receiver. When the data does not match, an exception is thrown, enabling other applications to display a warning.

4.3 Trust Management

The API also includes two methods to indicate a successful or failed transaction. The trust value of the contact involved is then updated in the database to reflect this: once it has reached a lower threshold, the contact can be revoked or the host application may display a warning. Note that these transactions are not recorded in the blockchain; they only provide a basic mechanism to revoke bad contacts and remember safe contacts. The transactional trust mechanism described in the TrustChain paper can provide a tamper-proof trust value while our application mainly deals with the spreading contact information without extensive safeguards to prevent incorrect information. Essentially this library trusts the end-user of the host application to provide correct information; an integration with the TrustChain model could remove this dependency by recording a real-life exchange as transaction that increases trust in contact information of both users.

4.4 Contact Revoke

While trust is not calculated automatically based on the trust of other contacts and transactions and while the revokal of contacts with low trust does not happen automatically, our API still supports the revokal of information. The host application using our API is responsible for determining after how many bad transactions a contact should be revoked. A revoked contact does not show up when you are searching for contacts, and once a contact is revoked it can never be added back to the database.

4.5 Encrypted Data Storage

In the last weeks we have developed a mechanism to encrypt and decrypt the blocks before they are sent to the database. Since crucial information is stored inside these blocks, other applications are now not able to read the data without access to a password. Details of the algorithm are found inside the Architecture Design document.

5 Human Computer Interaction Design

As a programmer you want to know whether the product you made meets the requirements of the product owners and the users. However in order to enforce quality, it is necessary to make sure that the product is tested for user-friendliness before you release it. Since our product is a library, our end-users will be programmers. This means that in order to make our library user-friendly, we have to test if what our library provides is easy to use. Our library exposes a single static API - Application Programming Interface - class with which other programmers can interact. In this chapter we will describe our methods and results of the user-friendliness tests that we have executed on end-users. What we want to know is: Is our API reliable, effective and easy to use for other programmers? Note that we do not care about the Graphical User Interface since it is not our goal to make a user-friendly and a self-learning GUI.

5.1 Methods

In this section we will discuss the methods we used for evaluation. There are two types of evaluations, analytical evaluation and empirical evaluation. In analytical evaluations the users are not involved in the discussion about the usability of the program, instead relying on the intelligence and experience of the developers [3]. On the other hand, an empirical evaluation involve end-users in the evaluation. This documents describes the empirical evaluations. First, we will say something about the participants. This will be followed by a section about how we measured the results. Then we will discuss the procedure and we will close this section by providing the empirical methods we used.

5.1.1 Participants

The participants that we used for our tests were five students of TU Delft. All of them were doing the bachelor Computer Science and Engineering. The participants were familiar with the concept of Blockchain and Trust chain. This makes them candidate users of our product. They were members of two groups that needed our API in order to make their own products usable. All of the users were about the same age and had the same gender. We also distinguish users who have experience in making an API from user who have not. We want to make sure their feedback has more weight than others. Although the participant might know each other we make sure they can not consult in between the tests.

5.1.2 Measurements

A good API requires seven important properties as we will discuss in section 5.2.4. We made an A4 format survey that contained these properties, on which the test subject had to write down for every property whether they strongly disagreed, disagreed, neither agreed nor disagreed, agreed or whether they strongly agreed. Every box filled in resulted in a score. Strongly disagreed resulted in the lowest score of one, while strongly agreed resulted in a highest score of five. We will calculate the average and we are satisfied when this average is above three.

5.1.3 Procedure

As mentioned before we do not have a Graphical User Interface. This means that we had to find a creative way to test our API. We invited five different programmers to our project room at different times. We sat down with a test subject and first told them that we will save their feedback and made clear that this data will be deleted before the 24th of June, 2017. We also told them that they could quite anytime. We emphasized that it is the system that is being tested, not the users.

