Blockchains and Overlay Networks 2020

Challenge Task Report

Tokenized Roulette with Gambling System

Group 2: Ethereum

Anna Jancso, anna.jancso@uzh.ch, 13-700-638
Pascal Kiechl, pascal.kiechl@uzh.ch, 16-927-998
Fabian Küffer, fabian.kueffer@uzh.ch, 15-931-421

1 Introduction

In this report we outline our approach to the Challenge Task (CT) posed in the module 'Blockchains and Overlay Networks' (BCOLN) during the spring semester 2020. Where in previous years the task was a predefined challenge, this year it was left to the individual groups to propose a challenge of their own within the specified parameters, which then had to be approved by the teaching assistants [1].
In our case, we decided to implement a roulette gambling system, where the roulette game logic is implemented fully in a smart contract (SC) on the Ethereum Blockchain.

1.1 Requirements

The implementation details and concrete design decisions were left to us, but a few fundamental requirements were given for all groups to adhere to [1]:

- The core functionality must be implemented and executed entirely within smart contracts (SC).
- The SC must implement an economic aspect, e.g., a payment system, incentives, gambling, or any economy-related functionality.
- The user must interact with the DApp (Decentralized Application) via a Graphical User Interface (GUI), for example, a Web-based one.

- The group must deliver a self-contained report documenting the SC, its operation, and the source code.

1.2 Assumptions

We were also given a few facts that can be assumed regarding our implementation [1]:

- The Smart Contract can be deployed in a private testnet or in the Ropsten testnet
- You can just use one node (Ganache, Parity or Geth client) with multiple addresses.

1.3 Roulette Game

Let us outline a couple of design decisions made with regard to our implementation. We wanted our implementation to capture the feel of an actual game of roulette as closely as we could manage.
As such, next to the basic functionality like being able to place various amounts of currency (ethers) on a variety of different bets and getting feedback on the outcome of the round via a visual representation of the roulette wheel, the implementation also supports a form of local multiplayer, where different users can join the game on different browsers. The roulette wheel then only spins, once all joined users have finished placing bets. Note that those different users all need to have their own valid account address to be able to join.
Additionally we opted to go for high-quality randomness in our oracle, instead of pseudo-randomness. For a closer look at randomness visit section 2.1.2, which deals with the generation of random numbers.
Lastly, we decided to have the user experience be intuitive and without the need of a manual, thus providing feedback in the UI via Modals, Alerts and context-dependent disabling and enabling of buttons.

2 Implementation

Our decentralized app (DApp) consists of the following components:

- Blockchain
- Smart Contracts
- Frontend

During development, we created Ethereum blockchains using Ganache [2]. Furthermore, we used truffle [3] to compile and migrate our contracts. In the following section, we will describe the implementation of the smart contracts as well as the frontend in more detail.

2.1 Smart Contracts

We implemented two contracts, Roulette and Oracle. The Roulette contract steers the whole game. It interacts with the oracle contract which is responsible for generating random numbers. The oracle contract in turn uses the Oraclize contract. In the following sections, we describe the implementation of the contracts.

2.1.1 Roulette

The roulette contract controls the submission and creation of bets, the start of the roulette and the payout of ethers to the winners.

Joining the roulette game

Before users can place bets, they need to join the game by calling the `join()` method on the contract. The contract does not explicitly store the user's identities, but simply keeps a counter (`clientCount`) for the number of users that have entered. This counter is needed to determine when the roulette may begin.

Submitting and Creating bets

A user may submit any of the bets supported in a typical European roulette game, namely:

Outside Bets:

- Even
- Odd
- Red
- Black
- First dozen (1-12)
- Second dozen (13-24)
- Third dozen (25-36)
- Lows (1-18)
- Highs (19-36)

Inside Bets:

- 1 number (Straight/Single)
- 2 numbers (Split)
- 4 numbers (Corner/Square, First Four)

For each type of bet there is a designated method in the contract that the user can call to submit a given type of bet with a certain amount of ethers. These methods are responsible for creating and persisting the appropriate bets. For this purpose, we implemented a `Bet` struct to record the submitter's address (`owner`) and the amount of ethers (`winningAmount`) that is paid out to the submitter when the bet is won. Furthermore, the `Bet` instance stores the numbers with which the bet can win the game (`winningNumbers`). The bet methods are responsible for populating these attributes correctly. For example, the bet method `evenBet()` pushes all even numbers between 1 and 36 to the `winningNumbers` array and sets the `winningAmount` to twice the amount of ethers placed by the submitter, while the bet method `betComboFour()` would push the four numbers entered by the submitter and set `winningAmount` to nine times the amount of ethers sent by the submitter. Finally, the `Bet` instance is added to an array (`bets`) which stores all bets submitted so far by all users.

