



University of
Zurich^{UZH}

Tokenized Fee Service Layer for ERC-865

*Severin Wullschleger
Dübendorf, Switzerland
Student ID: 13-715-081*

Supervisor: Eder Scheid, Prof. Dr. Thomas Bocek
Date of Submission: November 27, 2019

Abstract

This report on the work "Tokenized Fee Service Layer for ERC-865" was written as part of an Independence Study at the University of Zurich.

There are numerous token-based applications. Most tokens are ERC-20 tokens on the Ethereum blockchain. If an ERC-20 token is used as a payment method for an application, in addition to the token itself, ether is always required to pay the transaction costs. This represents an enormous loss of usability. Especially people who are not blockchain-savvy do not understand the context and thus there are enormous entry barriers.

Therefore a "Tokenized Fee Service Layer" for ERC-865 tokens was developed in this thesis. ERC-865 tokens allow a delegated transfer of tokens in which a third party writes the transaction to the blockchain and pays transaction costs in the form of Ether and receives a fee in tokens from the client in return.

The developed service layer consists of a web-based frontend that communicates with an API. The frontend does not require any interaction with an Ethereum client. This opens up countless possibilities to improve the usability of token-based applications and to make the blockchain "disappear" more and more. For example, this service layer can be visited without problems on a smart phone and a private key QR code can be scanned to sign the transaction. The service layer is independent from the frontend, so the API calls can also be integrated into existing applications. The only functionality that the client must provide is the correct encoding and signing of the transaction data.

The implemented end product was presented to different experienced test users and they were able to use it without any problems. The feedback was positive.

Future work could include generalizing the interface so that unspecified contracts can also be called. Or adding a dynamic fee calculation or integrating the API into an existing application to test and improve the service with a real use case.

Zusammenfassung

Der vorliegende Report zur Arbeit "Tokenized Fee Service Layer for ERC-865" wurde im Rahmen einer Independence Study an der Universität Zürich geschrieben.

Es bestehen zahlreiche token-basierte Applikationen. Die meisten Tokens sind ERC-20 Tokens auf der Ethereum Blockchain. Wenn ein ERC-20 Token als Bezahlungsmittel für eine Applikation benutzt wird, wird neben dem Token immer Ether benötigt, um die Transaktionskosten zu bezahlen. Dies stellt einen enormen Verlust der Benutzerfreundlichkeit dar. Vor allem nicht Blockchain versierte Personen, verstehen den Zusammenhang nicht und somit besteht enorme Eintrittsbarrieren.

Darum wurde in dieser Arbeit ein "Tokenized Fee Service Layer" für ERC-865 Tokens entwickelt. ERC-865 Tokens erlauben einen delegierten Transfer von Tokens bei dem eine Drittpartei die Transaktion auf die Blockchain schreibt und dafür Transaktionskosten in Form von Ether bezahlt und vom Auftraggeber im Gegenzug eine Gebühr in Tokens erhält.

Der entwickelte Service Layer besteht aus einem Web-basierten Frontend, der mit einer API kommuniziert. Das Frontend benötigt keinerlei Interaktion mit einem Ethereum Client. Das eröffnet unzählige Möglichkeiten, um die Benutzerfreundlichkeit von Token-basierten Applikationen zu verbessern und die Blockchain immer mehr "verschwinden" zu lassen. Beispielsweise kann dieser Service Layer ohne Probleme auf einem Smart Phone besucht werden und, um die Transaktion zu signieren, kann mit dem Gerät ein Private-Key-QR-Code gescannt werden. Der Service Layer ist vom Frontend unabhängig, somit können die API Aufrufe auch in bestehende Applikationen integriert werden. Die einzige Funktionalität die der Client bereitstellen muss, ist das richtige encodieren und signieren der Transaktionsdaten.

Das implementierte Endprodukt wurde unterschiedlich versierten Test-Usern vorgelegt und konnte ohne Probleme benutzt werden. Das Feedback war positiv.

Zukünftige Arbeiten könnten das Generalisieren der Schnittstelle beinhalten, sodass auch nicht spezifizierte Contracts aufgerufen werden können. Oder auch das Ergänzen einer dynamischen Gebührenberechnung oder das Integrieren der API in eine bestehende Applikation, um den Service mit einem echten Use Case zu testen und zu verbessern.

Contents

Abstract	i
Zusammenfassung	iii
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Thesis Outline	2
2 Related Work and Background	3
2.1 Background	3
2.1.1 ERC-20 token interface	3
2.1.2 Ethereum's Signatures	6
2.2 Related Work	7
2.2.1 ERC-223 token interface	7
2.2.2 ERC-677 token interface	9
2.2.3 ERC-865 token interface	10
2.2.4 Comparison	13
3 Design	15
3.1 Architecture Overview	15
3.2 Application Flow	17
3.3 Frontend Functionalities	18

4	Implementation	19
4.1	Implemented Architecture	19
4.2	Use Case and Example Smart Contracts	21
4.2.1	Use Case	21
4.2.2	The CHFT, the Swiss Franc Token contract	21
4.2.3	The Service Contract: a Simple Translation Service	26
4.3	Service Configuration	29
4.4	Frontend	31
4.4.1	User Interface	31
4.4.2	Signing the Transaction	37
4.4.3	API Calls and Notifications	40
4.5	Service Layer	42
4.5.1	REST API	42
4.5.2	Data Storage	44
4.5.3	Ethereum Interface	45
4.6	Ethereum Blockchain	48
4.6.1	Example Smart Contract	48
4.6.2	Full Node Specification	49
4.6.3	Deployment Script for Development and Testing	49
5	Evaluation and Discussion	51
5.1	Evaluation	51
5.2	Discussion	53
6	Summary and Future Work	55
	Bibliography	57
	List of Figures	58
	List of Tables	59

A	Installation Guidelines	63
A.1	Pre-Requisites	63
A.2	Service Layer Frontend	63
A.3	Ethereum Full Node	64
A.4	Smart Contracts	64
A.5	Service Layer Backend	64
B	Contents of the CD	67

Chapter 1

Introduction

1.1 Motivation

Utility tokens received attention over the past years due to Initial Coin Offerings (ICO) and Initial Token Offerings (ITO) as a mean to raise funds and share profit. A utility token is a token which can be used as payment on a blockchain-based platform using a cryptocurrency instead of fiat money. The Ethereum foundation [9] provides the ERC-20 token interface [5] which allows users to create their own token and save the transactions with these tokens on the Ethereum Blockchain (BC).

According to [2] there are more than 1200 decentralised platforms, under development or developed, which rely on a utility token to perform transactions. Some of these platforms use the token interface ERC-677 [7]. The ERC-677 token interface contains the ERC-20 interface [5] and specifies in addition a `transferAndCall` method which allows the user to transfer tokens to another Smart Contract (SC) and, if the transfer is successful, calling a specified method of this other contract within the same transaction. This process could be used as a fee for using the platform and the SC.

