Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ReusabiliToken: Encouraging Reuse Through Tokenization

BIOTS 2018 – Report

Authors:

Prashanth Chandran [a]	chandrap@student.ethz.ch
Tatu Lindroos [b]	tatul@student.ethz.ch
Tobias Rohner [c]	torohner@student.ethz.ch
Claudio Cannizzaro [d]	claudioc@student.ethz.ch
Silvano Cortesi [e]	cortesis@student.ethz.ch
Philipp Fischer [f]	fischphi@student.ethz.ch

ETH Zürich, 30.04.2018

[a]Wrote chapter 3
[b]Wrote chapter 1, contributed to chapter 2, chapter 6
[c]Wrote section 5.1 , chapter 6
[d]Wrote chapter 2, chapter 6
[e]Wrote section 5.2, section 5.3, chapter 7
[f]Wrote chapter 4, chapter 6

ABSTRACT

ETH Zürich
851-0591-01L BIOTS - Blockchain and Internet of Things
Written report, 18 pages
April 2018

This project report was written as a part of the spring 2018 course 'Blockchain And the Internet of Things (851-0591-01L)' run by M. Dapp, S. Klauser, and D. Helbing. We participated on FuturICT 2.0 challenge to implement a token obtainer with the Genesis framework. In two days, we developed two android applications using the provided framework to enable obtaining of tokens when a claim is verified (Silvano, Philipp, Tobias). We also thought of a use case concept (Claudio, Tatu) and made a simulation on how the economy could work if such a concept were realized (Prashanth). The software code which is part of this report is open source and available at `https://github.com/ETHBiots2018/ReusabiliToken`.

CONTENTS

1. INTRODUCTION

Blockchain is a potentially disrupting innovation that has gained an increasing amount of interest during the last ten years. Resources are being poured to start-ups and research initiatives that try to find possible applications in a wide variety of fields from finance and health sector to community organizing. The blockchain technology promises a decentralized solution to preserving information while maintaining certainty of past actions without the need of a trusted intermediary.

First appearing in a publication by Satashi Nakamoto in 2009 [1], the use of blockchains began with cryptocurrencies like Bitcoin and Ethereum [2] that utilize cryptographic hash functions to enable trustless networks to keep a digital ledger and conduct transactions between peers. However, in addition to finance, blockchains could also be used to invent new solutions to remarkable societal problems such water security, nonrenewable energy or lack of governmental transparency. During a BIOTS hackathon 2018, FuturICT 2.0 proposed a challenge to design a token system that would tackle a societal problem by incentivizing beneficial behavior. We chose to address the misuse of natural resources.

The wasteful consumption of natural resources is one of the factors that lead to increasing deforestation and loss of natural diversity globally. By encouraging reuse, we could provide savings in energy, water and materials needed to produce the appliances. Decreased production would also translate to reductions in greenhouse gas emissions and pollution. Our idea, that we will explore in this report, is to solve this problem by developing a token system that would incentivize reuse of containers such as coffee mugs at coffee shops. We use the words reuse and recycle in this report interchangeably to mean repeated use of an item.

In the next chapter, we will elaborate on our concept and present a flowchart to illustrate our token system. Chapter three explains and provides results of a Python simulation that was done to get a rough look on how viable our economic model is and which are the major factors we should focus on. Chapter four will then discuss the implementation of the concept on a general level. In the fifth chapter, we will give an outline of the android implementation we did, which is accompanied by the source code in the GitHub repository [3].

2. CONCEPTUAL MODEL

Our concept is that the customers would gain tokens from using reusable containers when buying consumables at stores. This would reduce the consumption of single-use containers that the store would usually have to provide. The key questions we should answer are what should the customers do with the tokens, and what motivates the stores to participate in this program? To address these questions, we invented a whole economic system which we will explain in this chapter.

2.1 Core Idea

To incentivize people to use reusable containers, there has to be a benefit for them in doing so. Thus, our idea is as follows: If a customer goes to a shop and uses a reusable container, they will get rewarded by gaining a token. In return, these tokens can then be used to claim a bonus at these shops.

But now there exists only a benefit for customers. To incentivize stores to join, we introduce a reputation system in which regular visits to a certain store gives a customer additional tokens. Thus, the perk of being a participating store is that they get regular customers, on top of saving the cost of buying disposable cups.

To counter a situation where a store would accept the reusable cups but not offer any bonuses in exchange to the tokens, we added a fee that the participating stores must pay. This fee could only be paid with tokens.

2.2 ReusabiliToken

The ReusabiliToken consists of two tokens: a value token and a reputation token. The value token is a standard ERC-20 token and thus freely tradable, whereas the reputation token is just a store dependent counter of keeping track of your reputation at a certain store.

2.3 Token Flow

The flow is illustrated in Figure 1. The ReusabiliToken is being generated by using a reusable container at a store. This is the only way the tokens can come into existence. The value tokens can freely be traded with other customers or third parties, but most importantly they can be spent at stores in exchange to bonuses or rewards. The stores then use the tokens they have traded from the customers to pay their participation fee. The fee must be paid in regular intervals e.g. once a month. We envisioned that the fee could be collected by smart contract which would then be unable to transfer them anywhere and the tokens would therefore be virtually destroyed, thus closing the cycle.

