CS 6675 – Advanced Internet Computing Systems

Course Project, Pascal Wissmann

PiTrust – A blockchain-based trust network

Abstract

In a world of global connections between people, most of them having never seen each other, it is hard to gain (but often easy to lose) trust. While centralized trust databases are already in place and serve well for particular purposes such as eBay or Uber, these are still designed for very particular purposes and are fully controlled by a central party while not being independently verifiable by any peer.

In this project, I propose a system to quantify the trustworthiness or expertise of an entity in a particular topic. The system will be based on blockchain technology in order to keep an non-revokable ledger on ratings and therefore expertise of the entity, which shall facilitate honesty across all participating parties and provide a high level of reliability and availability of the system. In order to achieve those goals, I will leverage the capabilities of an existing blockchain, which will be enhanced by a so-called smart contract.

This project is based on my submission for assignment M4 [1] and will therefore by nature be congruent to a certain extent. The source code of this project is available on GitHub.

Introduction

To retain a very particular use-case rather than a generic proposal for a "trust network", I consider the following real-world problem: A new employee joins a company of multiple thousands of colleagues. He does not know anybody except his direct colleagues yet, but has a very specific question, for example about export regulations for a particular product of this company. The "classic" way will be to ask his direct colleague, who may know somebody, who knows somebody, who has an idea about that topic. However, the knowledge of this colleague may be outdated, or he may even have left the company at that time. Additionally, that particular colleague may not provide the optimal answer since, even though he will certainly have some knowledge about that topic, others maybe know better about it. Furthermore, asking oneself through a large organization step-by-step may be a tough and time-consuming challenge. This scenario may then also easily be scaled to global public expertise network.

A centralized capability database may partially solve that issue, but these often come with a main issue: The ratings are either based on a self-estimation or on judgement of the employee's direct supervisor. Both will again lead to poor ratings with at most "local optimums".

So, the idea is that employees - or more generally: peers - rate each other's knowledge rather than being restricted to self or supervisory estimation. Furthermore, the system shall not be based on a static "1-to-5 stars" rating for expertise domains, where people will tend to overestimate their own or others knowledge[1]. Therefore, I chose a *transaction*-based system, in which a peer can rate the knowledge of another peer based on a particular action, for example answering a question or otherwise supporting on a specific topic. As opposed to a pure *scoring*-based system, this will allow more advanced techniques for data mining and competency evaluation as the *reasoning* behind an actual score can in the future be evaluated with more sophisticated algorithms than the one that is developed in this project. This may then also include advanced machine learning algorithms or correlations to other sources such as social media, which are not considered in the basic implementation. However, this is what is commonly called *off-chain analysis* and not considered to be part of this project.

This paper will begin with an initial need-finding which includes research on prior work, as well as the evaluation of the basic building blocks of existing work compared to the preliminary planning for this project. This will then be followed by a proposal for the baseline design, which will also include a brief explanation of the prototype implementation for the required smart contract and insights on the actual design process and a first evaluation of the performance metrics for the implementation. In the next chapter "Redesign through Refinements", I will propose major changes to the very first prototype, which aim on improving the resiliency and performance of the smart contract in daily usage. The actual effectiveness of the implementation will be planned and executed in the next two chapters by modelling and simulating a group of users using the system. The project whitepaper will finally be concluded with some additional remarks on what I learned throughout the project and how the PiTrust system could further be extended for future and more advanced use cases.

Initial Need-finding Analysis and Preliminary Evaluation

As briefly described in the previous chapter, the main goal of this work is to propose a system, which objectively consolidates ratings of peers about other peer's knowledge of a particular domain and enable to find experts registered to the network.

Extended research in the cryptocurrency space revealed that with Braintrust [2], a token with a comparable use-case, connecting freelancers to employers, already exists. However, in comparison to the proposed PiTrust token, the profiles are again based on self-estimation while the entire ecosystem is more focussed on the matchmaking, settlement, and monetarization of the liaised assignments while this work more focus on building and storing an objective metric for competencies. As the system proposed shall not only contain a globally acknowledged trust-score for a particular peer (and knowledge-domain), but also the reason for that score.

In principle, this may be implemented in a central database and that may indeed by sufficient for a permissioned use-case such as a company-internal knowledge database. However, this would also be prone to misuse and/or compromise, for example by malicious administrators, and is not applicable to another, more broad use-case, where the knowledge network shall be traceable and evidential for everybody. Furthermore, a central solution will not scale very well with a large number of users and would require additional (decentral) infrastructure.

[1] This effect is certainly heavily influenced by rating systems such as Amazon, where everything below 5 stars (the absolute maximum) is considered to be a complaint.

A decentralized system, such as EigenTrust [3] may generally serve the purpose of a permissionless scenario, i.e., anybody may join and use the system without explicit registration, but it does not allow any traceback, *why* the score of a particular user is especially low or high as any trust-building or destroying action is immediately reflected in the peer's trust score while the reason will not be retained by the network. Also, other work proposed such as [4] and [5] elaborate on how to calculate a trust score, but do not refer to retaining the reasoning behind the score. Retaining a ledger of transactions, which cannot be changed anymore after their commitment, however, is the core functionality of a blockchain such as Bitcoin [6] or Ethereum [7]. The choice of the blockchain to be used will be a major part of the Baseline Design.

To consider the implementation to be successful, five aspects have to be sufficiently accommodated, in descending order of importance:

- Smart contract security,
- Rating value,
- Transaction performance,
- Usability,
- PiToken *tokenomics*

In fact, the smart contract security must be considered of the highest importance, as a potential hack on it may render either, all ratings as well as the token itself, completely worthless. However, this will not be the centre of this homework as the security analysis of a smart contract would very well exceed the scope of this work.

The rating value, i.e., the objective correctness of the calculated ratings, however, depicts a major part of the design evaluation as this is a cornerstone of the actual idea, even relatively independent of the question, if it is a centralized, decentralized or blockchain based implementation.

Third, the transaction performance takes an important part of the baseline design and later evaluation as there are huge differences in the use of different blockchains as well as how the smart contract itself will be implemented. This will also be a major task within the refinement chapter.

The usability of a blockchain based solution stands and falls with the integration on any form of wallet to be used. As an easiest approach, the user interface is based on classic username/password websites, where the credentials are used to compute the necessary cryptographic keys for the blockchain interface. However, this would a) make password changes complicated and b) not be compliant with the clear goal to have a *decentralized* ledger of transactions. Furthermore, many people will be wanting to keep their cryptographic keys on a software or even hardware wallet such as Ledger or Trezor making such an approach infeasible. So, the approach will be to make use of a well-known software wallets such as Metamask[2], which also allows the usage of additional hardware wallets for more security-demanding users while retaining a simple interface for the "broad masses".

The last factor of success for the implementation of a blockchain based system are the *tokenomics* of the underlying utility token. In a nutshell, this is a mixture of decisions on the token's total supply, how it will be initially distributed across different parties (founders, community fund, public sale etc.) and how the count of tokens will evaluate over time by

[2] https://metamask.io/

creation/mining of additional tokens or *burn* of tokens. In addition to these changes on the total supply, the concept will also comprise an idea, how the tokens shall actually be used, e.g., as a reward for writing honest feedback and, of course, as a reward for providing a high level of knowledge to the users on the PiTrust platform, so that it will be adapted not only as a trading good like other crypto tokens, but as something which can also be *used*.