Transactions which are sent to the Ethereum BC lead to transaction costs (*i.e.* *gas*) which needs to be paid in Ether. Every step in a decentralised platform which communicates with the SC results in transaction costs payable in Ether. This leads to an inconvenience for the user since one has to have the specific token of the platform as well as Ether to pay the transaction costs.

For solving this problem, some platforms respectively SCs implement another token interface: the ERC-865 interface [8]. This interface defines a `transferPreSigned` function which enables token holders to pay transfer transactions in tokens instead of *gas*, in one transaction. The token holder delegates the transfer of the tokens to a third party, which pays for the gas in Ether and takes a fee in tokens. This allows the token holder to own the specific token only instead of buying Ether for paying the gas.

While there are ERC-865 SCs around, no service layer as the proposed exists.

1.2 Description of Work

This Independent Study (IS) presents a design and implementation as well as a real-world example of such a service layer. This IS introduces the concepts involved in the implementation of tokens in Ethereum, as well as the ERC-20 token standard [5], the ERC-865 [8] and the ERC-677 [7] standard. There have been some proposals of implementations of the ERC-865 standard, such as the one in the OpenZeppelin project [18]. However, the implementation is under active development, and the developers are still solving issues related to the standard. It should be noted that the ERC-865 standard is still in its infancy; thus, research about design decisions and its implementation must be conducted to address the challenges involved and to enhance the standard further, applying it in a real-world use-case.

The main goal of this IS is to implement a third-party service that acts as a standalone service layer for ERC-865 standard tokens. This third-party must pay for the consumed gas in Ether and receives a fee in the form of tokens paid by the user. Thus, users that do not hold Ethers can send their signed transaction to the implemented service layer, which will send this transaction to the BC and pay for the fees. The service layer is envisioned to be a standalone service provided by the Communication System Research Group (CSG) and run on a configured Ethereum node in the group's infrastructure.

In summary, the goals of this IS are:

- Research on ERC-20, ERC-865, and ERC-677 standards: Conduct a complete research on the standards proposed by the Ethereum foundation that serve as basis for the work.
- Design and implementation of the Third-Party Service: Design and implementation of the service layer allowing ERC-865 tokens to be transferred without having Ether on your wallet.
- Design and implementation of ERC-677 Functionality: Include, in the Third-Party Service, the function that allows users to call SC functions using the `transferAndCall` method of the ERC-677 token interface.

1.3 Thesis Outline

This IS report is structured as follows. Section 2 presents current efforts in the definition and implementation of Token contracts and interfaces. Then, Section 3 describes the design of the approach, including its architecture, application flow, and the basic structure of the functionalities. Section 4 contains the implementation of the approach components, which is evaluated and discussed in Section 5. Finally, Section 6 summarises the report and presents future work directions.

Chapter 2

Related Work and Background

This section presents the background on token interfaces and related work regarding the approach.

2.1 Background

In the following section it is described what an ERC-20 interface is and how Ethereum signatures look like, because this is basic knowledge to understand the following chapters.

2.1.1 ERC-20 token interface

The ERC-20 token interface is one of the first and the most used token interface in the Ethereum community. Etherscan [12], the most popular Ethereum BC explorer, states to find almost 220 000 token contracts implementing the ERC-20 token interface (October 2019). The reason why the ERC-20 token standard is so popular is that, it allows tokens to be reusable in every application (e.g., wallets or decentralised applications) which is familiar with the ERC-20 interface.

The Ethereum Improvement Proposal 20 (EIP 20) [10] includes the official documentation of the ERC-20 interface.

Listing 2.1 presents the ERC20 interface (IERC20) and in Listing 2.2 a simple Example ERC-20 token is provided by OpenZeppelin [17], an organisation which provides standards for secure BC applications.

```
1 pragma solidity ^0.5.0;
2
3 /**
4  * @dev Interface of the ERC20 standard as defined in the EIP. Does not
      include
5  * the optional functions; to access them see {ERC20Detailed}.
6  */
```

```

7 interface IERC20 {
8
9     function totalSupply() external view returns (uint256);
10
11    function balanceOf(address account) external view returns (uint256);
12
13    function transfer(address recipient, uint256 amount) external
14        returns (bool);
15
16    function allowance(address owner, address spender) external view
17        returns (uint256);
18
19    /**
20     * [...]
21     * IMPORTANT: Beware that changing an allowance with this method
22     * brings the risk
23     * that someone may use both the old and the new allowance by
24     * unfortunate
25     * transaction ordering. One possible solution to mitigate this race
26     * condition is to first reduce the spender's allowance to 0 and set
27     * the
28     * desired value afterwards:
29     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
30     * [...]
31     */
32    function approve(address spender, uint256 amount) external returns (
33        bool);
34
35    function transferFrom(address sender, address recipient, uint256
36        amount) external returns (bool);
37
38    event Transfer(address indexed from, address indexed to, uint256
39        value);
40    event Approval(address indexed owner, address indexed spender,
41        uint256 value);
42 }

```

Listing 2.1: OpenZeppelin's IERC20.sol: The ERC-20 interface written in Solidity.

```

1 pragma solidity ^0.5.0;
2
3 import "../GSN/Context.sol";
4 import "../IERC20.sol";
5 import "../math/SafeMath.sol";
6
7 /**
8  * [...]
9  * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
10  * functions have been added to mitigate the well-known issues around
11  * setting
12  * allowances. See {IERC20-approve}.
13  */
14 contract ERC20 is Context, IERC20 {
15     using SafeMath for uint256;
16
17     mapping (address => uint256) private _balances;

```

```
18     mapping (address => mapping (address => uint256)) private
19         _allowances;
20
21     uint256 private _totalSupply;
22
23     /**
24      * @dev See {IERC20-totalSupply}.
25      */
26     function totalSupply() public view returns (uint256) {
27         return _totalSupply;
28     }
29
30     // [...] Implementation of all functions of IERC20.sol
31 }
```

Listing 2.2: OpenZeppelin’s ERC20.sol: An example ERC-20 token.

The code snippet in Listing 2.1 shows that the ERC-20 token interface has at least 5 functions: `totalSupply`, `balanceOf`, `allowance`, `approve`, and `transferFrom`. A part of the comment before the `approve` method declaration was kept. A vulnerability when setting the allowance which still exists for older ERC20 contracts such as the Tether (USDT). Newer ERC-20 versions are therefore forced to add `decrease-` and `increaseAllowance` methods.