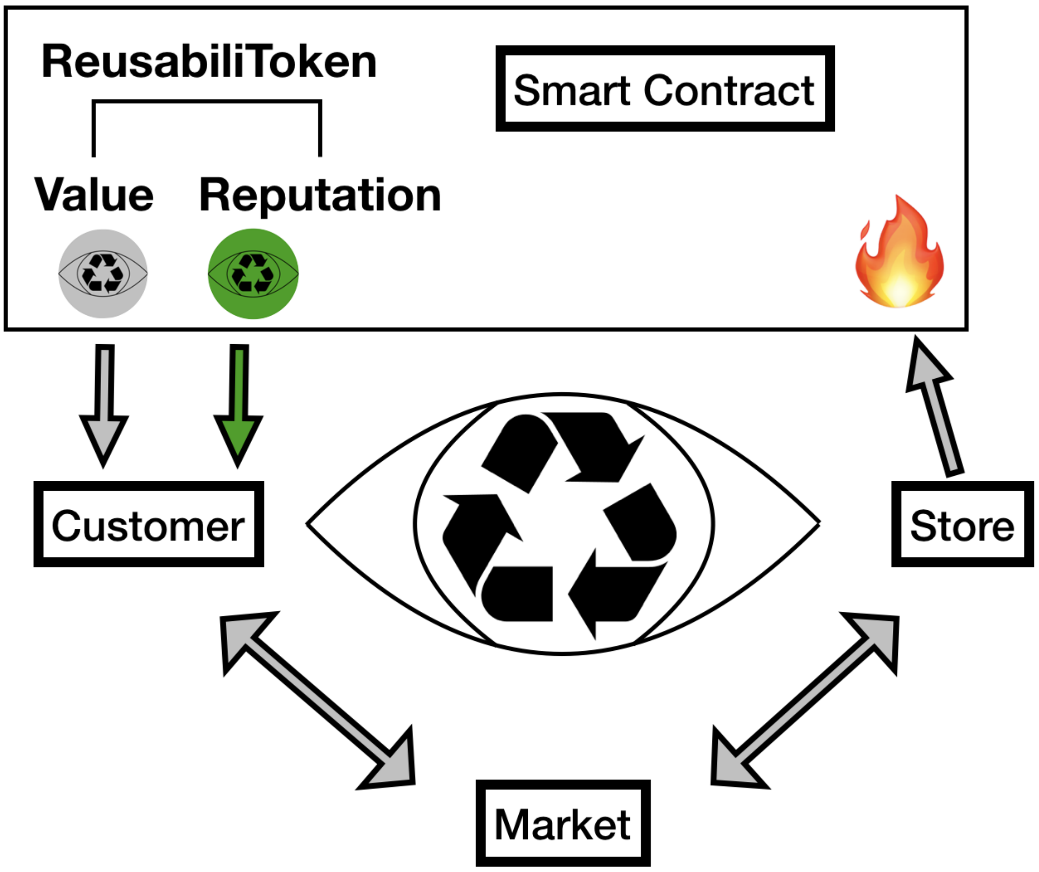


Figure 1: A concept schematic of the token flow as it is intended

2.4 Reputation

The Reputation is a store dependent token, which incentivizes customers to revisit a store time after time. This works as follows: The more reputation a customer has at a specific store, the more value tokens they get each time. Naturally, this increase in token gain should be capped. For the simulation purposes described in the next chapter, we introduce an upper bound to the reputation which is approached with diminishing returns i.e. a function like $M - \gamma e^{-\lambda k}$, where M is the upper bound, γ and λ are the factors which determine the diminishing returns exactly and k is the k$^{\text{th}}$ visit at a certain store. In addition, the reputation should decline over time to motivate the customers to keep visiting the store from time to time.

2.5 Fee

The fee is the mechanism which forces stores to offer some service in return to be paid in value tokens. That could be for example free coffee. We propose a system in which the cost is linearly dependent on the total amount of reputation tokens all customers of a certain store have. So that stores who profit more, must pay more.

3. SIMULATION

3.1 Simulator Core

To successfully simulate a token economy on the Blockchain, we need to understand the different players and components involved in the *ReusabiliToken* economy. In our token economy, the two key players are the customers and the shops that benefit from recycling and reputation tokens respectively. There is a smart contract that regulates the issue and collection of these tokens. The smart contract in turn interacts with an oracle to obtain information about certain facts from the real world. These basic components will be explained in the following sections.

3.1.1 Customer

In our system, a customer is a human who goes to a shop to buy a product. In doing so, a customer is presented with two choices on which they can decide.

- **Choice of shop**: In an open and competitive market, a customer is free to choose from many shops that are available to him/her. This choice depends on many factors. For example, some customers might prefer to go to a shop that is geographically close to them. There may also be customers who choose shops that are less expensive, while others might prefer quality and brand value over cost. Additionally, as customers continue to participate in our token economy, they may prefer going to shops where they have higher reputation because visiting such shops will be beneficial for them. Likewise, there could potentially exist many other factors that influences a customer's decision to buy at a specific shop and it is not hard to see that modelling all of them is quite challenging.
- **Choice of recycling**: After making the decision of buying at a specific shop, the customer has the option of recycling at the shop. Unlike the choice of shop where the customer had many different options to choose from, the decision to recycle is binary: i.e. a customer can either recycle during a given visit to a shop or not. This decision is influenced by a customer's willingness to participate in the token economy and their inclination to recycling.

In our simulator, we model three types of customers based on the differences in their behavior. We differentiate between customers based on how strong their preference of a shop is, and how likely they are to recycle every time they visit a shop.

3.1.1.1 Good customer

In our context, we define a good customer as someone who has a strong preference for a certain shop and always buys from there. A good customer also aggressively recycles every time they visit a shop. Therefore, a good customer can to maximize their reputation at a given store and also earn *ReusabiliTokens*.

- **Choice of shop**: A good customer has a shop randomly assigned as their preferred shop and always buys from the same store with a probability of 1.0.
- **Choice of recycling**: A good customer recycles 90% of time they visit a shop.

3.1.1.2 Bad customer

A bad customer is one who takes decisions randomly and almost never recycles. Consequently, a bad customer has a poor reputation at almost every store. We decided to place such customers in our economy to evaluate how a certain fraction of customers might behave in the real world and what their effect might be.