After this introduction we handed a manual document for the usage of our API over to the test subjects and gave them a few minutes to study this. They could ask us questions when something was unclear. We have to protect the idea of an API and tell them whether a certain question is appropriate or not. It should not be in their interest to know what the architecture of our program is and these kind of questions will not be answered.

Since the three Context Projects will not be integrated after all, our API will not be used. We can not let a test subject test our API in the traditional way (by letting him use our library in terms of coding). So we wrote Unit Tests for our API class without documenting these tests. We then asked the test subjects to study our tests and say aloud what they think and what they see and why they think it is there.

We also gave them the task to write a simple JavaDoc above each test. This way they are forced to analysis the tests and it is a nice way for us to find out if they find it clear. These documentations will be saved and analyzed later on. After the test subject finished documenting the Unit Tests, he was asked to fill in the A4 format survey that we will discuss in section 5.2.2.

When the next test subject walked in, the procedure repeated. Hence the JavaDoc of the predefined Unit Tests is removed and has to be filled in by this new test subject.

5.1.4 Analysis

After collecting the data that we gained from the tests we analyzed it. Does the JavaDoc correspond with our own JavaDoc? After studying the JavaDoc, we looked at the filled in A4 format agree/not agree paperwork. We counted the points and made a staff diagram where we filled in the average of the given points for every question. This way we could analyze which parts of the API we have to enhance.