Transferring ERC-20 token to SCs

One challenge of ERC-20 tokens is the transfer of tokens from an Externally Owned Account (EOA) to a SC. In Ethereum there are generally two types of accounts, externally owned accounts which are controlled by private keys, and contract accounts which are controlled by their contract code. The ERC-20 interface provides the methods `approve` and `transferFrom` to manage this scenario. Figure 2.1 depicts the sequence diagram of this process, which involves two steps:

1. Alice approves the service contract to transfer some of her funds by calling the `approve` method of the ERC-20 token contract with the address of the service contract and the amount which she allows to be used.
2. Alice calls a function of the service contract. This function calls the `transferFrom` method of the ERC-20 token contract. Since the amount was approved before by Alice the contract is allowed to move the specified amount to somewhere else. The service contract can react on the successful or not successful transfer call and can perform some according actions.

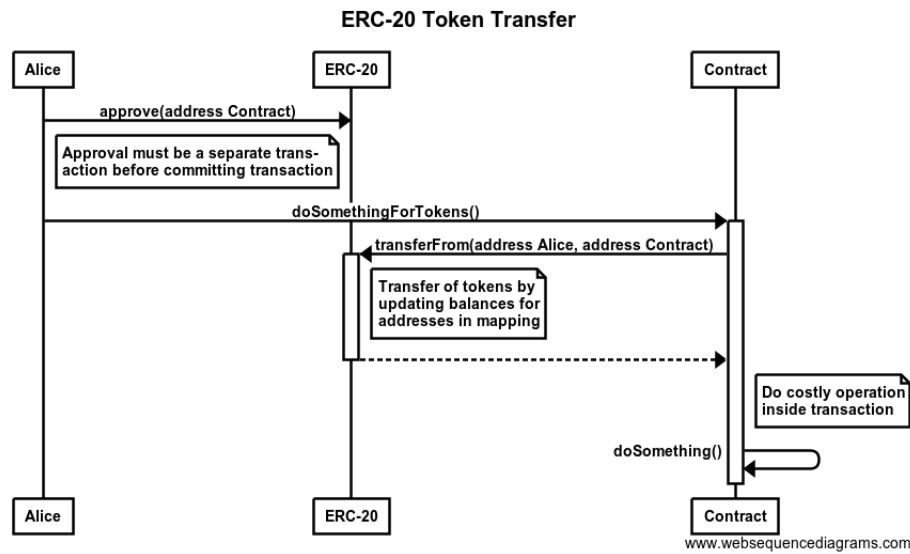


Figure 2.1: The sequence diagram of transferring ERC20 tokens to a smart contract and do something [6].

This process shows that, when it is about transferring ERC-20 tokens to SCs, Alice is not triggering the transfer-process, instead, she only approves the amount for the service contract and the service contract is triggering the transfer by itself.

This solution solves an issue but it opens another one: By not following the described process, end users can lose money. SCs which are not intended to work with ERC-20 tokens can not recognise incoming transactions. Therefore, when a user transfers ERC-20 tokens to a wrong SC, its tokens are stuck at the contracts balance and are lost forever [21].

2.1.2 Ethereum's Signatures

Ethereum [11] uses the asymmetric cryptography algorithm ECDSA, Elliptic Curve Digital Signature Algorithm, which is another option of the Digital Signature Algorithm. The difference between the ECDSA and the normal DSA is at the security level. To have the same security level an ECDSA private key can be relevantly shorter than a DSA private key [4].

Asymmetric cryptography is used to verify and assure the integrity of transactions and messages. Alice can use her public and private key together to sign every piece of data. If she sends the retrieved signature, together with the signed data and her public key to Bob, Bob can validate if she really signed that exact data. Bob uses the signature and the data to retrieve a public key. If it is Alice' public key, Bob knows that the signature is real and therefore the data is the one signed by Alice.

Ethereum's Signature (and also Bitcoin's) can be splitted in three parts: R, S and V. R and S are the normal outputs from an ECDSA signature. V is the so called recovery identifier, a 1 byte value. For a valid Ethereum signature, V is either 27 or 28 [11].

2.2 Related Work

In this section existing ERC implementations are displayed and compared.

2.2.1 ERC-223 token interface

The first who tackled the problem of loosing money by directly transferring ERC-20 tokens to not-prepared SCs was a Github user called "Dexaran". Dexaran implemented the ERC-223 token interface [6].

The main goal of ERC-223 is to prevent the loss of money. The transfer of an ERC-223 token does first a normal transfer, if this is successful and the `to` address is a contract and not an externally owned account, it calls a `tokenFallback` method of the receiving SC. If the `tokenFallback` method does not exist on the receiving SC, the whole transaction fails, preventing the user from loosing the tokens. A failing transaction because of a not-implemented `tokenFallback` method costs the user all of the provided gas, which is being critiqued a lot [21].

```

1  /**
2   * @dev Transfer the specified amount of tokens to the specified
      address.
3   *     Invokes the 'tokenFallback' function if the recipient is a
      contract.
4   *     The token transfer fails if the recipient is a contract
5   *     but does not implement the 'tokenFallback' function
6   *     or the fallback function to receive funds.
7   *
8   * @param _to      Receiver address.
9   * @param _value   Amount of tokens that will be transferred.
10  * @param _data    Transaction metadata.
11  */
12  function transfer(address _to, uint _value, bytes memory _data)
      public returns (bool success){
13      // Standard function transfer similar to ERC20 transfer with no
          _data .
14      // Added due to backwards compatibility reasons .
15      balances[msg.sender] = balances[msg.sender].sub(_value);
16      balances[_to] = balances[_to].add(_value);
17      if(Address.isContract(_to)) {
18          IERC223Recipient receiver = IERC223Recipient(_to);
19          receiver.tokenFallback(msg.sender, _value, _data);
20      }
21      emit Transfer(msg.sender, _to, _value, _data);
22      return true;
23  }
24
25  /**
26   * @dev Transfer the specified amount of tokens to the specified
      address.
27   *     This function works the same with the previous one
28   *     but doesn't contain '_data' param.
29   *     Added due to backwards compatibility reasons.