- **Choice of shop**: A bad customer randomly visits a shop at every iteration ("day") of the simulation. All shops in the market have an equal probability of being chosen by a bad customer.
- **Choice of recycling**: A bad customer recycles only 10% of the time that they purchase.

3.1.1.3 Neutral customer

A neutral customer is our baseline customer who neither acts idealistically nor like a cynic.

- **Choice of shop**: Every time a neutral customer wants to buy something, they randomly pick a shop from a small subset of all available shops in the market. All shops in a customer's preferred shop set have an equal likelihood of being chosen at every time step of the simulation.

  We introduce randomness in every customer's preferred shop subset, so that each customer has their own custom preferred shop list.
- **Choice of recycling**: A neutral customer recycles with a probability of 0.5 i.e. they choose to recycle half the time, and choose not to for the rest of the time.

Out of N total customers in our market, we need to define a distribution of different customer behaviors. We arbitrarily choose a distribution of 20% good customers, 20% bad customers and 60% neutral customers. This is however a tune-able parameter and our simulator can be run with any arbitrary customer distribution.

3.1.2 Shops

In our token economy, shops are the places where the smart contract is triggered. Every time a customer recycles at a given shop, the shop contacts the smart contract to issue reputation and reusability tokens to the customer. We can arbitrarily specify the number of shops in our simulator. We used it to analyze the effect of competition on our token economy.

3.1.3 Oracles

An Oracle is an integral part of a token economy. It is a way for the smart contract to gather facts from the real world. In our simulator, we have one such oracle, the shop oracle.

3.1.3.1 Shop Oracle

As mentioned in section 3.1.2, a shop is an entity that triggers a smart contract every time a customer recycles. This means that anybody on the block chain could trigger the smart contract and ask for it to transfer reputations and reusability tokens. This means shops/customers can trick the smart contract into falsely issuing tokens to suit their needs.

To avoid this, every time our smart contract is triggered by an external actor, the smart contract validates the sender's address with a shop oracle. The shop oracle is expected to host a list of valid shop addresses on the block chain to protect the smart contract from fraudulent transactions.

3.1.4 Smart Contract

Our simulator also consists of a fully functional smart contract that regulates the issue of tokens and collects fees from shops.

Token issue based on the law of diminishing returns: To keep a hold on the issue of tokens, our smart contract follows the rule of diminishing returns. This means that beyond a certain reputation, customers always receive the same number of reusability tokens every time they purchase with a recycle.

Diminishing reputation: The smart contract also houses mechanisms to erode a customer's reputation at a specific shop if they do not visit the shop in a very long time. This duration is configurable in our simulator and prevents customers from holding on previously built large reputations and encourages them stay active in purchasing and recycling.

3.2 A Typical Simulation Cycle

In this section, a sequence of interactions that constitutes a cycle (iteration or "day") in our simulation framework is discussed. These events are in chronological order.

- Every customer independently chooses a shop to buy things from. This choice is based on the individual customer's buying habits.

- Every customer independently chooses whether or not to recycle on that particular day of purchase.
- A customer visits the chosen shop and chooses their mode of payment.
  - **Pay with cash/card**: The usual transaction
  - **Pay with *ReusabiliTokens***: If the customer has enough ReusabiliTokens, they can also choose to buy products with it.
- If the customer had chosen to recycle that day, they make a claim to the smart contract about the recycle.
- The shop validates the customer's claim.
- The smart contract contacts the shop oracle to verify if the claim and its proof are valid.
- At the same time, the smart contract verifies if the concerned shop has paid its due properly. If not, no tokens are issued and the transaction is closed.
- If the claim and the proof were valid, and if the shop had properly paid its dues to the smart contract, the smart contract transfers appropriate amounts of reputation and reusability tokens.
- At the end of every simulation cycle, shops check if their payment to the smart contract is due and make payments if necessary.
- The entire sequence repeats on day two.

3.2.1 Hyperparameters

We used our simulator to gain insights on some of the following hyper-parameters.

- **Optimal fee and payment cycle**: What is the right amount of fee that should be levied on shops and how often should shops make their payments?
- **Reputation limit**: What is a reasonable threshold to have on the maximum reputation a customer can earn?
- **Coins (tokens) to issue**: What is a reasonable number of coins that can be issued for every transaction?

3.3 Visualizations and Discussion

A typical visualization that we generate is shown in figure 2, explained below.