Baseline Design Method and Measurement Results

The baseline to use a smart contract for implementing PiTrust was already made because of the requirement for decentral and secure storage of the rating ledger. A purely transaction-storing blockchain, such as Bitcoin, will not be sufficient for the implementation, which shall not only comprise the transaction (i.e., ratings), but also enable calculations out of these, which will discussed in-depth at a later stage. Starting with the blockchain to be used seems a little bit “upside-down” but doing so was simply necessary to not get lost between all options and to focus on one development platform as this will be the first smart contract to be implemented by myself. So, I will start with comparison of several smart contract enabled blockchains:

	Transactions/s	Gas fees	SC language
Ethereum	25	18$	Solidity, Vyper
Binance Smart Chain	160	0,33$	Solidity, Vyper
Cardano	250	0,50$	Marlowe, Plutus
Algorand	1000	0,00$	TEAL (Python)
Terra	10000	0,01$	Rust
Solana	50000	0,00$	Rust, C, C++
Polygon	65000	0,01$	Solidity, Vyper
Waves	100	0,00$	Ride
EOS	4000	0,00$	C++
Tezos	40	0,01$	Michelson

Most numbers for Transactions/s taken from [8], all other sources are consolidated in the extended reference list as they would totally blow up the regular reference list. Even though it looks as if Ethereum is the worst decision to program a token for, I deliberately chose to do so. The reasons are the following:

- Ethereum is by far the most anticipated blockchain, right behind Bitcoin, which does not enable any smart contract implementation, therefore the platform is proven and established
- Due to the same reason, the community support for programming smart contracts for Ethereum is by far larger as any other community, making finding help much easier
- Smart contracts written for Ethereum can be migrated easily to work on the BSC or Polygon as well

While I am personally more used to program in Python, which would encourage to use either Vyper or TEAL for programming the required smart contract, I chose to rather use Solidity as the community using Solidity is of magnitudes larger than any other smart contract programming language, leading to much more available documentation including a full-blown IDE called Remix[3]

[3] https://remix-project.org/

In order to ease the very first iteration of the implementation, each *block* of the blockchain only consists of exactly one transaction. For a real-world implementation, one would surely include more transactions into one block to reduce the required overhead and network load. The basic design of storage of the transactions is depicted in Figure 1 with Transaction 0 being the “genesis transaction” which did not contain any relevant transaction. Storing this information on the (Ethereum) blockchain is the maybe easiest smart contract to be implemented and serves as the starting point for the solution’s implementation.

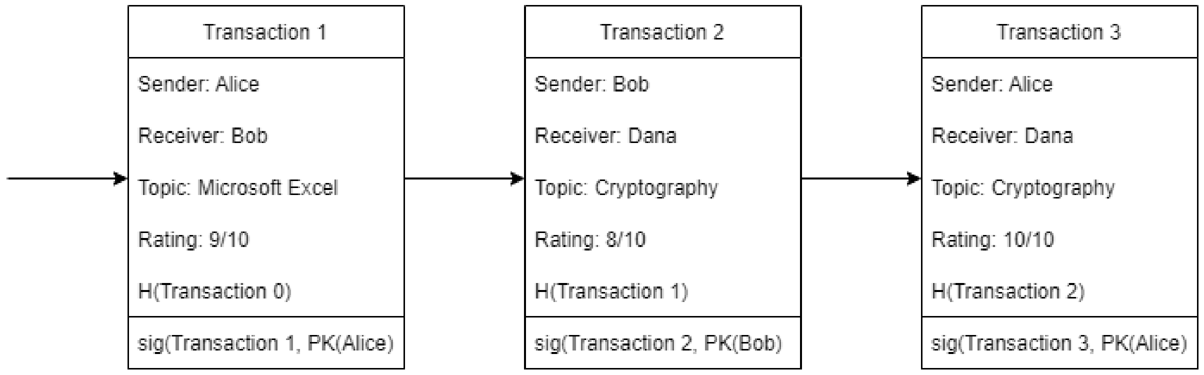


Figure 1: Basic transaction design

Since the PiTrust smart contract is written based on an existing blockchain, it does not even have to deal with the actual signatures and chaining, but can just save the actual rating (Sender, Receiver, Topic, Rating). Or more precise: An updated array of all ratings. Brief additional research revealed that this was also evaluated prior to this work, even though the scope was completely different. [9] This however will lead to an excess cost of storing the entire array after each transaction of $13.82/KB already 1 ½ years ago [10] so that the programmatically easy solution had to be abandoned and going back to the initial idea of storing only the *one* actual transaction/rating on the blockchain and “rolling them up” every now and then to reflect the actual rating of a person’s knowledge based on past transactions as it was also proposed in the prior assignment. [10] However, in fact the above-mentioned cost prohibits to store any more data on the blockchain as needed at all. Therefore, the rollup-mechanism as proposed in the prior assignment must be revised and beside storing the actual rating, the new rating will be immediately calculated and stored on the blockchain.

As a first estimation, the competency of a user may be calculated by his average rating divided by the overall average rating in that respective area. This can easily be calculated by looking up the current rating of the user r_{old}, weighting his score by the ratings he got so far (n), adding the scaled new rating r and divide the result by n+1:

$$r_{new} = \frac{(r_{old} \times n) + r}{n + 1}$$

New users (and fields) will be added to the ledger upon their first occurrence. This may lead to newly joined users to join the system likely with a maximum score of 10/10, which will not be sustainable for a longer period as this very first rating may also be a self-estimation of the skill (where the “sender” and “receiver” will be the same person). However, this behaviour will be traceable on the blockchain ledger, so there will be an, at least, social incentive for honest behaviour.

The baseline design is still incomplete for two reasons, which may not be entirely important in a more or less trusted environment such as a corporate network, but render it unusable in an untrusted, public, environment:

1. The system may be exploited by creating fake accounts to boost (or downvote) an account.
2. Even if issue 1 is solved, the system is still prone to honest but incorrect up- (or down-) rating people without professional reasoning (as stated above; someone who can create a pivot table is not an Excel guru).

Even though newly joined user's ratings do barely contribute to the actual score of another user, just flooding the system with new users and use them to rate (positively or negatively) a particular user may still lead to a significant undesirable effect. To prevent this, the *PiTrust token* comes into play. The token acts as kind of currency to *pay* to be able to rate another user. So, the primary idea is, that new users/accounts start without any balance in their wallets and need to earn tokens by *getting* rated (or *buy* them). By granting r-1 tokens to a user receiving a rating r (out of 10 points) and burning 1 token[4] of the user's wallet upon his vote. With that approach, it becomes impossible to create sustainable circles of ratings, especially not for *down*rating. Depending on the actual (fiat) price of the token this would make exploiting the system quite costly. Furthermore, this behavior may easily be discovered by analyzing the blockchain and my then be penalized, either socially or algorithmically.

However, why shall anybody be willing to *pay* for their own contribution? In a permissioned system, every user could receive a fixed number of tokes each month (for example 10), each being equivalent to one vote. In a permissionless system however, this will not work as this would even incentivize creating lots of fake accounts to gather "voting power" faster.