```

```

30  *
31  * @param _to      Receiver address.
32  * @param _value Amount of tokens that will be transferred.
33  */
34  function transfer(address _to, uint _value) public returns (bool
    success){
35      bytes memory empty = hex"00000000";
36      balances[msg.sender] = balances[msg.sender].sub(_value);
37      balances[_to] = balances[_to].add(_value);
38      if(Address.isContract(_to)) {
39          IERC223Recipient receiver = IERC223Recipient(_to);
40          receiver.tokenFallback(msg.sender, _value, empty);
41      }
42      emit Transfer(msg.sender, _to, _value, empty);
43      return true;
44  }

```

Listing 2.3: From Dexaran’s ERC223.sol: The transfer methods of an example ERC-223 token.

```

1  pragma solidity ^0.5.1;
2
3  /**
4   * @title Contract that will work with ERC223 tokens.
5   */
6
7  contract IERC223Recipient {
8      /**
9       * @dev Standard ERC223 function that will handle incoming token
        transfers.
10
11      *
12      * @param _from Token sender address.
13      * @param _value Amount of tokens.
14      * @param _data Transaction metadata.
15      */
16      function tokenFallback(address _from, uint _value, bytes memory
        _data) public;

```

Listing 2.4: Dexaran’s IERC223Recipient.sol: The interface for a SC which is able to receive ERC-223 tokens.

In the Listings 2.3 and Listing 2.4 present Dexaran’s implementations of the transfer methods as well as the recipient interface are shown. The recipient needs to implement a `tokenFallback` method whereas the ERC-223 token itself needs to implements a transfer method which calls recipient’s `tokenFallback` method with the specified `_data` parameter. To make the token backward compatible, he suggests to implement a normal transfer method without `_data` parameter and call the new transfer method with an empty data array.

The Github issue is still ”open” and the ERC status still on ”Draft”. There are ongoing debates if it is a feasible solution and probably its the reason why it was not officially implemented yet.

2.2.2 ERC-677 token interface

The ERC-677 token interface specifies a "transferAndCall Token Standard". Currently it is specified only in Ethereum's EIP Github repository, as the issue number 677 [7]. In addition to that also the state of the issue is at the moment of the writing of this report still on "open" and the state of the ERC is saved with status "Draft" as well (Oktober 2019). It is important to know that the status is not set to final and therefore changes might occur.

While ERC-223 is changing the behaviour of ERC-20's transfer method by adding the call to the `tokenFallback` method, ERC-677 [7] tries to combine the functionality of the ERC-223 without changing the behaviour of the ERC-20 token interface. The ERC-677 interface is an adapter to the ERC-20 interface and therefore fully compatible with it. This is achieved by adding the new functionality but not removing the old one, specifically the `approve` and `transferFrom` methods.

The ERC-677 token interface adds a the new function `transferAndCall` to the ERC-20 token contracts. `transferAndCall` can be called to transfer tokens to a contract and then, if the transfer is successful, call the `onTokenTransfer` method of this contract with the additionally provided data. Listing 2.5 shows that functionality is the same as in ERC-223. The `transferAndCall` method checks if the to-address is a contract and, if so, it calls the `contractFallback` method which calls the `ERC677Receiver` and its `onTokenTransfer` method. The purpose of `onTokenTransfer` is exactly the same as the one from `tokenFallback` in ERC-223. It was just renamed by the author of ERC-677.

Due to that renaming and the origin source of ERC-223, some implementations of ERC-677 call it `tokenFallback`, some call it `onTokenTransfer` and some others even differently.

```

1  pragma solidity ^0.4.11;
2
3
4  import "../token/ERC677.sol";
5  import "../token/ERC677Receiver.sol";
6
7
8  contract ERC677Token is ERC677 {
9
10     /**
11      * @dev transfer token to a contract address with additional data if
12      *      the recipient is a contract.
13      * @param _to The address to transfer to.
14      * @param _value The amount to be transferred.
15      * @param _data The extra data to be passed to the receiving contract.
16      */
17     function transferAndCall(address _to, uint _value, bytes _data)
18         public
19         returns (bool success)
20     {
21         super.transfer(_to, _value);
22         Transfer(msg.sender, _to, _value, _data);
23         if (isContract(_to)) {
24             contractFallback(_to, _value, _data);
25         }
26     }
27 }

```

```

25     return true;
26 }
27
28
29 // PRIVATE
30
31 function contractFallback(address _to, uint _value, bytes _data)
32     private
33 {
34     ERC677Receiver receiver = ERC677Receiver(_to);
35     receiver.onTokenTransfer(msg.sender, _value, _data);
36 }
37
38 function isContract(address _addr)
39     private
40     returns (bool hasCode)
41 {
42     uint length;
43     assembly { length := extcodesize(_addr) }
44     return length > 0;
45 }
46
47 }

```

Listing 2.5: ERC677Token.sol: A simplified example ERC-677 token contract [16].

2.2.3 ERC-865 token interface

The ERC-865 token interface tackles another problem of the ERC-20 tokens. When ERC-20 tokens should be used as utility tokens, the usability of a token becomes crucial. For example, if a token is going to be employed in a payment system for a decentralised platform, it is crucial that such platforms are straightforward and simple to use..

The authors of the ERC-865 [8] token interface want to make the user experience better by allowing the token holder to pay the transaction costs in tokens instead of Ether. The token holder can describe his/her **transferPreSigned** transaction and finally sign it with its private key. Instead of directly sending it to the BC, the token holder delegates the transaction to a third party which pays for the gas and takes a fee from the token holder instead.

Listing 2.6 shows the two relevant token contract functions. The method **transferPreSigned** is called by the third party which is paying for the gas (Charlie). The parameters are the **_signature** of the transaction from the token holder (Alice), the **_to address** (Bob's address) the transferred **_value** and the **_fee** which is paid from Alice to Charlie, because Charlie is paying the gas of Alice' transaction. In addition to that, a **_nonce** is passed which acts as "Presigned transaction number" and should be unique per user.

Alice hashes (using the keccak256 function) and signs her transaction data and passes its signature and the relevant parameters within the transaction to Charlie. Then, Charlie sends the transaction to the token contract. The **transferPreSigned** method reproduces the hash of Alice' transaction data with help of the **transferPreSignedHashing** method

(line 21 in listing 2.6). The relevant parameters are passed to this function and the keccak256 hash is calculated.

If the data hashed and signed by Alice and saved in the `_signature` parameter and the separately passed parameters were the same, the hashes reproduced by the `transferPreSignedHashing` method of the ERC-865 token contract and the one signed by Alice are exactly the same. If so, the token contract can now recover the signer address (Alice' address) by passing the reproduced hash and Alice' `_signature` to a recover method (line 22).

The recover method divides the signature in its `v`, `r` and `s` values and uses Solidity's `ecrecover` method to retrieve the signer's address. See Listing 4.4 in section 4.2.2 for more information.

The below implementation of the `transferPreSigned` method checks then if a an address was successfully recovered (line 23).

In line 25 and line 26 the method makes sure, that a transaction can only be sent once. This is also the reason why the `_nonce` parameter is used. If Alice wants to do the same transfer twice, she needs to still make sure, that the `_nonce` is different, otherwise the require clause in line 26 is failing. This assures that the signature is always different and that Charlie cannot decide by himself if he wants to send the transaction to the BC twice.