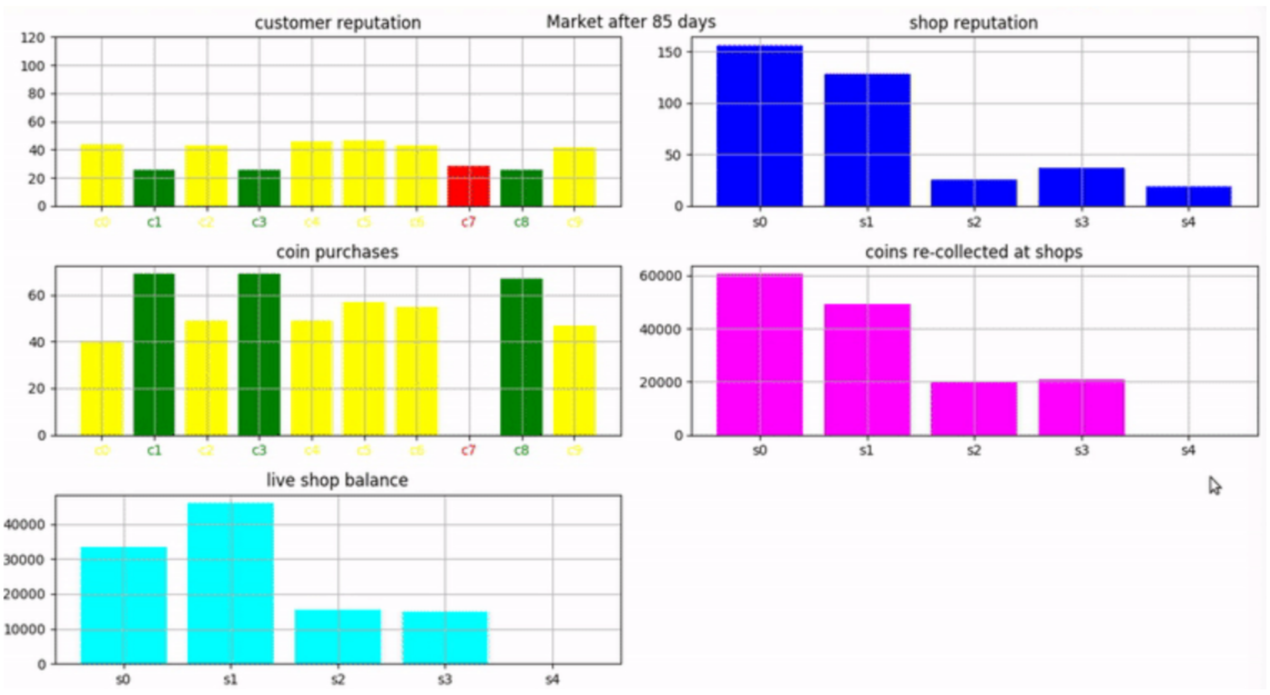


Figure 2: A snapshot of our simulator in action.

3.3.1 Customer reputation plot

On the top left, we track customer reputation i.e. the total reputation that a customer has across all stores that they shop from. Each bar corresponds to an individual customer: bars shown in green correspond to those of good customers. Bars in yellow correspond to those of neutral customers and those in red represent bad customers. We could expect that customers with high loyalty and strong recycling tendencies earn high reputations but in this simulation run, the neutral customers earned the highest reputations.

3.3.1.1 Shop reputation plot

On the top right, we track the total reputation of all customers that buy products at a specific shop. Recollect that the fee that a specific shop is charged is proportional to this quantity. Amongst other things, a high value in this graph would indicate that such a shop is promoting its customers to recycle with good incentives. This in-turn promotes loyalty and more recycling.

3.3.1.2 Coin purchases

On the left half of the second row, we track the total number of times that a customer has brought goods with *ReusabiliTokens*. As before, green, yellow and red bars represent good, neutral and bad customers respectively. A large value here indicates that a specific customer is actively using their tokens to purchase products. When multiple customers across the entire economy have high values for the number of coin purchases, it indicates a healthy system where consumers are being rewarded for their recycling, while shops are benefiting from increased customer loyalty.

3.3.1.3 Coins re-collected at shops

On the right half of the second row, we track the total number of tokens that were used at a given shop for purchasing goods. A high value here indicates that customers are being rewarded well by this shop for spending their ReusabiliTokens. This in-turn encourages more recycling and loyalty to such shops.

3.3.1.4 Live shop balance

Finally, in the last column we track the coins that are available at a shop after it has paid its dues. A positive amount indicates that the shop is still active in the economy. As can be seen, Shop 4 has zero coins recollected from the customers, could not pay its fees and is not active at the end of the simulation run.

3.3.2 Evaluating a simulation

We consider a simulation run with a given configuration to be a success when at the end of the simulation, there are still shops that are active in our token economy. This is because it is possible that shops can drop out of the economy in situations where customers do not spend enough tokens at this shop, and the shop will eventually run out of tokens to pay to the smart contract.

4. SOLUTION

The idea of the implementation is to have users of our system verify certain actions at a trusted source in such a way that they can then claim their reward. A source should be identified as trusted through the verification of a digital signature.

4.1 User Frontend

A user such as a customer in a shop must be able to send a claim of an action they completed. The action is proven by a certified instance such as the cashier and therefore there needs to be a secure channel between the user and the one sending the proof. To make sure the reward gets sent to the right address the proof should also contain the user address, which is why this information should be sent to the verifying instance in the first place. The smart contract can then receive claims that contain addresses proven to have concluded a certain action.

4.2 Trusted Source Frontend

The trusted source must, in the first place, verify that it is trusted. It therefore contains a private key which it can use to sign information. The user, such as a customer in a store, sends its address via a secure channel to the cashier and they sign the address with their key if they approve that a certain action has taken place.

4.3 Smart Contract

The smart contract must contain all the information about the tokens. It has functions that regulate how many tokens are generated, for which action, and when tokens are destroyed again. If an action has been verified by a trusted source, the smart contract can check this in a database that belongs to the blockchain and is therefore secure. A sent claim must also contain an id that can be used by the smart contract to find the public key that belongs to the instance that verified the action. After checking the signature, the smart contract can give the programmed amount of tokens to the address.

This leads us to the problem of how the smart contract can verify a trusted source. A source is only considered trusted if its id and public key is stored in the database. This database has therefore to be filled by an oracle. In the use case of shops, we were thinking of either a legal office that registers the stores to act as a secure provider of this information, or to have the smart contract fill in the database that queries a secure register of all enlisted stores via an oracle.

4.4 Benefits

This system might at first glance look like there is no real benefit to it since we still need an oracle to manage a database. However, we find it much easier to have one oracle that can verify other users. This allows for a system where we can make arbitrary users such as cashiers to trusted sources. The users we want to incentivize to do something can select a claim and have it verified by our certified instances which removes the need for any complicated checks to verify an action. The system is thus easily expandable by other intended actions and their claims, and can be applied to any sector where trusted instances can be chosen to verify them.