In order to prevent "hoarding" of tokens, there shall be a decay implemented, removing a token (including fractions) 1 month after it has been transferred to the wallet, except for those which have been initially bought for fiat money. As an incentive for not letting the gained tokens void but rather spend them for ratings, the number of ratings done by a particular wallet address may additionally be tracked and rewarded by, for example, using it as an additional factor for incoming votes and not only granting the initially proposed $\frac{r-1}{10}$ tokens, an additional $\frac{(r-1)\ x\ \#votes}{\#total\ votes}$ so that using the actual voting power will act as some kind of long-term investment, especially for highly capable experts. Furthermore, it is imaginable to prevent the token to be traded back into fiat currency (or other cryptocurrencies) as it is *designed* to be spent and not as an investment object. However, due to the nature of cryptocurrencies, this approach will likely fail and making buying the token completely uninteresting for any user.

During development it turned out that it is not possible to keep track on-chain, when a token was redeemed due to it fungible nature. However, the decay is very easy to implement on a block chain by a very simple and well-known mechanism: inflation. Even though a stable or even deflationary supply is usually desirable for a cryptocurrency this is not the case for the *PiTrust* token. This implemented by minting an additional 20% of tokens to the so-called funding wallet every year. The function *inflation(),* which checks, if a year is over will simply be called upon every registered rating. This in fact means that the inflation will not be continuous, but marks big steps in the total supply (Figure 2) and may also not happen *exactly* one year after the last step (or the initial deployment), but rather on the next transaction *after* a year has passed by.

[4] The common wording for transferring tokens/coins to a non-usable address, i.e., where the private key is unknown. In the initial implementation, it will be sent to a "funding wallet" owned by the smart contracts creator.

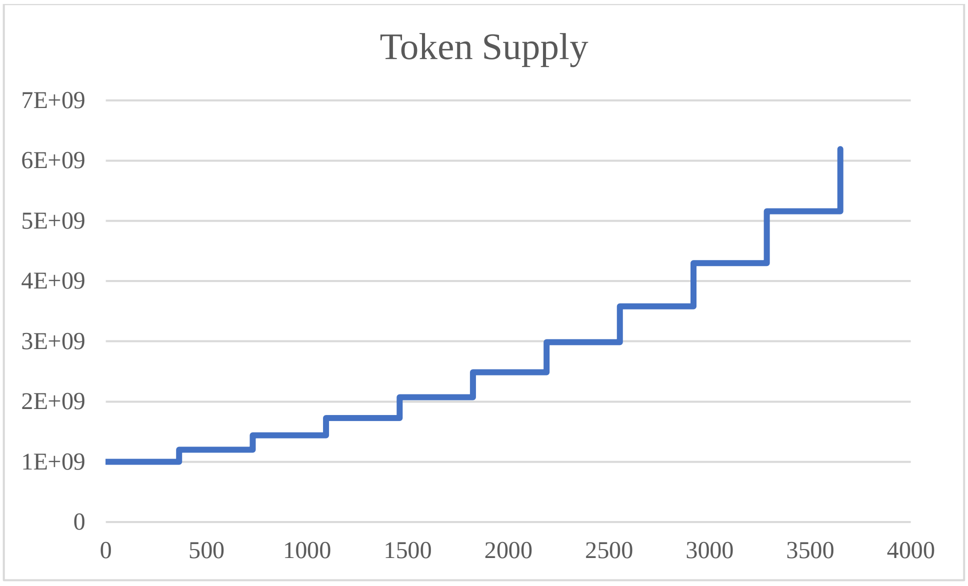


Figure 2: Token Supply for PiTrust token over time (days)

After having implemented the smart contract as described, this will actually be the solution as it was initially proposed in assignment M4, excluding what was meant to be an "additional consideration" (see next chapter) and a user interface. However, the main challenge of this project's part was not the refinement of the actual proposal but rather its real implementation, which came with its own challenges as I personally did not have any prior knowledge of how to program smart contracts resulting in a very steep, but interesting, learning curve.

Redesign through Refinements: ~3 pages

Refinement to ERC20 token

While the basic implementation works quite well in a local simulation, it is lacking one major property: It is not yet adhering to the most important standard for smart contracts on the Ethereum blockchain, ERC-20 [11]. Refactoring the token to be compliant to that standard makes it a) tradeable, b) verifiable and c) migratable to other blockchains. Therefore, the following functions and events have been properly implemented (Source: [11]):

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
function approve(address _spender, uint256 _value) public returns (bool success)
function allowance(address _owner, address _spender) public view returns (uint256 remaining)
event Transfer(address indexed _from, address indexed _to, uint256 _value)
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

Most of the functions are quite self-explanatory (or can be looked up in the above-mentioned source) while events can be considered as callback functions, which can be subscribed to by any internal and external program (e.g., a blockchain explorer) to be notified upon the particular event. The most notable semantic change to the code has been the implementation of the *approve* and *allowance* functions, which are designed to limit the impact of a smart contracts behaviour on an entities wallet. In this case, the approval was important to be given to the *PiTrust* smart contract for actual usage of the tokens as well as the possibility to grant this access to cryptocurrency exchanges such as Uniswap[5] to enable the onramping with the token, i.e., enabling people to trade other tokens such as USDT for PiTrust tokens as well as enabling any interaction with standardized off-chain systems such as Web3 frontends.

In order to maintain compliance to the standard, even for functions which will not be explicitly implemented, a well-known framework called OpenZeppelin[6] is available and will be used. Furthermore, it will support the development of a basic frontend at a later stage. This also boiled down the code of the very first prototype from 113 lines to 86 while providing all necessary interfaces and additional security features according to the OpenZeppelin website.

Optimization on rating weights

However, the basic implementation will not solve the issue of the "average" case where many people upvote mediocre knowledge. One way to solve this issue will be to weight people's vote either on their overall voting behaviour or based on their own rating on a particular topic assuming that a person who is rated high by a person who himself has a good rating on a topic (i.e., is an expert) will be more trustworthy than a person who is appreciated by persons without any knowledge on the respective area. The formular out of the original homework

$$R_A = \frac{1}{11(n+1)} \sum\nolimits_B R_A^B (R_B + 1)$$

with R_A being the rating of person A, R_A^B being the rating of person B for person A and n being the total number of persons, who voted within a particular area of knowledge at all, shall be applied. *n* is the count of *previous* votes. The rating R_B needs to be actually set to $\max\,(R_B, 1)$ so that also votes of users without knowledge in an area or who have newly joined the system (i.e., having an own rating of zero), are also considered, even though heavily scaled down compared to already known experts. This approach is comparable to the EigenTrust system [3] even though in the current implementation, weights are assigned statically and will not reflect increasing or decreasing weights of voters over time reflecting their expertise to change over time.

The formula however cannot be calculated in a closed form, because given a time *t*, the information about any ratings R_A^B prior to t are not available anymore. Additionally, it turned out that the programming language solidity does not allow any floating-point operations. Therefore, I chose to scale up all ratings by a constant factor of 10000, which shall be a sufficiently high precision (5 digits behind the comma). The final formular to be applied on any rating event is the following:

$$R_A^t = \frac{(R_A^{t-1} \times n_A) + (R_A^B \times R_B)}{n_A + R_B}$$

with a R_A^{t-1} being the rating of an entity prior to the newest vote. The overall design is depicted in Figure 3. It must furthermore be mentioned that the processing of one transaction does not correspond to a block on the Ethereum/Polygon blockchain, but the *results* of the process or, more likely, multiple subsequent processes will be stored in one block.