Line 28 to line 30 do the actual transfers. The `_value` is transferred to Bob (`_to address`, line 29) and the `_fee` is transferred to Charlie, represented by the `msg.sender` variable in line 30.

If the transfers were successful, the `hashedTx` mapping value is set to true, that this transaction can't be done a second time (line 31). The `Transfer` and `TransferPreSigned` events are emitted (line 33-35) before the function is returning.

```

1      /**
2       * @notice Submit a presigned transfer
3       * @param _signature bytes The signature, issued by the owner.
4       * @param _to address The address which you want to transfer to.
5       * @param _value uint256 The amount of tokens to be transferred.
6       * @param _fee uint256 The amount of tokens paid to msg.sender, by
7       *   the owner.
8       * @param _nonce uint256 Presigned transaction number. Should be
9       *   unique, per user.
10      */
11      function transferPreSigned(
12          bytes _signature,
13          address _to,
14          uint256 _value,
15          uint256 _fee,
16          uint256 _nonce
17      )
18      public
19      returns (bool)
20      {
21          require(_to != address(0));

```

```

21     bytes32 hashedParams = transferPreSignedHashing(address(this),
22         _to, _value, _fee, _nonce);
23     address from = recover(hashedParams, _signature);
24     require(from != address(0));
25
26     bytes32 hashedTx = keccak256(from, hashedParams);
27     require(hashedTxs[hashedTx] == false);
28
29     balances[from] = balances[from].sub(_value).sub(_fee);
30     balances[_to] = balances[_to].add(_value);
31     balances[msg.sender] = balances[msg.sender].add(_fee);
32     hashedTxs[hashedTx] = true;
33
34     Transfer(from, _to, _value);
35     Transfer(from, msg.sender, _fee);
36     TransferPreSigned(from, _to, msg.sender, _value, _fee);
37     return true;
38 }
39
40 /**
41  * @notice Hash (keccak256) of the payload used by transferPreSigned
42  * @param _token address The address of the token.
43  * @param _to address The address which you want to transfer to.
44  * @param _value uint256 The amount of tokens to be transferred.
45  * @param _fee uint256 The amount of tokens paid to msg.sender, by
46  *   the owner.
47  * @param _nonce uint256 Presigned transaction number.
48  */
49 function transferPreSignedHashing(
50     address _token,
51     address _to,
52     uint256 _value,
53     uint256 _fee,
54     uint256 _nonce
55 )
56 public
57 pure
58 returns (bytes32)
59 {
60     /* "48664c16": transferPreSignedHashing(address,address,address,
61         uint256,uint256,uint256) */
62     return keccak256(bytes4(0x48664c16), _token, _to, _value, _fee,
63         _nonce);
64 }

```

Listing 2.6: The on-chain operation methods from the official ERC-865 token interface

2.2.4 Comparison

Table 2.1 lists the discussed ERC token interfaces and its differences, advantages and disadvantages

Token Interface	Brings/Solves	Advantages	Disadvantages
ERC-20	- transferable tokens - makes tokens reusable in applications	- most popular interface, is supported a lot	- to transfer tokens to a SC for doing something, 4 transactions needed - approve/allowance vulnerability - tokens are lost when accidentally transferred to a SC
ERC-223	- tokenFallback method	- no lose of tokens when transferring to a SC	- not backwards compatible, changes ERC20 contract - failing method because of not implemented tokenFallback method, cost the user all provided gas
ERC-677	- transferAndCall method - combines ERC-20 and ERC-223	- compatible with ERC-20 - no lose of tokens - possibility to transfer to an SC and call a function of it	- service contract needs to implement onTokenTransfer method - only this method can be called
ERC-865	- transferPreSigned method - delegate the sending of the transaction	- transfer possible without having Ether for paying the gas	- different signing methods need to be considered - when having a third party doing the transaction, decentralization is lost

Table 2.1: Comparison of the ERC token interfaces 20, 223, 677 and 865.

Chapter 3

Design

This chapter presents the design of the service layer approach to pay for SC calls using tokens instead of Ethers. The designed architecture, the application flow and the planned functionalities are presented in the following sections.

3.1 Architecture Overview

Figure 3.1 illustrates the designed architecture of the service layer.

The service is visioned as decentralised service. This means that more than one should provide the service to be stable and assure availability. Since the states and the amount of saved data is designed to be low, the syncing between multiple machines is not a big challenge.

The data utilised by the service layer is saved in a generic database in the data storage component. This data is only to make the user experience more convenient and is not really necessary for a functional service layer. It includes transaction states and receipts or error messages of past transactions. The data is a subset of the data accessible on Etherscan [12].

The service layer is connected to the Ethereum BC. Each server is connected to its own Ethereum full node. The communication between the server and the full node is done with JSON-RPC [15].

Within the servers the JSON-RPC protocol is wrapped into Ethereum interface depending which language the server is written in. In addition, the servers are able to call the full nodes also with plain JSON-RPC without using any Ethereum interface.

The service layer provides a REST API. This REST API exposes the relevant functionality to the outside world. This includes the following features:

- **Delegated transfer:** By calling this endpoint, the service transfers tokens from one address to another and takes a fee for paying the gas. After sending the transaction to the BC, the service response with the transaction hash.

- **Delegated transfer and call:** By calling this endpoint, the service transfers tokens from one address to a SC and if this transfer is successful, a method of this SC is called automatically. The service takes a fee for paying the gas. After sending the transaction to the BC, the service response with the transaction hash.
- **Get transaction:** This endpoint is requested with a transaction hash parameter and returns relevant information about this transaction; if it is still pending, already mined and if the transaction execution was successful or the EVM state was reverted.
- **Get transactions:** The `getTransactions` endpoint returns a list of transaction objects with the same information as mentioned above

The REST API can be integrated with any frontend client. The only requirement for the client is, that it needs to be able to sign transactions. Moreover, it must need to know which sign transaction to use and the format of the to be signed transaction.

For signing the transaction the user's private key is used. Therefore a private key handler is required. The service layer designed is agnostic to the private key handling service. Thus, the service can be integrated with any private key handling solution defined by the user. The service should be accessible and usable by any user with a private key handling solution of its choice.