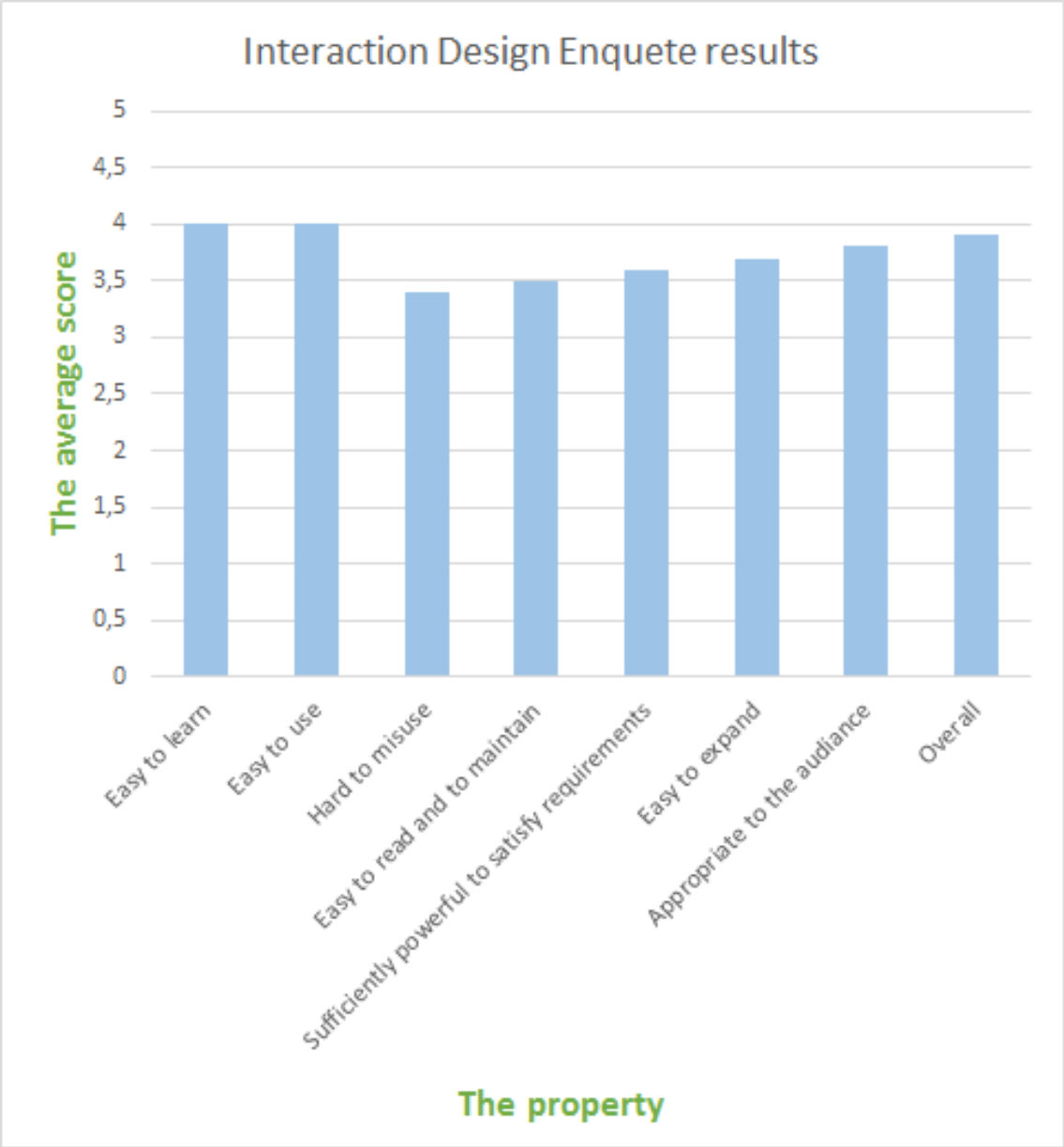
5.2 Results

This section focuses on the results we retrieved from the tests with the test subjects. We have different types of results. We have the results of the survey that the five test subjects had to fill in, the voice recordings of the testers while they were writing the JavaDoc and we also have the notes that we wrote during these sessions. In section 5.3.1 we will give an overview of the statistics of the data we collected. In section 5.3.2 we will discuss the problems that we identified after analyzing the statistics and other data.

5.2.1 Statistics

A good API has to score high on seven properties [1]. It should be easy to learn, to use and to expand. It also should be easy to read and to maintain the code that uses it. Beside these properties it also should be hard to misuse and appropriate to the audience. The last property it should have is that it should be sufficiently powerful to satisfy the requirements. For each property we asked the testers whether they strongly disagreed, disagreed, neither agreed nor disagreed, agreed or strongly agreed. As mentioned in section 5.2.2, every filled in box had their own weight. Strongly disagree had the lowest weight (one) and strongly agree had the highest weight (five). We computed the average for every property as stated in table 1. As you can see our average scores are very good. Our goal was to get an overall average of at least a 3.0. Our overall average score after the tests is a whopping 3.9.

Average time writing JavaDoc each member : 14 minutes. This is pretty quick and we are happy with this result.



5.2.2 Problems identified

As you can see we our lowest score is for the property 'API is hard to misuse'. This is mainly because some testers had no idea how to misuse an API and filled in the box 'neither disagree nor agree'.

It was not for every tester clear what the difference was between an Acquaintance and a Contact. They said they were not familiar with the word acquaintance, but after explaining this a lot of the testers did understand the difference between a contact and an acquaintance. We could change it into PairedPerson but we choose to stick to acquaintance because we believe that acquaintance is a better name and we can not take into account someone's english vocabulary. When writing an API Manual, we will explain what an acquaintance is, so this way this problem should be resolved.

We also got some complaints about how we named some methods. After receiving this feedback we decided to change some method names. For example we changed 'verifyIban' into 'verifyIbanTrustUpdate'. The naming is very important, so we were grateful for this.

Also did we receive the feedback to changed the name of our API. By its name it is not clear what kind of API this is. We decided to change the name into 'BlockChainAPI' since our API is related to the blockchain and its database. We also found out that we have a problem in the method verifyIban. This method updates the trust after verification of your IBAN, but this would mean that you can always update your trust by just verifying. We could solve this matter by

keeping up the information about a verified IBAN such that it will not change the trust value when you keep verifying.

5.3 Conclusion and discussion

5.3.1 Recapitulation

Albeit the API was a hard subject to create Interaction Design tests on, the process of interviewing the test subjects went smoothly and better than expected. We made sure all of our test subjects know about out disclosure before any tests took place. There were occasional questions about the working of the application but all of them were answered and understood quickly.