[5] https://uniswap.org/
[6] https://openzeppelin.com/

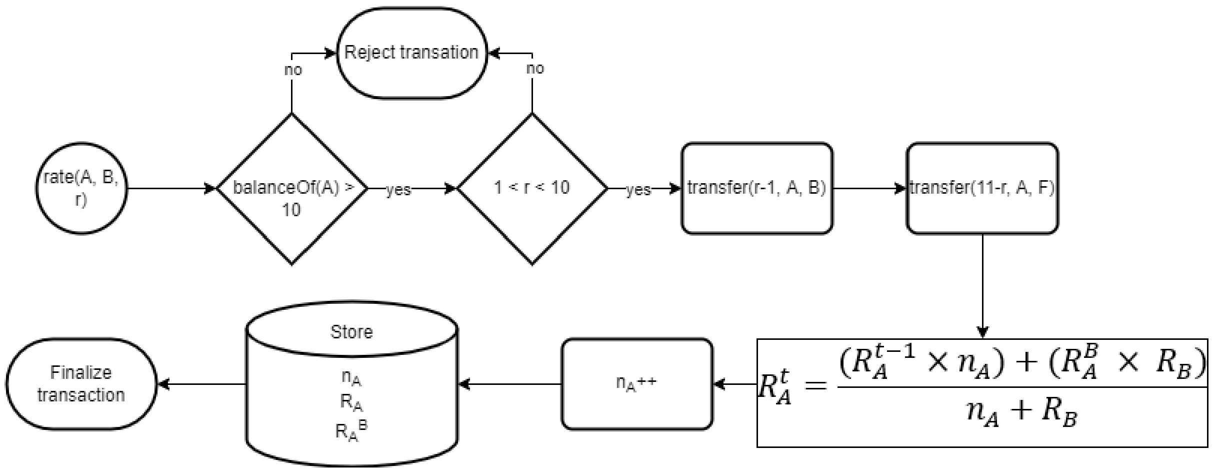


Figure 3: Process of a rating transaction

Frontend Development

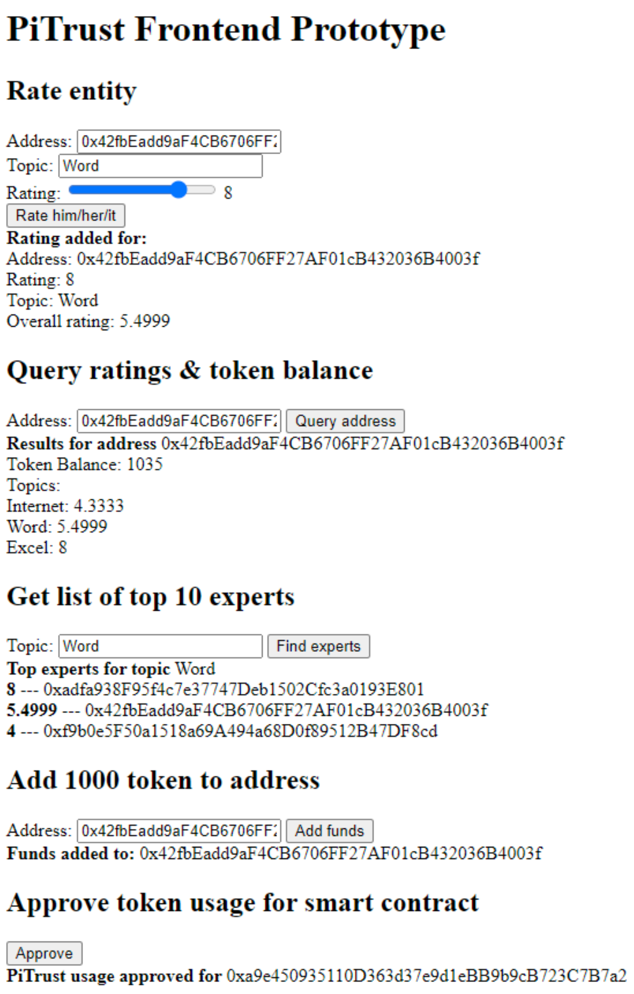
Furthermore, the system is yet lacking a frontend. While this project's focus lies more on the smart contract functionality, at least a basic frontend which is able to interact with the smart contract will be required. This includes the following tasks:

- Create a rating for a known address and a particular field of expertise
- Query the current ratings (all fields) of a known address
- Get the top 10 experts for a given field of expertise
- Query the balance of a wallet address
- Add 1000 PiTrust tokens to a given address (for test purposes only, will be removed)

As frontend development is not my center of expertise and even does not make sense due to the manifold options of existing frontend frameworks, which are well-established, more secure, and appealing than anything which a programmer could code from scratch and at the end spare lots of time. Most of them, like Infura[7], Moralis[8] or Alchemy[9] offer free plans to use their framework and web services for prototyping also making the provisioning or rent an Ethereum/Polygon node obsolete. However, they do not allow local prototyping using the Ganache local blockchain, therefore I proceeded with the Truffle framework using the "react boilerplate", which is basically a barebone of code to be used for a React application which shall communicate with an Ethereum smart contract. The frontend is based on the proposals of [12].

After deploying the smart contract to the local blockchain looks as shown in Figure 4. The interface is indeed very basic but is sufficient to interact with the smart contract manually and prove the functionality as well as challenges when interacting with a blockchain rather than a centralized database in the last chapter.

[7] https://infura.io/
[8] https://moralis.io/
[9] https://www.alchemy.com/


PiTrust Frontend Prototype

Rate entity

Address: 0x42fbEadd9aF4CB6706FF2

Topic: Word

Rating: 8

Rate him/her/it

Rating added for:
Address: 0x42fbEadd9aF4CB6706FF27AF01cB432036B4003f
Rating: 8
Topic: Word
Overall rating: 5.4999

Query ratings & token balance

Address: 0x42fbEadd9aF4CB6706FF2 Query address

Results for address 0x42fbEadd9aF4CB6706FF27AF01cB432036B4003f
Token Balance: 1035
Topics:
Internet: 4.3333
Word: 5.4999
Excel: 8

Get list of top 10 experts

Topic: Word Find experts

Top experts for topic Word
8 --- 0xadfa938F95f4c7e37747Deb1502Cfc3a0193E801
5.4999 --- 0x42fbEadd9aF4CB6706FF27AF01cB432036B4003f
4 --- 0xf9b0e5F50a1518a69A494a68D0f89512B47DF8cd

Add 1000 token to address

Address: 0x42fbEadd9aF4CB6706FF2 Add funds

Funds added to: 0x42fbEadd9aF4CB6706FF27AF01cB432036B4003f

Approve token usage for smart contract

Approve

PiTrust usage approved for 0xa9e450935110D363d37e9d1eBB9b9cB723C7B7a2

Figure 4: Frontend prototype screenshot

Evaluation Plan

The evaluation plan consists of two main goals. First, the system, including especially the PiTrust token, shall be proven to be usable in a stable way in a local simulation. Stable means that: The rating formulars must ensure a consistent rating of entities between 1 and 10 under all circumstances, i.e., ratings below 1 and higher than 10 may not occur regardless how many voting power (i.e., PiTrust tokens) an adversary expends. While the first part of the test will be purely qualitative, i.e., trying out different scenarios by hand, the second step will take a deeper look into the metrics of execution time, storage requirements, the above-mentioned consistency and resiliency against adversary groups such as consistently up- or downvoting single entities. Furthermore, it will have to be proven that there are no loopholes to generate PiTrust tokens out of thin air, but that the circulating supply, not considering the balance of the funding wallet, will be consistently declining without a significant agglomeration on particular accounts (except the funding wallet, due to deliberate inflation). The simulations will be leveraged by using the Unit testing functionality of the Remix IDE, which provides a vast functionality in that regard, which is comparable to other (commercial) products such as PyCharm for Python developments.