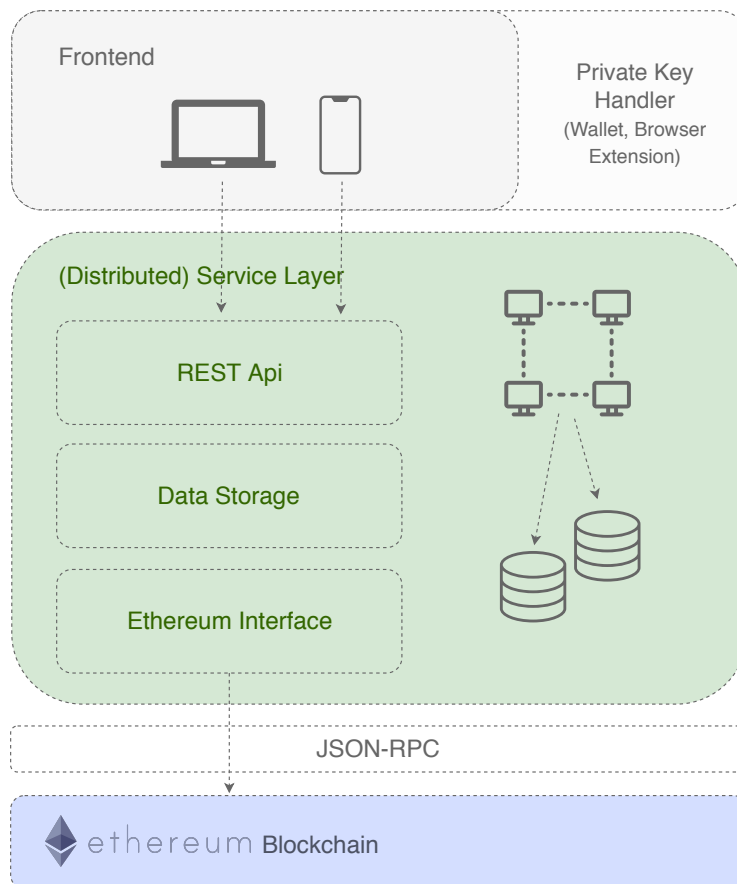


Figure 3.1: The designed service architecture

3.2 Application Flow

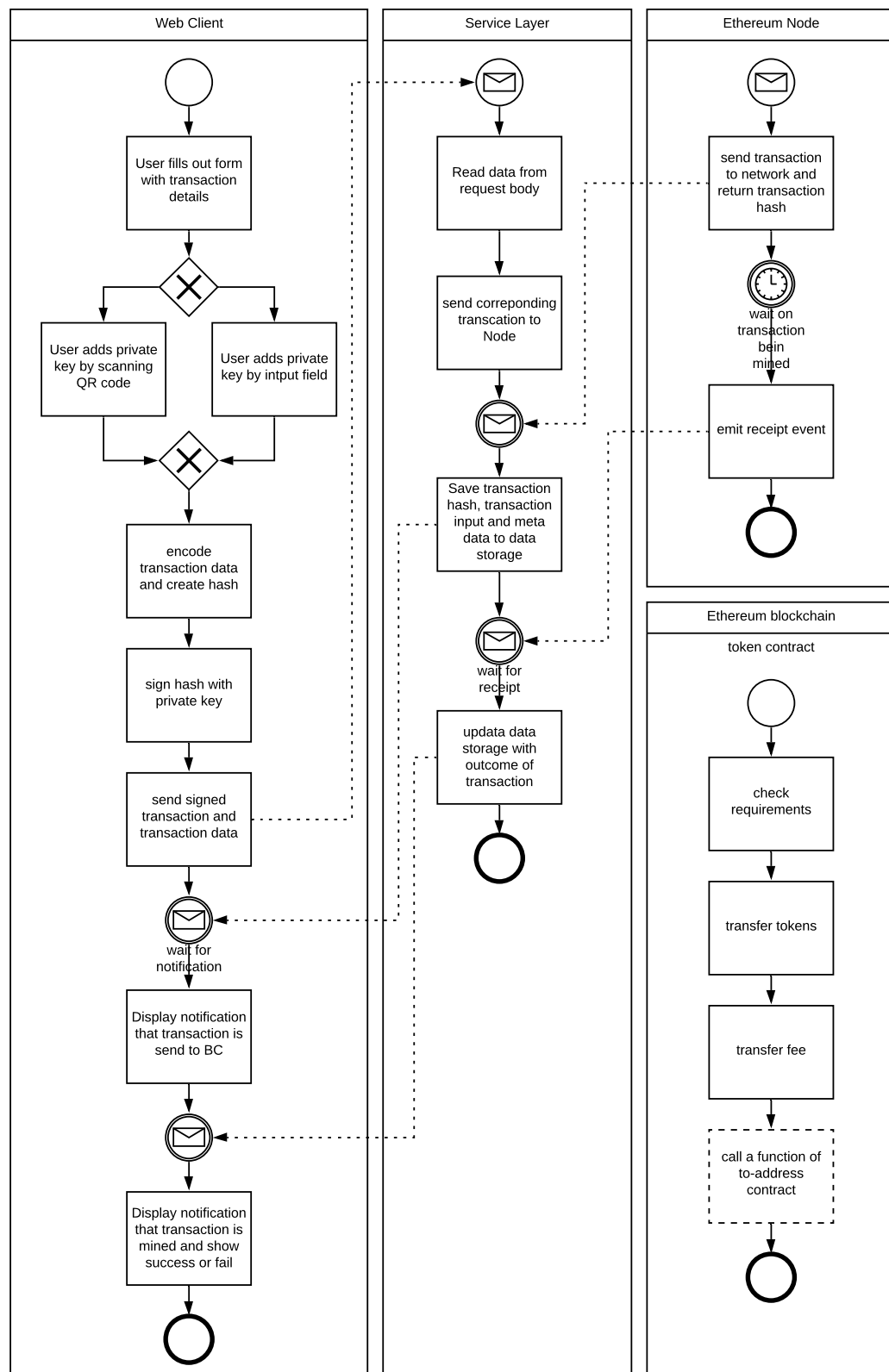


Figure 3.2: The process flow model of the application and its components.

Listing 3.2 represents a basic process model diagram from the whole application. There are basically three components communicating with each other by sending JSON objects to each other.

The user interfaces the web client only and fills the forms with its transaction data. The user adds the private-key to the frontend component by either scanning the QR code or filling it into the input field. When the clicks the Send-Button, the user's data is encoded and hashed, signed with the private key and then send to through the REST API to the Service Layer backend. The service layer then reads the data and sends the transaction with a JSON-RPC call to the underlying Ethereum node which sends the transaction to the network. The node immediately receives the transaction hash and sends it back to the server. The server saves the transaction hash, together with transaction input and other meta data to the storage and returns the transaction hash to the web client where a "successfully sent transaction" notification is displayed.

Once the transaction is mined and the node receives the new block, it emits a receipt event. Because the server is listing on this event, it can react to it: It updates the transaction data in the data storage with the transaction status. The web client is informed about the transaction update and displays the second notification to the user.

3.3 Frontend Functionalities

The frontend is kept simple but user-friendly. The functionalities are stripped to minimum besides the possibility to look up the result of the sent transaction. If the user stays on the same page, he or she is also informed about the outcome of the transaction by a simple notification feature.