Starting the roulette game

Once the user has placed all desired bets, he/she must notify the contract by calling the method `setReady()`. This method increments a counter (`readyCount`) that records the number of users that are ready, and calls the `allReady()` method that determines whether the roulette may begin at this point. The roulette starts when all users that have joined the game are ready, technically speaking, when the `clientCount` and `readyCount` counters are equal.

Payout of Ethers

Next, the contract generates a random number using the `Oracle` contract (for more details, see next Section). It then evaluates (`evaluate()`) which bets contain the random number in their list of `winningNumbers` by looping through the `bets` array. When the random number occurs in the bet, the contract pays the submitter out with the given `winningAmount` (`payout()`). Once the payout is performed, the game's state is reset by deleting all bets and setting counters to their initial values (`teardown()`). Following this, an event (`RouletteDone`) is triggered to start the wheel in the frontend sending along the random number.

2.1.2 Oracle

Generating (Pseudo) Random Numbers

Every roulette game deals inherently with random numbers, and as such we had to come up with a way to generate random numbers reliably and deterministically.

Since Solidity is not able to create random numbers, and complex algorithms for pseudo random numbers would cost too much gas, and thus be unfeasible for our needs, we went through various approaches to deal with the issue of generating random numbers for our roulette [4].

A simple, and also our first approach was to hash the current block timestamp using keccak256, on the first hand it presents us with the advantage that no player of our roulette game is able to tamper with the number generation, but on the other hand we put our trust into the randomness of the number generation with the miners [4]. Since miners are able to influence the block time, we decided to not use this simple solution [5].

```
uint256(keccak256(abi.encodePacked(now)))%37;
```

Similar solutions would be to include, or to only hash the block difficulty or the block hash, but these sources of random numbers do not offer any substantial advantages, if any [4].

For our next approach, we looked into the Niguez Randomity Engine, which is a smart contract as well, and as such our oracle smart contract outsources the random number generation. The Niguez Randomity Engine works by generating 24 sequences of numbers, which can be then combined in various ways, e.g. modulo and addition operations, to create a pseudo random number. It offers the advantage, that it is free to use and the sequence operations would not cost much gas. Additionally, the author argues that it is immutable, since the user of the randomity engine would be the one to decide on which slots of the generated sequence would be used [5]. As such, we understand that miners would not be able to directly influence the outcome. We decided to not look further into this approach, since we were intrigued by true randomness.

For our final approach, we settled on using RANDOM.ORG for our number generation. They are able to generate true randomness through atmospheric noise, which has a very high level of information entropy [6]. This approach offers the distinct advantage of not having to rely on pseudo random numbers, but complicates the oracle logic a fair bit, since we were presented with the problem of sourcing these numbers. Additionally, we not only need to trust their claims of randomness, but their service as well.

Oracle Services

To source our random numbers from RANDOM.ORG's API, we needed a way to be able to communicate with the outside world. While this might sound trivial, it is not an easy feat in the Ethereum blockchain world, since all mining nodes must get the same result from an API call, which cannot be guaranteed when interacting with the outside world [7,8]. Thus, it is not possible for a smart contract to directly access the required data, and to be able to communicate with the outside world, we need to use oracle services [9,10].

Oracle services watch and read smart contracts and by using off-chain entities they are able to send queries. The results of those queries are then returned into the smart contract via a callback function [9]. The oracle service that we decided to use for our API calls is Oraclize (rebranded to Provable now).