Secondly, the rating smart contract as well as the token smart contract shall ultimately be deployed on a real-world blockchain such as Ethereum and prove its usability for a few tens or hundreds of transactions without exaggerating execution times and/or gas fees. Since the latter are extremely high on the native Ethereum blockchain, I decided, after having tested the deployment on the Ethereum Testnet, to deploy the smart contracts on the Polygon blockchain[10] which features the same functionality as the required "Ethereum Virtual Machine" (EVM) to run the smart contracts with much lower gas fees to be paid in "MATIC" tokens. The processes on a public blockchain such as regularly signing transactions will not allow for automated evaluation. Furthermore, the main metrics and goals of this test are rather qualitative:

- The tokenomics must work as desired (inflationary, incentivizing, secure).
- Ratings shall be possible for a reasonable gas fee (to be paid in MATIC tokens)
- Either entering ratings as well as getting them shall be possible over a, very basic, web interface, which interacts with the smart contract/s
- The smart contracts need to be deployed on a public blockchain (Testnet)
- The token shall become tradeable on a well-known exchange such as Uniswap or, in case of Polygon, QuickSwap[11]. Even though the token may likely not be *listed* on the exchange, i.e., the address will have to be entered by hand.

While the Remix IDE was sufficient to perform basic tests like ensuring the ability to compile the code and check for correct behavior upon function execution, already the very first test is not possible in an efficient way. By doing some research and including a hint out of the Ed Discussion[12], I chose to implement the frontend connection using the Truffle framework[13] including its virtual blockchain Ganache, which enables also a deep analysis of the blockchain behaviour without actually deploying the smart contract & token to a blockchain even though it partly fails in simulating the real, varying, gas fees.

The first three success factors can be tested offline using the above-mentioned Truffle framework, but the latter two will require actual deployments on the Polygon Testnet, i.e., a real-world blockchain where smart contracts can be tested without any real-world costs as the required coins, such as ETH or MATIC can be created out of thin air in those networks. The deployment of the smart contracts on a **production** blockchain (Mainnet) will be evaluated as the final validation of the implementation. The single tests will be performed as follows:

For tests 2 & 3, I will generate 5000 transactions from 100 addresses pointing at the same 100 target addresses[14]. 80 of the addresses will act randomly while 10 addresses will forcefully try to downvote a particular address and other 10 will try to upvote one of the group members. Each address will start with a variable number of tokens. One test will be performed with 100 and 1000 tokens distributed to each user. In this test case, the "Topic" will be fixed to be the same for all ratings. For each of the 5000 transactions, the eligible voter will be chosen randomly out of the list of valid addresses while it is made sure that at least every 100 transactions either an "upvoter" or "downvoter" will be voting.

[10] https://polygon.technology/
[11] https://quickswap.exchange/
[12] https://edstem.org/us/courses/16525/discussion/1357860
[13] https://trufflesuite.com/
[14] Initially, I intended to perform 10000 transactions and also a case with 10000 initial tokens, but in the current deployment, this simply takes too much time to be practical (~1s per transaction) and the difference between 1000 and 10000 start tokens was negligible as accounts do barely run out of tokens also in the 1000-token case.

Web Interface

The evaluation of the web interface will be only qualitatively:

- Will it be responsive?
- Does it respond *correctly* to any of the test cases?

The test will be concluded with some ideas of further development.

Tokenomics

The evaluation metric will be quantitative, looking on the distribution of overall ratings as well as the distribution of tokens, which shall have decreased in total over the time (as the previously announced inflation will have no impact) as well as no significant accumulation shall be noted.

Gas Fees

The gas fees of to be paid for the transactions shall be near-to constant and stay at a low level for each execution. The fees will furthermore be compared with other well-known smart contracts such as Quickswap swaps.

Blockchain deployment (Testnet)

This test will be successful if the deployment on the Testnet was successful and one transaction using the web interface could be made. Either the deployment as well as the blockchain transaction will be proven as the transactions are stored on the Polygon Testnet blockchain and can be verified there.

Tradability

Just as the deployment itself, also the tradability will be proven on-chain. In this case, the smart contract/token's address will be imported manually on the Quickswap Testnet interface and then be traded against MATIC tokens. To do so, a liquidity pool will have to be founded as well, which will be explained in the execution chapter.

Blockchain deployment (Mainnet)

As a last proof of applicability, the token will be deployed on the Polygon mainnet. However, neither liquidity pooling nor any trade will be performed there due to the high potential of losing real cryptocurrencies and therefore money at this stage.

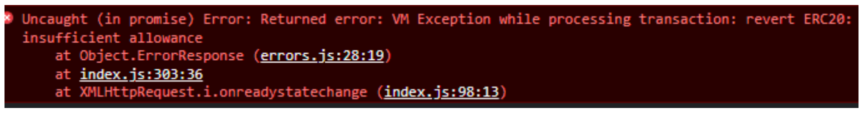
Evaluation Execution and Results

As most of the tests have been done qualitatively as part of the iterative development process, this chapter will also to a certain extent contain the optimizations, which have been made during development.

Web Interface

While creating and basic evaluation of the smart contract itself was quite a journey as Solidity was a new programming language to me, the very good documentation made that part rather easy. Things became more complicated when trying to connect a frontend to it as I was neither proficient in JavaScript nor the Web3 package to be used. One stumblestone I came across was the necessity to *authorize* the spending of PiTrust tokens in the smart contract prior to do (Figure 5), which lead to the last button shown in **Error! Reference source not found.**. The final interface actually consists out of 5 files:

- index.html - contains the actual HTML code and references to the other files
- main.css - contains some basic style information (mainly for rating the slider)
- frontend.js – contains all JavaScript to react on button clicks and show results
- backend.js – contains all JavaScript to actually interact with the smart contract



```
Uncaught (in promise) Error: Returned error: VM Exception while processing transaction: revert ERC20:
insufficient allowance
    at Object.ErrorResponse (errors.js:28:19)
    at index.js:303:36
    at XMLHttpRequest.i.onreadystatechange (index.js:98:13)
```

Figure 5: Error message for unauthorized token usage

Tests have been done manually. As the site is very barebone, it responds instantaneously on all actions. However, it turned out that is not possible (or very expensive in terms of gas fees) to store searchable multidimensional tables (expert, topic, rating) in a smart contract. Therefore, on the one hand the Solidity code had to be refactored to contain distinct tables storing active users-ids (addresses), active topics and two distinct mappings of users->topics and topics->users. The “Top Expert” query also must iterate over all users which have been found for a topic in order to sort the list. This visibly slows down the page update after initiating the search and may be improved in future iterations. Furthermore, the frontend design is far from being state of the art. The final prototype design can be viewed in Figure 4 in the previous chapter.

Integrating the functionality in a more modern framework such as React[15], with a clear and responsive corporate design, will greatly increase – or frankly said just introduce – any user acceptance for the system and therefore plays a big role for a real-world application.