5. SOLUTION DESIGN (ANDROID)

This chapter first describes our extensions to the FuturICT2 Genesis library [3]. Afterwards, the realization of the protocol defined in chapter 4 in two Android apps is discussed. One app is developed for the customer (section 5.2) and one for the trusted source (section 5.3).

5.1 Genesis Library

FuturICT2 provided us with a comprehensive framework with its Genesis library. Our task was to extend it to also include our use case.

Our project consists of two different parts. One runs on the smartphones of store employees and customers, and the other runs on the blockchain. To assist fast development during the hackathon, both parts are implemented in Java and run on Android smartphones. The interface to the parts, eventually situated in a smart contract on the blockchain, is designed in such a way that the migration from Java to Solidity or a similar language should require no changes. All code that should eventually run on the blockchain but does not so yet is prefixed by `Dummy` to provide easy distinction from code that runs on a smartphone even in the finished project.

This chapter first provides an insight into the mechanics of the smart contract, followed by an explanation of the client-side aspects of the library.

5.1.1 Smart Contract

The smart contract situated in the blockchain handles the generation and transfer of tokens. In our extension to the Genesis library, the smart contract is represented by the `DummyRepo` class implementing the `IRepository` interface. The intent of the `Repository` class was to only keep a list of existing tokens and let the tokens keep track of account balances. As all dummy classes only have to emulate the smart contract on the blockchain as a whole and no single class instance will ever be used alone, we decided to implement the account balance tracking for our value and reputation tokens directly in the repository class instead of creating two tokens keeping track of their own balances. This approach is not very nice and defeats the purpose of the `IToken` interface, but it was fast to implement and it works for demonstration purposes.

As a consequence, the `DummyValueToken` and `DummyReputationToken` classes are not very complex, but require a member variable that points to the `Repository` the token is located in to be able to process transactions.

5.1.2 Client-Side

On the client-side, the most important object is an action. To provide an implementation of our primary idea that is as generally applicable as possible, we created an abstract base class `AHumanConfirmableAction` that can be used to implement any action which can be confirmed by a human or other trusted source of confirmation. An instance of such an action contains a short description of what has to be done to earn some small reward. The amount of tokens rewarded on successful completion of the action and the address of the individual claiming the action is also stored inside this class. Furthermore, a `store_id` is provided. This ID uniquely identifies an instance that is accepted to confirm the specific action. A public map, likely located on the blockchain, exists and maps these store IDs to public RSA keys that are used to sign a proof for the completion of an action. This is necessary to guarantee the validity of proofs provided to the smart contract. If no such database would exist, a proof could easily be faked.

To demonstrate our intentions for the class `AHumanConfirmableAction`, we implemented two example actions that extend it: `BringOwnCup` and `BringOwnPlate`. The first one provides some tokens as a reward if someone brings their own cup to a coffee shop instead of using a disposable one. The second action gives a reward for bringing a plate to a food stand and reducing plastic waste that way. The only thing both actions do is to define the description and reward for a completion of the action. They also override the `getType` method required by the `IAction` interface.

When a user claims an action, they have to deliver a proof that they actually completed said action. In our case, this proof is in the form of a confirmation of some trusted entity. To prohibit the claim of actions with false proofs, every confirmation of an action must be signed with a DSA keypair where the public key is stored in a publicly accessible database likely located on the blockchain. The data that is signed is the hash of the action some customer wants to claim. This prevents him from obtaining the signature and later changing the type of action to maximize their reward while not invalidating the signature. Furthermore, this hash can be reconstructed in the smart contract on the blockchain such that no or at least very little hash collisions can be exploited. A proof for some action is represented with an instance of the `HumanConfirmableActionProof` class. This class only has one data member containing the signature of the action to claim.

The class `StoreDatabase` acts as an emulation of the public database containing a mapping of IDs to public DSA keys. It is implemented as a singleton to provide access to one single global database. As it is still local to the process that constructed it, the customer and cashier apps have to initialize two equivalent versions of the `StoreDatabase` at start-up. This will not be necessary once the database is located on the blockchain, however then it will need additional features such as the guarantee to be synchronized with the data stored locally on phones at any time. A problem still to be tackled is that only trusted sources can enlist in the public database, as everyone could just sign their own claims otherwise.

5.2 Customer App

The customer app consists of four activities:

The **Main Activity** allows the customer to enter their Ether-Address. The customer can also scan their address using the implemented QR-code reader, which can be accessed via the "SCAN ADDRESS AS QR-CODE" button. By touching the "SAVE ADDRESS" button, the entered address can be saved such that the last entered address will automatically be loaded the next time the app is being restarted or reopened. Before an address is stored, we check if it matches the format of a blockchain address and if it is invalid, an error is displayed via `Toast`[4] messages. Correct addresses are stored with the standard Android Studio class `SharedPreferences` and the Activity changes to `ClaimActivity`.

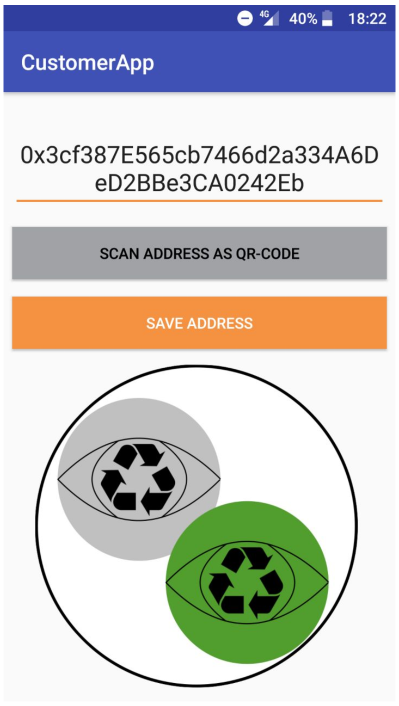


Figure 3: Login Customer App

The **Claim Activity** puts and shows all actions contained in an object of type `HumanConfirmableOperation` in a `ListView` (dynamically created). At the moment, there are two implemented actions (our applications could be easily extended by setting and defining new actions in the Cashier App):

- Bring your own cup to a coffee shop
- Bring your own plate instead of using a disposable one

By touching an action, it can be claimed. A change to the `QRCodeActivity` takes place.