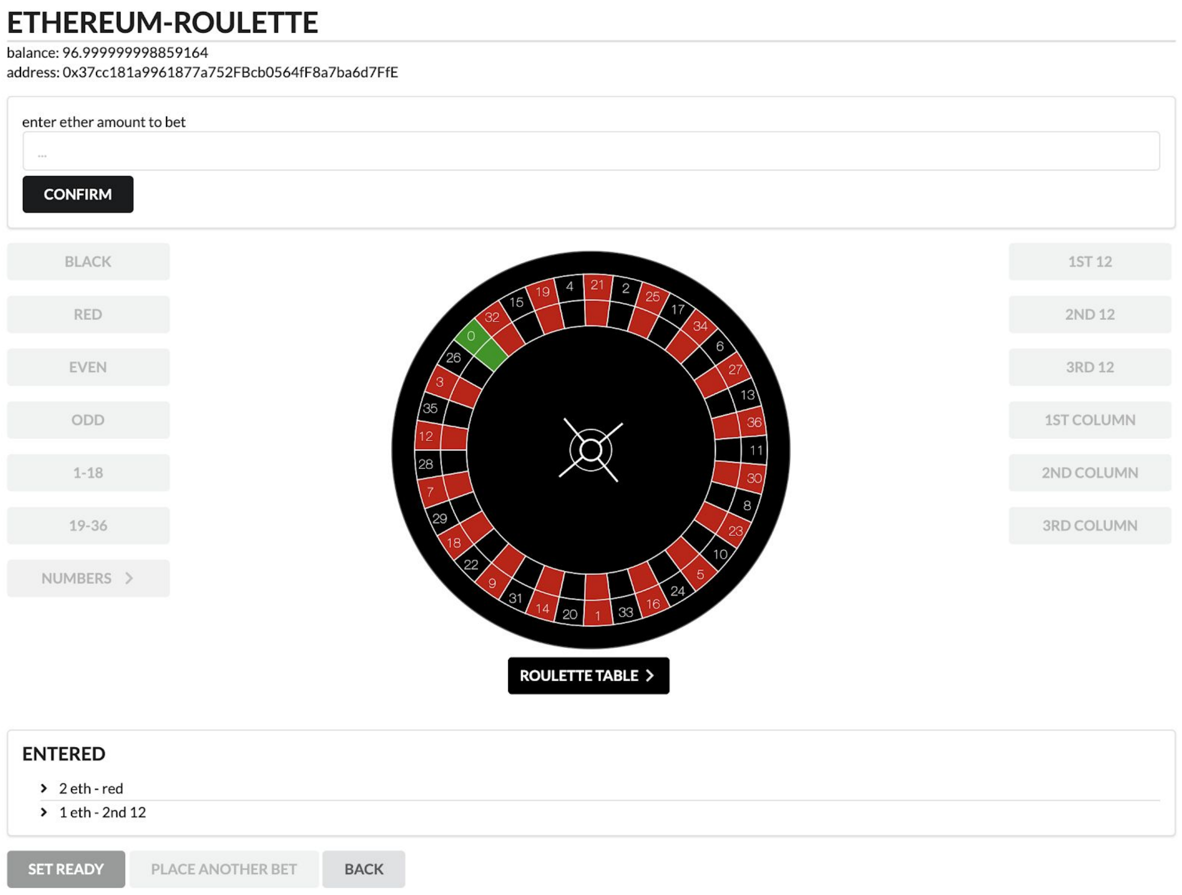
Although we solved our problem of trusting the random numbers, we introduced another problem, since we added a centralized third party. We need to fully trust the Oraclize service, and it is not clear how security breaches on their end would affect the operability of our roulette game [8]. Whilst we cannot guarantee operability of our roulette game, we would still be able to detect if our random generated number queries have been tampered with, since RANDOM.ORG offers to digitally sign the results and also to check whether signatures originate from RANDOM.ORG [11]. Due to constraints, we choose to not add this feature, but we will discuss it further in Section 3.

2.1.3 Frontend

For the UI we decided to work with the react.js library [12], whilst using semantic ui, or more specifically its react integration [13], for most of the CSS work. Furthermore we use the web3.js library [14] to enable the frontend to communicate with the SCs on the blockchain. We also make use of the Metamask Browser-Extension to allow users to sign their betting transactions [15].

User Interface

As mentioned in the Introduction section, our main goal with regards to the UI was that it should be usable without a manual, thus sufficiently simple. Figure 1 shows the main page of the UI, where a user is directed to upon entering a valid account address.


ETHEREUM-ROULETTE

balance: 96.999999998859164

address: 0x37cc181a9961877a752FBcb0564fF8a7ba6d7FfE

enter ether amount to bet

ENTERED

- 2 eth - red
- 1 eth - 2nd 12

Figure 1*: Main page of the UI*

Roulette Wheel

The centerpiece of a roulette game is (we feel) the roulette wheel, as it acts as the main source of feedback on what is currently going on in the game. The state of the wheel relates directly to the state of the game. If the wheel is spinning, then the game is currently generating the random number. If the wheel is standing still and there is no ball in any of the numbered slots, then the game is in the betting phase and finally, if the wheel is standing still and there is a ball on one of the numbers, then it is time to evaluate the bets and perform the appropriate payouts.
This translates perfectly to the UI of any given roulette implementation, provided that the necessary tools and more importantly the necessary abilities to create such a graphical representation are at hand. In our case, none of us has any experience in creating graphical objects with interacting, moving parts like a roulette wheel and a ball that moves across it as it spins. Our initial idea was thus to use an abstraction of the wheel that still provides the ability to relate to the user a sense of the dynamics of a spinning wheel, as well as the ability to highlight some element on the wheel to indicate the winning number.