Tokenomics

While the basic development and evaluation, i.e., if the contract follows the pattern of the design, was performed manually. The big-scale evaluation of the system had to be automized. Therefore, I developed a small test-script using the Truffle framework, which automatically sends the transactions as described in the last chapter to the blockchain and stores the token distribution and ratings of all users in a CSV-file. In fact, the first test already revealed an issue in the formula used to calculate the rating: As Solidity is not capable to perform floating point operations, the calculation had to adapted in the code even though the theoretical formular as priorly proposed did not change. Having adapted the code, I repeated the test and received the results as shown in Figure 6. It has been cut off after 2000 iterations because all accounts except number 6 (the target for upvoting) have been drained at that time. Even though the target was to prevent accumulation of tokens at one account, it is clearly visible that in all test cases, the account chosen to be collectively “upvoted” by 10% of the network sooner or later possesses nearly all tokens. Thinking back, this is also obvious because every token given to one of the “upvoters” will be sooner or later relayed to the chosen beneficiary. The effect is much smaller given a higher number of initial tokens as visible in Figure 7. There, account number 1 (chosen as beneficiary in that test) slowly accumulates tokens as

[15] https://reactjs.org/

well, but at a much slower rate, because all accounts will have enough tokens to spend to rate other beneficiaries as well and are not drained that fast. However, the trend is already visible in the chart.On the other hand, the target of a deflationary token circulation was achieved as visible in all figures.

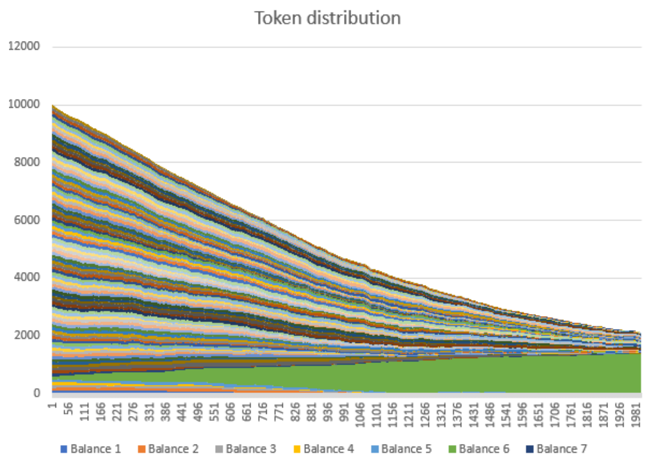


Figure 6: Token distribution over time (100 initial tokens)

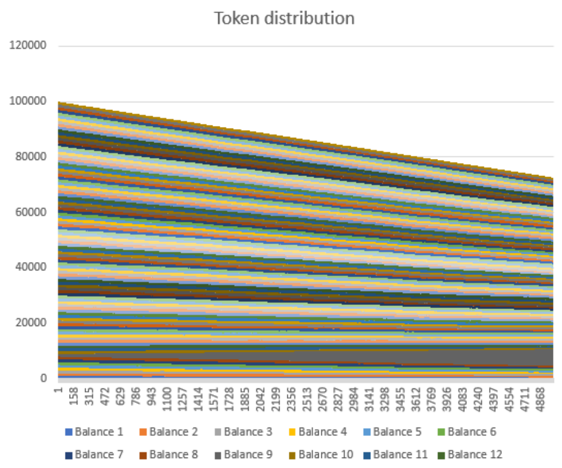


Figure 7: Token distribution over time (1000 initial tokens)

It has to be admitted that the accumulation of tokens at the one chosen account is not according to the protocol's goal, but on the other hand it was upvoted by <u>**10%**</u> of the entire network, which is quite significant. It also has to be remarked that the accumulation at this one account is basically according to the principle as being an incentive for positive votes. In a real-world scenario, either the percentage of the network voting in favor of exactly one account will be much smaller as well as that negative votes on his/her/its knowledge by other parties will likely exceed the artificial acknowledgement of the minor adversary group.

Gas Fees

During the test of the tokenomics, also the gas fees for transactions have been monitored. At the time of writing a one unit of "gas" costs around 100 Gwei (= 0.0000001 ETH) which is around $0.0003 at the time of writing[16]. That is the reason why I chose to rather deploy the contract to the Polygon network, where the gas amount to be paid is roughly the same, but the base unit, MATIC instead of ETH, is only worth roughly $1.50 instead of $3000 at the time of writing, therefore lowering the transaction costs by 200 times. Picking two exemplary voter's accounts, number 10 and 11, it is visible, that different calls of the contract caused different costs. Actually, there are three distinguishable cases:

- The first rating done by the account at all
- The rating done by the rater is the first for the target account
- All subsequent ratings

[16] For current exchange rates, see for example: https://coinmarketcap.com/

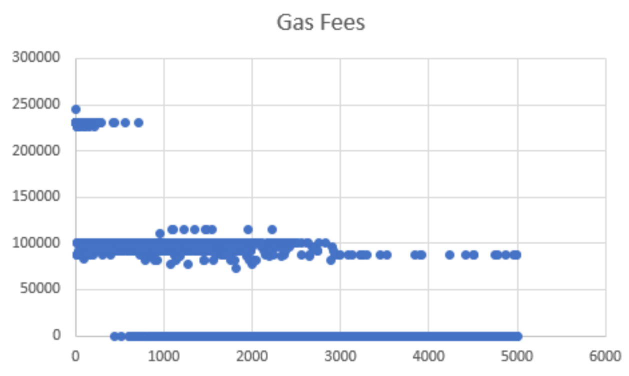


Figure 8: Gas Fees for PiToken transactions

A quantitative analysis of the costs shows that, except the very first transaction of an account, all transactions come at a cost of roughly 100,000 Gwei summing up to 0,01 ETH/MATIC. Only the first rating of a (new) user requires the visible increased cost of roughly 240,000 Gwei, but this does not matter much on the long run.

The fourth *special* case, which was not mentioned before, are ratings which have been declined (reverted) due to lack of token in the account's wallet, which result in 0 gas fees. Gas fees may always vary slightly by some few Gwei due to the complex calculations in the contract, but it is very important to mention, that that do not increase over time and the contract & token may therefore sustainably used on a blockchain. This is mainly achieved by not always writing back entire arrays to the blockchain, but only the information (i.e., ratings), which has *changed* due to the last transaction.

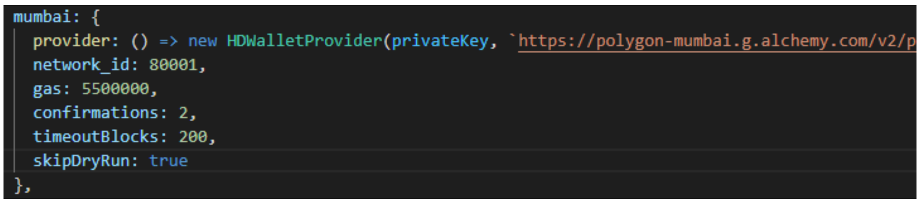
While the costs of 0.01 ETH would equal round about $30 on the Ethereum blockchain, making the system effectively unusable, it is only $0.15 equivalent on the Polygon blockchain. This appears to be just expensive enough to prevent spamming attacks, it is cheap enough for a broad usage given that the use case can finally be advertised well. However, there may still be room for improvement on the contract's performance and therefore gas fees, especially in the calculation of the overall ratings and the storage of the values (which for example are currently all uint256 values and arrays of them for the sake of consistency).

The deployment of the contract costs 2549801 Gas summing up to 2549801*100 Gwei = 0.255 ETH/MATIC. This is rather expensive, but as this is only a one-time cost, it is not worth investigating too much on the optimization here, which would basically mean a compression of the compiled code.