The frontend has the following three tabs:

- **Transfer tokens:** To transfer tokens only
- **Call a function:** To transfer tokens to a SC and call a function of it
- **My transactions:** Lists the transactions already made from this browser

In the "Call a function" tab, the user can choose from the functions he is able to call on the SC. In addition to that, when the function is chosen he can directly specify the parameter values for this function.

To improve the usability of the service even more, the user can scan a QR code which represents either the private key or with even more information which automatically fills the relevant fields in the frontend.

Chapter 4

Implementation

In this chapter the implementation of the service layer is documented. It starts with the implemented architecture and what differs to the designed architecture. Then it is shown how the service can be configured in Section 4.3. In Section 4.4, the implementation of the user interface and the signing functionalities are displayed. Section 4.5 documents the service layer respectively the first part of the backend. The relevant information of the second part of the backend, the Ethereum BC full node, is presented in Section 4.6.

4.1 Implemented Architecture

A diagram of the implemented architecture can be seen in Figure 4.1. One can see that the implementation is very close to the proposed design in Section 3.1. For implementing, testing and evaluating the service, it is not necessary to have a distributed service. Therefore it was decided to reduce the complexity by implementing a single server. This resulted in a simple server-client architecture with one difference to "normal" server-client applications: In this architecture, the server is connected to a full node of the Ethereum BC.

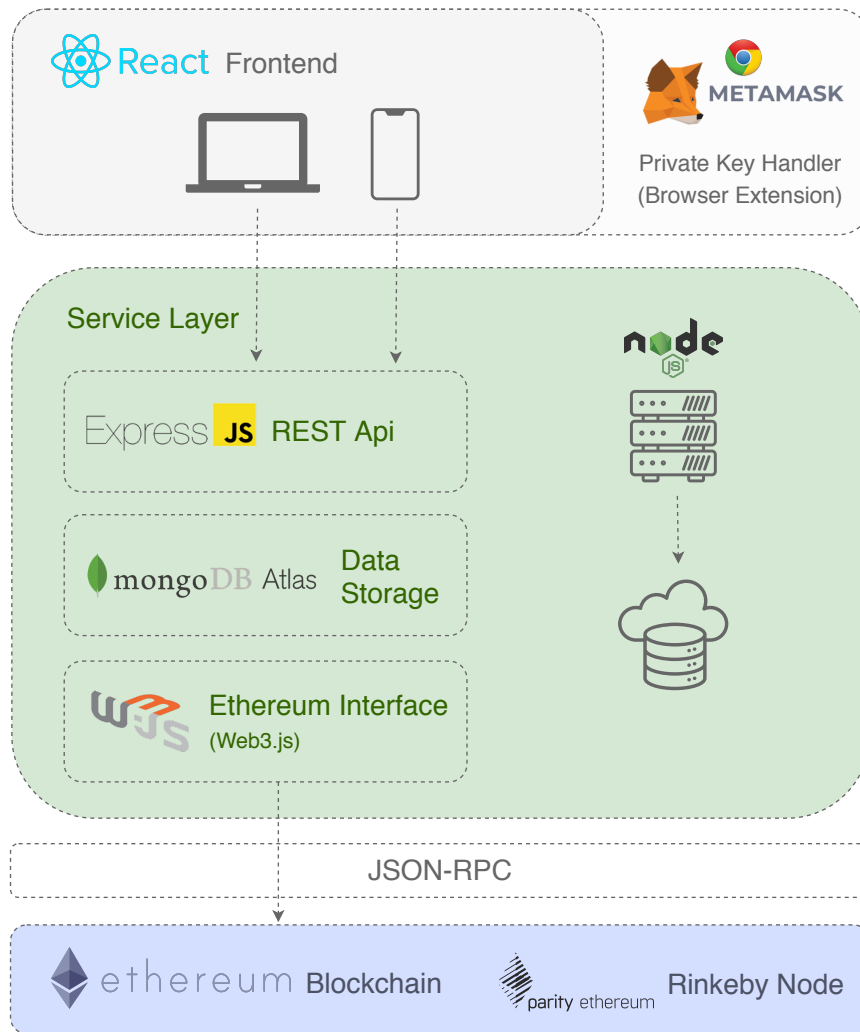


Figure 4.1: The implemented service architecture

The frontend is written in ReactJS [20], a frontend library to build user interfaces. Besides that the frontend is optionally connected with Metamask, a Chrome browser extensions to handle Ethereum accounts.

The service layer is set up as a Node.js server. The React frontend and the server communicate via a JSON REST Api implemented with Express. Express is a minimalist web framework for Node.js. It eases the implementation of Node.js applications and middlewares such as an HTTP REST Api.

For storing the data of sent transactions and also service configuration data, a cloud data storage is used: mongoDB Atlas provides document databases available on AWS, Azure or Google Cloud Platform.

The communication between the service layer and the Ethereum full node is done with JSON-RPC calls. The server uses the Web3.js framework. It basically wraps the most used JSON-RPC functions into JavaScript methods. Moreover, it includes some useful utils function.

The server is connected to an Ethereum full node on the Rinkeby Testnet. During development the Parity Development chain was used.

4.2 Use Case and Example Smart Contracts

Every component of the service layer is depending on the implementation of the SCs, token contract as well as service contract. Therefore, a representative use case and examples of contracts are presented in this Section. Important to note is that this is a proposed implementation to use all functionalities of the service layer. The SCs can also be implemented differently.

4.2.1 Use Case

The example use case is an online translation service where users can request translations. Alice can lock funds in form of tokens on a service contract together with a URL pointing to something which needs to be translated. Bob can translate it and submit another URL with the translated text. As soon as a translation is submitted, Alice cannot withdraw her request and the money anymore. If she accepts the translation, she can unlock the funds for Bob. The funds need to be paid in a stable coin backed with Swiss Francs, the so called Swiss Franc Token (CHFT). The advantages of such a stable coin compared to Ethereum is the reduced volatility since the token is connected to the Swiss Francs Price.

4.2.2 The CHFT, the Swiss Franc Token contract

The Swiss Franc Token implements the ERC-20 interface as well as a customised `ERC865-Plus677ish` interface. This means the CHFT SC implements all functions required by the ERC-20 interface, such as `transferFrom` and `balanceOf`, as well as methods from the ERC-865 and ERC-677 interfaces. The `ish` in the interface name states that the implementation differs from the official implementations of ERC-865 and ERC-677 described in Section 2.2.

Listing 4.1 shows the beginning of the CHFT contract. It is written in Solidity 0.5 and imports the SafeMath library and some necessary functions. Some necessary functions are described in the following Sections and chapters.