Figure 2 and Figure 3 depict two such potential abstractions. First a data picker, as you might know it from iOS devices. Second, a similar concept, but more in line with a slot-machine [16]. The second option's source code is even available under the MIT-license

[17], but would need some adjustments to port it from jQuery to react, as according to some it is not advised to use them in conjunction [18].

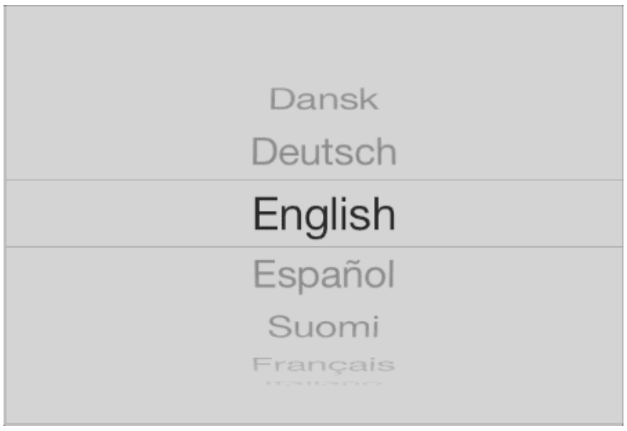


Figure 2: *iOS data picker [19]*

However, we also found an actual visual representation of a roulette wheel [20], once again available with source code under the MIT-license. We decided to use this implementation and make the necessary adjustments to the wheel to port it from an american roulette wheel to a european one [21]. The resulting wheel can be seen in Figure 1. Additionally, adjustments to the start and stop logic of the wheel and the ball had to be made, to make it compatible with our UI and game-flow.

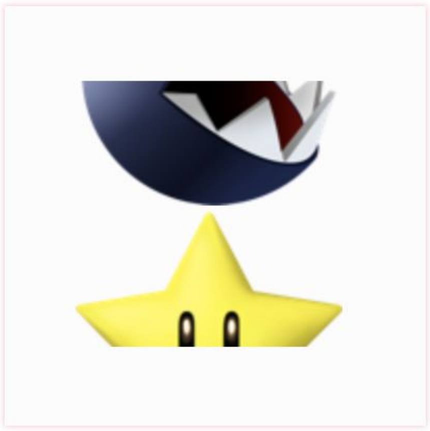


Figure 3: *st-marron demo [22]*

Reliance on Events

For our communication between the fronted and the SCs, we needed a way for the client to know if certain routines in the SCs have run. Take for example the information if the current round of roulette has been concluded and a winning number has been generated on the SCs, such that the roulette wheel can begin spinning in the UI.
We did not want to constantly poll the SCs, to see if the SCs have performed certain computations, as that would lead to an unnecessarily high amount of requests being sent. This is accentuated when there are multiple users participating in a round and some are waiting, whilst others are still placing bets.

Instead we opted to work with events, which the SCs can emit and which the clients then can act upon. This way, the SCs are the ones performing the communication and the clients that are done betting can simply listen for the incoming event.

3 Discussion

3.1 Areas of Improvement

3.1.1 Digital signing and verifying of results

Per Section 2.1.2, RANDOM.ORG offers to digitally sign the results, which is a feature that was not implemented. In conjunction with verifying each result as well, it would improve our system's security greatly.
We imagine that users could also be interested in verifying the results directly with RANDOM.ORG, and not having to only trust our claims of the outcome of the results. This feature could be added in a graphical way to the client.

3.1.2 UI/UX Changes

While we are generally happy with our UI, we feel that a clickable roulette table to place bets would offer a better user experience. The table could also show the bets directly and would make the user interface simpler and more intuitive. Additionally, we feel that we could improve the feedback for the user, like showing how many players are in the game room and how many have pressed ready.

3.1.3 Security

Using asymmetric cryptography, we could avoid handling the account addresses in plaintext in the Smart Contract, to improve security.