Blockchain deployment (Testnet)

Now after the basic functionality and applicability has been proven, it is time to implement the token on a real world (test) blockchain. As stated before, I chose to use the Polygon blockchain for deployment, but the steps to be done for the Ethereum or another EVM compatible blockchain will be nearly the same, except the target addresses. The process is quite straight forward: Just as it has been deployed on the local blockchain, following steps have to be done:

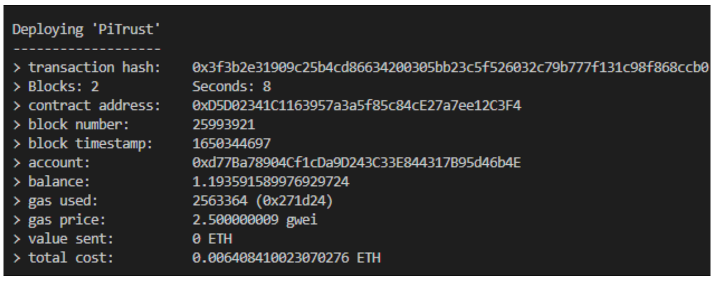
- Create an application on the Alchemy infrastructure provider[17] with target blockchain: Polygon Mumbai (aka Testnet)
- Change the Metamask wallet target to Polygon Mumbai (will be done automatically with the app creation in the Alchemy dashboard
- Add some MATIC tokens to the wallet using a so-called faucet[18], which produces and sends coins/tokens on a testnet out oft thin air, to pay the gas fees for deployment.
- Adapt the settings in the Truffle configuration file to deploy the contract on the Mumbai blockchain rather than the local one



```
mumbai: {
  provider: () => new HDWalletProvider(privateKey, `https://polygon-mumbai.g.alchemy.com/v2/p
  network_id: 80001,
  gas: 5500000,
  confirmations: 2,
  timeoutBlocks: 200,
  skipDryRun: true
},
```

Figure 9: Snippet from truffle-config.js for Mumbai deployment

- Start the migration using the command *truffle migrate --network mumbai* with the following result:

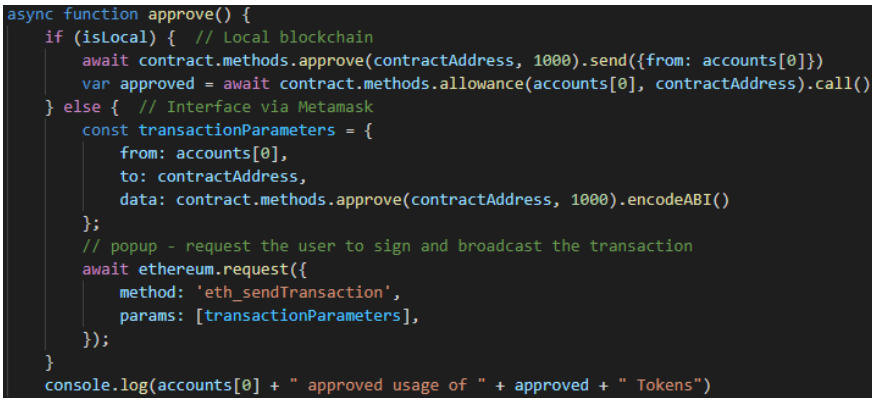


```
Deploying 'PiTrust'
-------------------
> transaction hash:    0x3f3b2e31909c25b4cd86634200305bb23c5f526032c79b777f131c98f868ccb0
> Blocks: 2            Seconds: 8
> contract address:    0xD5D02341C1163957a3a5f85c84cE27a7ee12C3F4
> block number:        25993921
> block timestamp:     1650344697
> account:             0xd77Ba78904Cf1cDa9D243C33E844317B95d46b4E
> balance:             1.193591589976929724
> gas used:            2563364 (0x271d24)
> gas price:           2.500000009 gwei
> value sent:          0 ETH
> total cost:          0.006408410023070276 ETH
```

Figure 10: Deployment confirmation

- After the deployment has been done, the contract appears on the Mumbai blockchain and can be used there. It is also visible that the gas costs which have been paid for the deployment have been roughly equal to those on the local blockchain (2563364 Gwei), but as the Gas *price* is much lower (2.5 Gwei) it only counts to an effective cost of 0.006 MATIC (here referred to as ETH). The successful deployment on the Polygon Mumbai testnet on contract address ***0xD5D02341C1163957a3a5f85c84cE27a7ee12C3F4*** can be verified using the PolygonScan website[19]
  - The functionality of the smart contract is then tested using the (local) frontend, where just the target and contract address must be changed according to the deployment in the Polygon Mumbai testnet.
  - Furthermore, most of the functions in backend.js need to be adapted to rather use the Metamask API than locally stored (and known) private keys. The simplest example is the approve() function needed to allow the usage of the ERC20 tokens in the smart contract based on the isLocal variable (whose value is set based on the local availability of the private keys)

[17] https://dashboard.alchemyapi.io/ -- Other providers such as the before-mentioned Infura or Moralis or even a local Ethereum/Polygon blockchain node may be fine as well.
[18] https://faucet.polygon.technology/
[19] https://mumbai.polygonscan.com/tx/0x3f3b2e31909c25b4cd86634200305bb23c5f526032c79b777f131c98f868ccb0



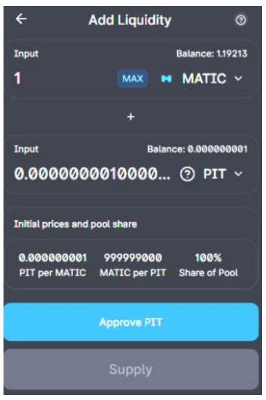
```
async function approve() {
    if (isLocal) {  // Local blockchain
        await contract.methods.approve(contractAddress, 1000).send({from: accounts[0]})
        var approved = await contract.methods.allowance(accounts[0], contractAddress).call()
    } else {  // Interface via Metamask
        const transactionParameters = {
            from: accounts[0],
            to: contractAddress,
            data: contract.methods.approve(contractAddress, 1000).encodeABI()
        };
        // popup - request the user to sign and broadcast the transaction
        await ethereum.request({
            method: 'eth_sendTransaction',
            params: [transactionParameters],
        });
    }
    console.log(accounts[0] + " approved usage of " + approved + " Tokens")
```

Figure 11: Code adaption for approve() function

- Additionally, the functions for actual ratings and adding funds to an account have to be adapted. Read-Only functions such as getting the top experts on a topic or getting the expertise and balance of a topic do not need adaption because they don't require any transaction to be signed.
- The successfull transaction of adding funds to the address 0x4F39E6F580967C447920a4159ccb36C2A4353c61 as well as giving it a rating of 5 in the topic „everything“ can also be reviewed on the Polygon Scan website[20]

Tradability

Adding a token to a decentralized exchange like Quickswap requires the creation of a new so called "liquidity pool" enabling exchanging between two tokens at market price. Describing how a liquidity pool works would exceed this work and is, for example described in [13]. Here it just has to be noted that there was an issue with the token denomination because the actually available 1.000.000.000 tokens appear as 0.000000001 tokens in the interface (Figure 12), i.e., an error in magnitude of 10^{18}, but without hampering the actual functionality. After creating the liquidity pool, the token can be traded on the exchange, although it has to be manually added by any interested person, because it has not been added to the default list of tokens to be tradeable (this will require additional verification with the exchange provider) as shown in Figure 13.