To us this means that the tests were arranged efficiently and effectively. Test subject number 3 even gave compliments on how well made the tests are so we do not doubt about the quality of our tests. The average is 3.9, this is higher than our standard and we are satisfied with such result. The API can be improved much more but only the most important changes will be applied to the current API as we have limited time and resources for this task.

6 Product Evaluation and Failure Analysis

The story of our final product knows a rocky history. The sheer difficulty combined with bad communication made for a hard course of development; in this part we will highlight several mistakes and failures we underwent in order to arrive at our final product.

6.1 Initial Godclasses

Starting the development with vague specifications we created two classes, BlockController and DatabaseHandler, that contained all our logic. BlockController's method were mostly empty proxies to DatabaseHandler's methods and almost no checks were done; it was possible to instantiate a Block with completely invalid properties - hash value, contact hash, previous hash - and then just insert it into the database. A lot of merge conflicts ensued as there was not enough code for five people. There also was an BlockFactory class with a purpose of creating different Block types; however there was only one type so that it was essentially a useless factory, inserted only to satisfy the condition of having a design pattern included in our program.

Another problem with this code was the large amount of work it took to create a correct blockchain: you had to set the hash value and all other properties manually. There were around seven such properties, and bad values were either not detected or surfaced later in irrelevant code.

6.2 Travis

An ongoing problem that lingered for the entire duration of the project was getting a continuous-integration system to work. At first there were configuration errors because our project was not directly placed in the root of our git folder, then there were timeouts as it took Travis too long to start an Android emulator. This forced us to submit and merge code that failed most of the times. Sometimes when lucky the code would succeed to build but this was not consistently reproducible.

6.3 Complex Blocks and Failing Unittests

When Blocks were finally improved half of the existing unittests failed. The new code allowed for easier block creation and did not allow blocks with placeholder values for its fields, however all unittests did exactly this. Since most of the tests did not test any actual functionality they were finally removed and replaced with better tests.

6.4 API Redesign

Another failure from our side was the early creation of an API. This API contained methods to add keys, revoke keys, and list keys of contacts and was well-tested, however it was not used. Later refactors caused it to give compiler errors and later failing unittests, so we made the decision to remove it, only to add another API several weeks later. This time the API allowed the retrieval of not only keys but also names and IBANs.

6.5 Block Subclasses

Still another deviation from the final design was the temporary addition of block subclasses: GenesisBlock, KeyRevokeBlock and KeyAddBlock. They gave a purpose to the BlockFactory and could partially verify their own integrity: the GenesisBlock for example checked that his contact hash was always "N/A". Outside from this they contained no functionality: a later refactoring of all block-related objects removed all those block subclasses.

6.6 GUI

For the first part of the project we were under the impression that we had to create a fully-fledged app with the possibility to exchange contacts via Bluetooth (which other groups would supply). We had created a GUI which supported the manual addition and revokal of contacts plus the viewing of those contacts. However, later on we learned that our main purpose was to build a library and that other groups would create the app with a GUI so that we would only need an extremely simple GUI to demonstrate our program.

While the removal of the GUI felt a bit wasteful it boosted our code coverage since the GUI was hard to test. Test code was written but was later removed since it returned null-pointer exceptions on the Travis server with later refactors.

6.7 Complete Architecture Redesign

When in the third-last week we decided on something that we should have done in the beginning: to create a complete design. Until then, our app had grown organically, which can look beautiful on cities but produces ill-fated code. Almost all our code was revised, which made the prior weeks seem a bit of a waste, but we also realized that we could not have done this from the beginning since we did not have the knowledge required for a solid software design. The redesign featured, among other things, a complete refactoring of the database handling, taking all power from the once-almighty God Classes.