The **QR Code Activity** will then display a `QRCode`, containing information about the action to be claimed and about the `Customer Address`. Now the verification by a cashier takes place. They can scan the QR code and verify the action which produces a signed QR code that the customer can use as claim and send to the blockchain. Until now, the request is not submitted to the blockchain, but by a further improvement of the Genesis-Library, this can easily be implemented. By clicking on the "SCAN CASHIER'S APP TO GET PROOF" button, the `QRReader` opens, and the proof in form of a QR code can be scanned. The app will then return to the `MainActivity`, where a new claim can then be scanned.

The **Barcode-Capture Activity** is the responsible activity for scanning a QR code. It is an implementation of the QR Code Reader described in the tutorial of "Varvet" [5], rewritten from Kotlin to Java. It uses Google Play services and the Mobile Vision API[6], which allows to scan barcodes quickly and locally on the device.

5.3 Cashier App

The cashier app consists of four activities:

The **Main Activity** allows the cashier to log in or register. A new account is created by pressing the button "SIGN IN". If the cashier logs in using an existing account, they enter their ID and click on "LOG IN". The system then switches to the linked activity.

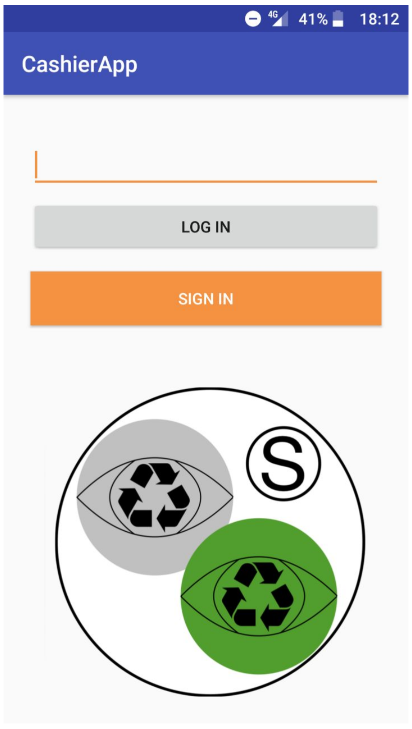


Figure 4: Login Cashier App

The **Sign In Activity** (`StoreSignIn`) allows the store to create a new account. They enter their ID and name, and a DSA key pair is generated for them. The ID and public key are stored in a database (not yet implemented). For debugging reasons, the hashed public and private keys are displayed. The keys are provided by the Java library `security` [7] and its classes `KeyPair` and `KeyPairGenerator`. The interface to the database provides the self-implemented class `StoreDatabase`, which then also allows access to the token. A touch of the "SIGN IN" button switches to the `SignedStore` Activity. If an invalid name or ID is entered, a `Toast` notification indicating the error will appear.

The **SignedStore Activity**, which is invoked after a successful log-in, can sign a claim. By clicking on the "SCAN CLAIM" button the `BarcodeCaptureActivity` already described in chapter 5.2 is started and a claim in the form of a OR code can be scanned. After a successful scan, the claim can then be confirmed if the cashier agrees. A QR code is then displayed, which is the confirmation for the customer and acts as a claim that can be submitted to the blockchain.

To ensure that only stores can verify actions, the customer claim is signed with a digital signature (`DSA`)[8]. Since the blockchain holds an already mentioned database with IDs and public keys of all registered stores, it can easily verify that the claim has been approved by a trusted source, namely the cashier. The encoding of a claim looks like this: "Store_ID Digital_Signature PublicKey". The strings are joined together and separated by a space. For additional error minimization, some of the scanned values are checked, and in case of invalidity, an error message in the form of a `Toast` notification is displayed.

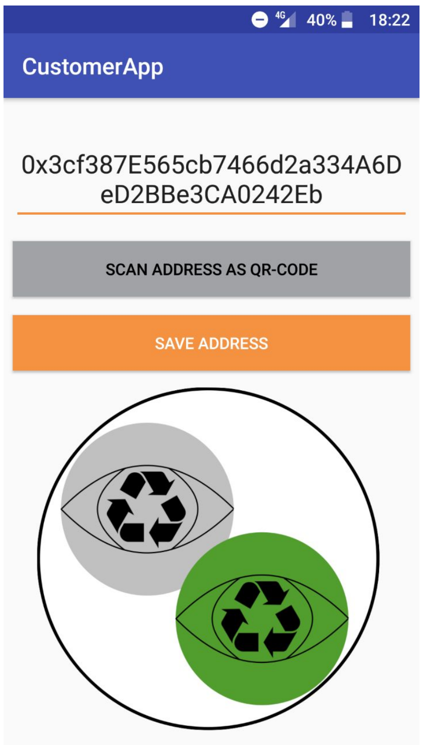
6. EVALUATION AND CONCLUSION

We believe a microeconomic system such as the one proposed could be a useful tool in reduction of plastic waste as suggested by the simulations ran and discussed in chapter 3. Our solution is easy to use and has a benefit of only uploading trustworthy data to the blockchain. Another plus is that our solution is not dependent on extra hardware and can be installed right away on existing devices. At the moment, our prototype is working and new actions to claim tokens can easily be added. However, there are still many improvements to be made. On the conceptual side, all the parameters we propose in our model have to be determined exactly by experimentation or simulation and finally tested thoroughly in the blockchain. We were not able to finish the whole program in the limited amount of time we had during the hackathon. Thus, the reputation token is not yet implemented. The current code should also be overhauled to a maintainable form and parts of it do not use or implement the API provided by the Genesis library as intended.