4 Conclusion

In this project, we have built a web-based roulette gambling system in which all roulette-related actions are recorded on an Ethereum blockchain mediated by smart contracts. While the system is not ready to go into production yet since we have only tested it locally on private blockchains and because it has some security issues, we consider our implementation a proof-of-concept. We leave it to the interested reader to make necessary modifications to the code to make it production-ready. The source code is available on GitHub [23].

References

1. Challenge Task 2020. [cited 13 May 2020]. Available: https://www.csg.uzh.ch/csg/en/teaching/FS20/p2p/challenge.html

2. Ganache | Truffle Suite. In: Truffle Suite [Internet]. [cited 13 May 2020]. Available: https://trufflesuite.com/ganache

3. Truffle | Truffle Suite. In: Truffle Suite [Internet]. [cited 13 May 2020]. Available: https://trufflesuite.com/truffle

4. Mulders M. Solidity Pitfalls: Random Number Generation for Ethereum — SitePoint. In: SitePoint [Internet]. 4 Jun 2018 [cited 13 May 2020]. Available: https://www.sitepoint.com/solidity-pitfalls-random-number-generation-for-ethereum/

5. The Plutocrat. Niguez Randomity Engine - Niguez Randomity Engine - Medium. 2019 [cited 13 May 2020]. Available: https://medium.com/niguez-randomity-engine/generating-random-numbers-on-the-ethereum-blockchain-using-solidity-random-number-generator-solidity-9f503c7e4d92

6. Information Entropy. [cited 13 May 2020]. Available: https://www.random.org/statistics/information-entropy/

7. Darius. How Ethereum contract can communicate with external data source. 2017 [cited 13 May 2020]. Available: https://medium.com/aigang-network/how-ethereum-contract-can-communicate-with-external-data-source-2e32616ea180

8. How can an Ethereum contract get data from a website? In: Ethereum Stack Exchange [Internet]. [cited 13 May 2020]. Available: https://ethereum.stackexchange.com/questions/2/how-can-an-ethereum-contract-get-data-from-a-website

9. Willems L. How to create a DApp using Truffle, Oraclize, ethereum-bridge and Webpack. 2018 [cited 13 May 2020]. Available: https://medium.com/coinmonks/how-to-create-a-dapp-using-truffle-oraclize-ethereum-bridge-and-webpack-9cb84b8f6bcb

10. Provable Documentation. [cited 13 May 2020]. Available: https://docs.provable.xyz/#background

11. Signed API (Release 2). In: RANDOM.ORG [Internet]. [cited 13 May 2020]. Available: https://api.random.org/json-rpc/2/signed

12. React – A JavaScript library for building user interfaces. [cited 13 May 2020]. Available: https://reactjs.org/

13. Introduction - Semantic UI React. [cited 13 May 2020]. Available: https://react.semantic-ui.com

14. ethereum. ethereum/web3.js. In: GitHub [Internet]. [cited 13 May 2020]. Available:

https://github.com/ethereum/web3.js

15. MetaMask. [cited 14 May 2020]. Available: https://metamask.io
16. akira-kuriyama. akira-kuriyama/roulette.js. In: GitHub [Internet]. [cited 13 May 2020]. Available: https://github.com/akira-kuriyama/roulette.js
17. Contributors to Wikimedia projects. MIT License. 2001 [cited 13 May 2020]. Available: https://en.wikipedia.org/wiki/MIT_License
18. How to use JQuery with ReactJS. In: Stack Overflow [Internet]. [cited 13 May 2020]. Available: https://stackoverflow.com/questions/38518278/how-to-use-jquery-with-reactjs
19. iOS data picker. [cited 13 May 2020]. Available: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/TransitionGuide/Art/picker_7_2x.png
20. ledlogic. ledlogic/roulette. In: GitHub [Internet]. [cited 13 May 2020]. Available: https://github.com/ledlogic/roulette
21. American Roulette Wheel vs European Roulette Wheel: Which is Best? In: Best Online Casino Bonuses | Casinochecking.com [Internet]. [cited 13 May 2020]. Available: https://casinochecking.com/blog/american-and-european-roulette-wheel/
22. Roulette.js Demo. [cited 13 May 2020]. Available: http://demo.st-marron.info/roulette/sample/demo.html
23. PKiechl. PKiechl/BCOLN. In: GitHub [Internet]. [cited 13 May 2020]. Available: https://github.com/PKiechl/BCOLN