6.8 New API Misuse

Looking back, despite the facts that security and integrity checks are now performed, our new API is still relatively easy to misuse as it operates on blocks, not on actions. The syntax allows you to for example submit a completely different block when changing the trust value of an existing block; this is only caught at runtime. The API also returns a list of blocks instead of a list of Users, forcing the end-user to write a routine to extract information from the blocks.

7 Outlook

While our current library supports the bare basics of a blockchain, it does not support some features which would make it truly revolutionary, such as peer-to-peer networking or proven trust calculations. If the library were to be used by the general public the following features would improve the relevance of the library.

7.1 Multiple Key Support

A limitation of our library is that it only supports one public key per person. Multiple keys per user would mean more security, since a user can invalidate one of keys with the other keys if that particular key is stolen. In order for this to work the multiple keys do need some kind of order indicating which key can revoke which key. With this system a practical user could for example keep three keys, applying the following scenario:

- The first, most powerful key he gives to a good friend in case he loses the other two keys
- The second key he keeps at his home hidden inside a safe
- The third, weakest key he keeps with him all the time on his USB stick for daily use

When his USB stick is stolen or when he loses his bag - not quite uncommon - he can simply use his spare key to generate a new daily key and proof that he himself generated it. When his house is flooded or burned down he can go to his friend to obtain the most powerful backup key to generate new keys for himself in his new house. When the friend loses his original key, the user cannot regenerate a more powerful key with his second key, however he can 'shift' the hierarchy one key down and use his second key as new most powerful key, generating new keys.

Of course other scenarios are possible such as a company giving keys to all its employees and storing a few backup keys on different locations, or a happy coders family sharing family keys. Backup servers could exist which store encrypted copies of private keys.

7.2 Speed Improvement

We have not tested our database with truly large amount of users and blocks, however as long as you do not transfer your entire multichain with the provided methods we do not believe any major speed issues will arrive. Should they arrive then it will be easy to add indices to our SQLite database to speed up query execution.

7.3 API Improvements

While our API received an unexpectedly high rating from our test subjects, there were still some improvements possible, mainly because the API operates on blocks, which directly represent a block in our database. The Block object, which is used in most API methods, exposes details of the inner workings of our library and this is not good practice. We should completely hide the Block object and instead let users interact only with the User object. This object provides all info that an end-user can possibly want without exposing the internal workings of the API.

7.4 Peer-to-peer Network

Currently our library supports only the direct import and export of entire chains for transfer and the manual addition of blocks so that the host application still playes an important role in keeping the trust values of people up-to-date. Nicer would be if the application could connect with an (existing) peer-to-peer network on which it can send and receive block updates in real-time. This allows for faster growing of the database since trusted on the other side of the world could be added in no time. The library could still support the manual verification via Bluetooth or IBAN as this would still mean a large trust boost.

7.5 Trust

Currently the application stores trust along with other block properties in the database. It could be calculated a lot better when we allow transactions to be recorded in the blockchain and then calculating the trust value using the algorithm described in the TrustChain paper. Another feature we could then add is the automatic revokal of users with bad trust, and with the upgrade described earlier to send it directly over the network to the other peers so they know not to trust a peer.

7.6 Non-functional Improvements

When making this library universal it would be a good idea to rewrite it to other languages such as C. Currently there are some Android-specific dependencies tying it to the Android platform, limiting the use in programs on the PC, reaching an even larger audience.

References

[1] Google Joshua Bloch. How to design a good api and why it matters. `http://zoomq.dn.qbox.me/ZQCollection/presentations/Howto-Design-a-Good-API_Why%20it%20Matters.pdf`. Accessed: 2016-06-22.

[2] Simon Read. Is paypal right to freeze customers' accounts? `http://www.independent.co.uk/money/spend-save/`

simon-read-is-paypal-right-to-freeze-customers-accounts-2360058.html, 2011. Accessed: 2016-06-22.

[3] VirginiaTech University. Analytic and empirical methods. http://courses.cs.vt.edu/~cs3724/summer2-03somervell/lectures/cs3724-emperical.pdf, 2003. Accessed: 2016-06-22.