To provide all the necessary functionality for our project to be tested and used, most importantly a smart contract needs to be deployed on the blockchain. In the spirit of the Genesis library it has to be able to receive claims from anyone and verify them by a look up of the corresponding public key in an attached database. Furthermore, it needs to manage this database and store information about trusted sources such as the store IDs and their public keys that is given to it by an oracle. The smart contract also has to manage the tokens and distribute them to whomever submitted a valid claim. To extend our project further, we suggest adding additional claims with a meaningful classification. The customer app could then be updated to allow for multiple actions to be claimed at once. Also, to reduce the risk of exploitation, additional stages of security could be added.

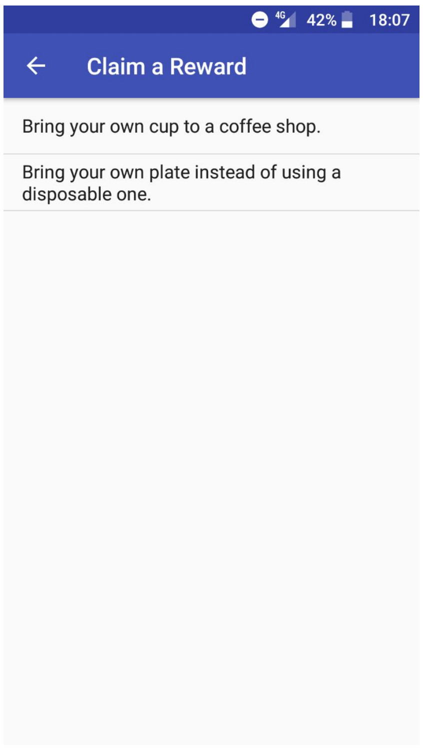
7. ILLUSTRATIONS

7.1 Customer App



0x3cf387E565cb7466d2a334A6DeD2BBe3CA0242Eb

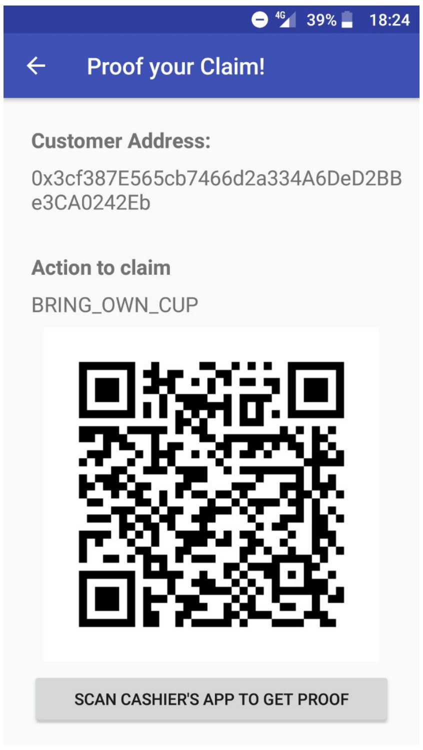
Figure 5: Main Activity



Bring your own cup to a coffee shop.

Bring your own plate instead of using a disposable one.

Figure 6: Claim Activity



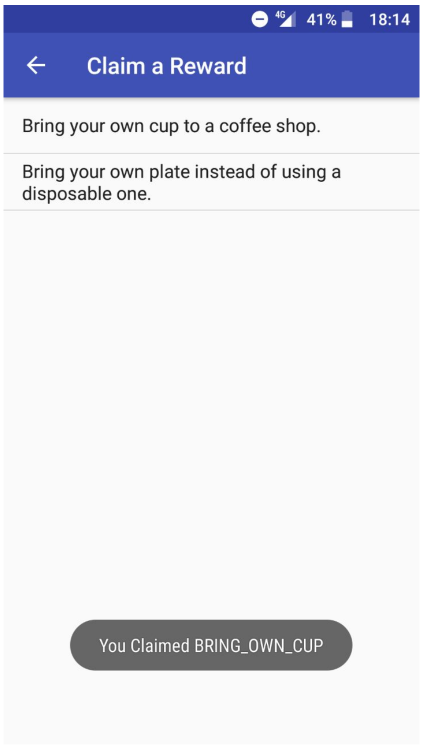
Customer Address:

0x3cf387E565cb7466d2a334A6DeD2BBe3CA0242Eb

Action to claim

BRING_OWN_CUP

Figure 7: QR Code Activity



Bring your own cup to a coffee shop.

Bring your own plate instead of using a disposable one.