Add Liquidity

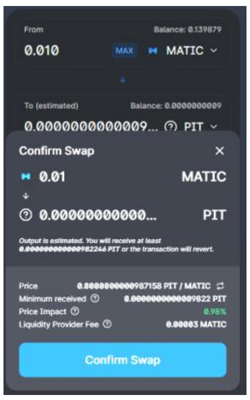
Input Balance: 1.19213
1 MATIC

Input Balance: 0.000000001
0.0000000010000... PIT

Initial prices and pool share

0.000000001 PIT per MATIC
999999000 MATIC per PIT
100% Share of Pool

Figure 12: Adding PiTrust token to the exchange



From Balance: 0.139879
0.010 MATIC

To (estimated) Balance: 0.0000000009
0.0000000000000009... PIT

Confirm Swap

0.01 MATIC

0.000000000000... PIT

Output is estimated. You will receive at least 0.0000000000000982246 PIT or the transaction will revert.

Price 0.0000000000000987158 PIT / MATIC
Minimum received 0.0000000000000009822 PIT
Price Impact 0.98%
Liquidity Provider Fee 0.00003 MATIC

Figure 13: MATIC vs. PiTrust token swap

[20] https://mumbai.polygonscan.com/address/0x4F39E6F580967C447920a4159ccb36C2A4353c61#tokentxns

Blockchain deployment (Mainnet)

The last step of the verification/test of the token implementation is marked by its deployment on a production blockchain, in this case the Polygon PoS chain. The procedure is the same as for the Mumbai testnet deployment with the only adaptions being a change of the target URL (i.e., target blockchain) and the requirement to add “real” MATIC tokens to the deploying wallet. This was done by buying and transferring roughly 2 tokens (worth ~$3.5 at the time of writing) to the deploying wallet from a centralized exchange, which allows buying cryptocurrencies for fiat, i.e., real, money. For this task I used the exchange KuCoin[21], but there are also many more available. The deployment was successful to the contract address *0xD5D02341C1163957a3a5f85c84cE27a7ee12C3F*4, which again can be verified using the Polygon Scan website[22].

Concluding Remarks

Developing a real-world application based on an initial idea and learning either the programming languages Solidity as well as JavaScript on that way was a great journey. The project therefore combined blockchain technology with web server technology (although not described in detail) and P2P Trust management, three fields which are of utmost importance in current times and will become more important in the future. While most of the theoretical designs were already done during the prior homework, the main task of this project was to bring these ideas to live and evaluate the applicability of the theoretical construct. While the creation of the core functionality of the smart contract (capturing and storing of ratings) was done rather fast the practical aspects of searching for users & topics as well as dealing with the incapability of using float operations in Solidity made the backend programming a challenge.

However, the hardest part of this project was the creation of a frontend, which is able to interact either with a local (Ganache) blockchain as well as a public (Polygon) blockchain as those two target behave very differently: When using a local blockchain, the private keys to be used for signing transactions can be made available to the code, while this is not applicable to a real world application, where the private keys shall be managed by a dedicated (hardware or software) wallet such as MetaMask.

Additional changes

Incentivation

Based on the peer feedback (thank you!), I also realized that the system is yet lacking incentives to actually use the system. The incentive of being rated positively is quite clear: Getting acknowledgement and potentially more requests which may lead to monetarization. However, it is not yet clear, why rating *others* will be incentivized. While there is a basic will to acknowledge good experiences (and punish bad ones), doing this in such a novel system may not be as natural as it is on a well-established system such as Amazon, where also only a fraction of buys is actually rated. So, there must be a clear incentive to do so. As one idea, providing ratings to others may scale up the redemption rate of PiTrust tokens in case of

---

[21] https://www.kucoin.com/

[22] https://polygonscan.com/address/0xD5D02341C1163957a3a5f85c84cE27a7ee12C3F4

being rated by *others*. In that way not only rating is incentivized, but also sharing one's experience to others (i.e., being rated themselves), which may to a self-sustaining and -amplifying loop. Getting more tokens for ratings may then be an immediate monetary incentive as these may, by nature of being a cryptocurrency, be traded against other currencies. However, this will have a rather severe impact on the tokenomics, which may exceed the scope of this project and is therefore postponed to further development.

Commenting functionality

If the system is compared to common centralized rating systems, such as Amazon or eBay, it is lacking any prosaic judgement on the expert's knowledge. While it is helpful to have a quantifiable rating on the first run, to be able to create also comments on that rating may be particularly helpful to judge if an expert's knowledge is really matching to the user's needs. However, it has to be mentioned that this may drastically increase the storage necessities and therefore transaction costs of any rating to be performed. This may be circumvented by offloading the actual storage off-chain or to another dedicated blockchain such as Arweave[23].

References

[1] P. Wissmann, *CS 6675 Assignment M4,* 2022.

[2] T. B. T. Foundation, „Braintrust: The Decentralized Talent Network,“ September 2021. [Online]. Available: https://www.usebraintrust.com/whitepaper. [Zugriff am 3 March 2022].

[3] M. T. S. H. G.-M. Sepandar D. Kamvar, „The Eigentrust algorithm for reputation management in P2P networks,“ in *Proceedings of the 12th international conference on World Wide Web*, New York, NY, United States, 2003.

[4] C. J. Zhao Yuhong, „A P2P trust model based on trust factor and feedback aggregation,“ in *3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE),* 2019, pp. 214-219.

[5] B. B. Y. L. P. A. Yuhui Zhong, „A Computational Dynamic Trust Model for User Authorization,“ *IEEE Transactions on Dependable and Secure Computing,* Bd. 12, Nr. 1, pp. 1-15, 2015.

[6] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System,* 2008.

[7] V. Buterin, *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform,* 2014.

[8] Aleph Zero Blog, „What Is The Fastest Blockchain And Why? Analysis of 43 Blockchains,“ 4 January 2021. [Online]. Available: https://alephzero.org/blog/what-is-the-fastest-blockchain-and-why-analysis-of-43-blockchains/. [Zugriff am 30 March 2023].

[23] https://www.arweave.org/

[9] C. M. B. M. G. Gamze Gürsoy, „Using Ethereum blockchain to store and query pharmacogenomics data via smart contracts,“ *BMC Medical Genomics,* Bd. 13, Nr. 74, 2020.

[10] N. Feuerstein, „StackExchange: What is the cost to store 1KB, 10KB, 100KB worth of data into the ethereum blockchain?,“ 23 July 2020. [Online]. Available: https://ethereum.stackexchange.com/questions/872/what-is-the-cost-to-store-1kb-10kb-100kb-worth-of-data-into-the-ethereum-block. [Zugriff am 05 April 2022].

[11] „EIP-20: Token Standard,“ 19 November 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20. [Zugriff am 07 April 2022].

[12] Unknown, „Ethereum Secret Messenger,“ [Online]. Available: https://learning-dcs-bbn.netlify.app/part%2005-module%2001-lesson%2001_new%20introduction%20to%20ethereum/17.%20ethereum%20secret%20messenger. [Zugriff am 14 April 2022].

[13] T.-W. U. B. Z. G. M. L. Nguyen B. Truong, „Strengthening the Blockchain-Based Internet of Value with Trust,“ in *IEEE International Conference on Communications (ICC),* Kansas City, MO, USA, 2018.