You Claimed BRING_OWN_CUP

Figure 8: Return to Claim Activity, after successful proof

7.2 Cashier App

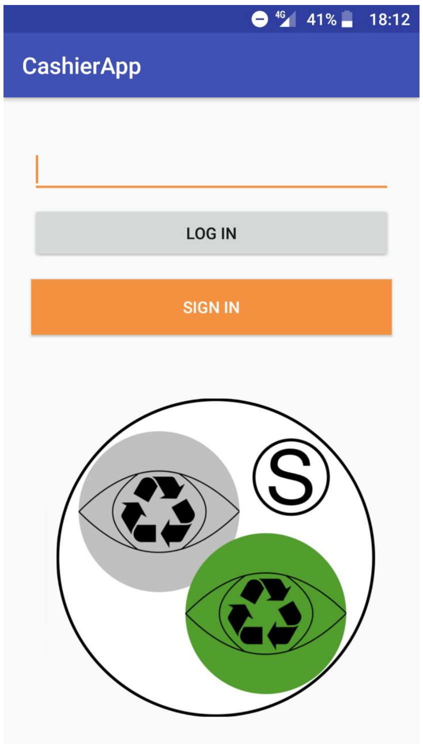
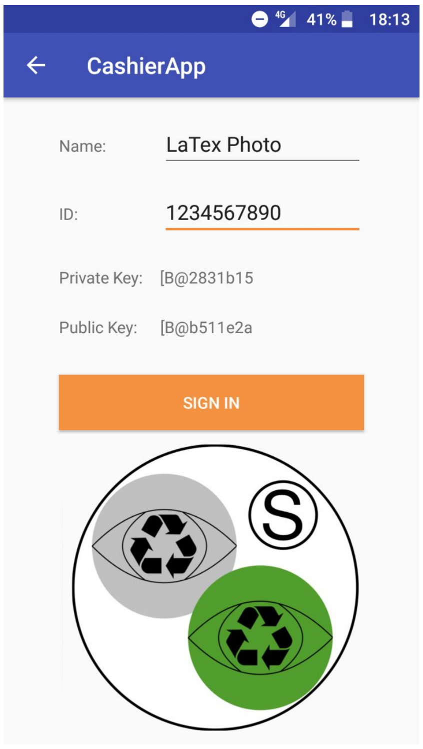


Figure 9: Log In Screen



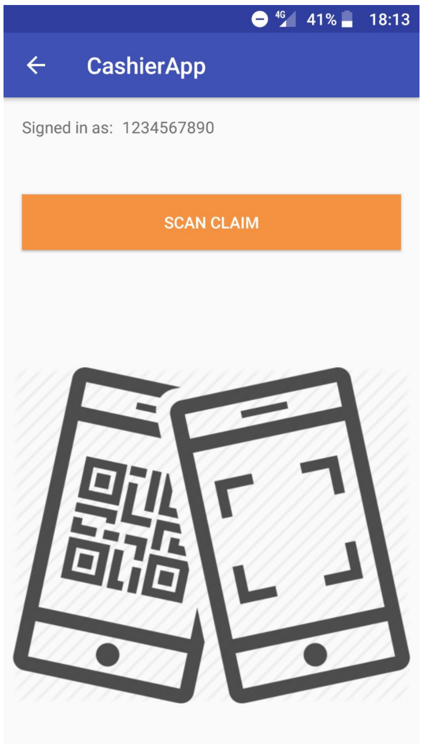
Name: LaTex Photo

ID: 1234567890

Private Key: [B@2831b15

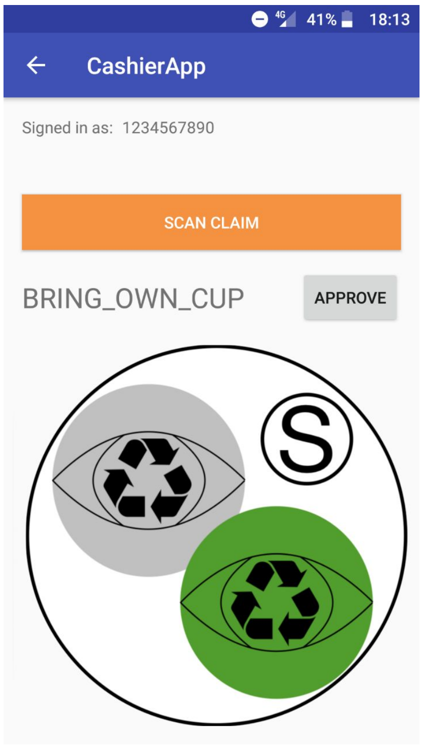
Public Key: [B@b511e2a

Figure 10: Sign In Screen



Signed in as: 1234567890

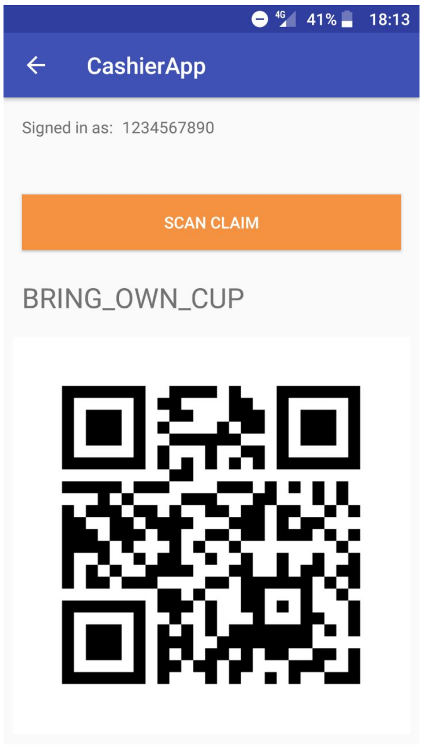
Figure 11: Scan Claim



Signed in as: 1234567890

BRING_OWN_CUP

Figure 12: Approve Scanned Claim



Signed in as: 1234567890

BRING_OWN_CUP

Figure 13: Claim Approved

REFERENCES

[1] Satoshi Nakamoto. Bitcoin white paper. `https://bitcoin.org/bitcoin.pdf`, 2009.

[2] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 2013.

[3] FuturICT2. Genesis. `https://github.com/FuturICT2/Genesis`, 2018.

[4] Google Inc. Toasts. `https://developer.android.com/guide/topics/ui/notifiers/toasts.html`, 2018.

[5] Daniell Algar. Android qr code reader made easy. `https://www.varvet.com/blog/android-qr-code-reader-made-easy/`, 2016.

[6] Google Inc. Mobile vision. `https://developers.google.com/vision/`, 2018.

[7] Oracle America Inc. Package java.security. `https://docs.oracle.com/javase/8/docs/api/java/security/package-summary.html`, 2018.

[8] David W Kravitz. Digital signature algorithm, July 27 1993. US Patent 5,231,668.