The implementation of line 46 guarantees that the CHFT contract implements all the functions of the two interfaces.

The differences to the officially described ERC-865 and ERC-677 interfaces are also stated in the contract in the comment Section in line 18 to line 28. The `transferAndCall` method is generic, that means no `transferFallback` named method is called after the transfer, instead the message sender can specify the `_methodName` with the third parameter (see line 33) and the parameters of specified function with the `_args` bytes parameter. The

first parameter specifies the SC address to call and the second parameter the `_value` to be transferred to that SC.

Because of the combination of the two token interfaces, a new `transferAndCallPreSigned` method is required (line 42 in Listing 4.1). It enables the service layer or any other third party to not only use the transfer but also the `transferAndCall` functionality. Therefore the implementation also deviates from the ERC-865 interface.

In the CHFT contract itself, three mappings are implemented. The balances and the two-dimensional allowed mapping in line 53 and 54 are coming from the ERC-20 interface. The allowed mapping specifies which account can dispose of which amount from another account. The signatures mapping is really important for the security of the contract. It saves if a signature was already used and the check against this array makes sure that every signature is only used once and therefore that every transaction can only be sent once.

```

1  pragma solidity ^0.5.11;
2
3  import "./SafeMath.sol";
4  import "./Utils.sol";
5
6  contract ERC20 {
7      function allowance(address owner, address spender) public view
          returns (uint256);
8      function transferFrom(address from, address to, uint256 value)
          public returns (bool);
9      function approve(address spender, uint256 value) public returns (
          bool);
10     function totalSupply() public view returns (uint256);
11     function balanceOf(address who) public view returns (uint256);
12     function transfer(address to, uint256 value) public returns (bool);
13     event Approval(address indexed owner, address indexed spender,
          uint256 value);
14     event Transfer(address indexed from, address indexed to, uint256
          value);
15 }
16
17 /**
18  * @title ERC677 transferAndCall token interface
19  * @dev See https://github.com/ethereum/EIPs/issues/677 for
          specification and
20  * discussion.
21  *
22  * We deviate from the specification and we don't define a tokenfallback
          . That means
23  * tranferAndCall can specify the function to call (bytes4(sha3("setN(
          uint256)")))
24  * and its arguments, and the respective function is called.
25  *
26  * If an invalid function is called, its default function (if
          implemented) is called.
27  *
28  * We also deviate from ERC865 and added a pre signed transaction for
          transferAndCall.
29  */

```

```

30
31 contract ERC865Plus677ish {
32     event TransferAndCall(address indexed _from, address indexed _to,
33         uint256 _value, bytes4 _methodName, bytes _args);
34     function transferAndCall(address _to, uint256 _value, bytes4
35         _methodName, bytes memory _args) public returns (bool);
36
37     event TransferPreSigned(address indexed _from, address indexed _to,
38         address indexed _delegate,
39         uint256 _amount, uint256 _fee);
40     event TransferAndCallPreSigned(address indexed _from, address
41         indexed _to, address indexed _delegate,
42         uint256 _amount, uint256 _fee, bytes4 _methodName, bytes _args);
43
44     function transferPreSigned(bytes memory _signature, address _from,
45         address _to, uint256 _value,
46         uint256 _fee, uint256 _nonce) public returns (bool);
47     function transferAndCallPreSigned(bytes memory _signature, address
48         _from, address _to, uint256 _value,
49         uint256 _fee, uint256 _nonce, bytes4 _methodName, bytes memory
50         _args) public returns (bool);
51 }
52
53 contract CHFT is ERC20, ERC865Plus677ish {
54     using SafeMath for uint256;
55
56     string public constant name = "CHF Token";
57     string public constant symbol = "CHFT";
58     uint8 public constant decimals = 6;
59
60     mapping(address => uint256) balances;
61     mapping(address => mapping(address => uint256)) internal allowed;
62     // nonces of transfers performed
63     mapping(bytes => bool) signatures;
64
65     uint256 public totalSupply_;

```

Listing 4.1: The beginning of the Swiss Franc Token contract which implements the ERC-20 and the ERC865Plus677ish interface.

The Pre-Signed and Delegated Transfer-and-Call

Since the `transferPreSigned` method is already described in Section 2.2.3, it is focused here on the delegated `transferAndCall` method: `transferAndCallPreSigned`. The implementation of `transferPreSigned` is very similar.

Listing 4.2 shows the `transferAndCallPreSigned` method, how it is implemented in CHFT. After a check if the signature is not already used (line 4), the hash of the transaction input is generated. Listing 4.3 shows how the hash is generated. The exact same hash needs to be generated during the sign process in the frontend (see Section 4.4.2). Only if the hashes are the same the signer address (or from address) can be retrieved.

How the signer address is retrieved can be seen in Listing 4.4. `Utils`' `recover` method splits the 65 bytes long signature (`sig`) into its three components `r`, `s` and `v` (line 15-19). Solidity's `ecrecover` method takes care of the rest and returns the signer address when passed the hash, `v`, `r` and `s` (line 39).

In between these lines the vulnerability of allowing signature malleability is avoided: If one would allow for the value `v` both 0/1 and 27/28, one would allow two signatures resulting in the same valid public key. Signatures with `v` equal 0/1 and signatures with 27/28 would both be valid and would open double spending attack possibilities. Therefore and because for example `geth` is transforming 0/1 to 28/27 values [14], the `recover` method only accepts the `v` values 27 and 28 (line 34). In addition to that, only the lower half order of the `s`-value range is accepted (line 30). For more information about the signature malleability, see Thomas Bocek's comment in the OpenZeppelin implementation (lines 21-29, Listing 4.4) [19].

During the implementation of the service layer, one figured out, that when the two hashes are not the same, Solidity's `ecrecover` method returns a random address instead of the expected 0x0-address. Although the chances are really small, that the CHFT token balance of a random address is greater than 0 (the tokens would be deducted from this balance), it was decided to pass the `_from` address as parameter and add an additional check if the retrieved `from` address is equal to the `_from` parameter (line 11 in Listing 4.2).

If this check is successful, the transfer is triggered and the signature boolean is set to true to avoid sending the same transaction twice (line 14-15). If the receiver address (`_to`) is a SC, the in `_methodName` specified function is called with `from` and `_value` as well as the `_args` as parameters (line 25).

```

1  function transferAndCallPreSigned(bytes memory _signature, address _from
    , address _to, uint256 _value, uint256 _fee, uint256 _nonce,
2      bytes4 _methodName, bytes memory _args) public returns (bool) {
3
4      require(!signatures[_signature]);
5      bytes32 hashedTx = Utils.transferAndCallPreSignedHashing(address(
        this), _to, _value, _fee, _nonce, _methodName, _args);
6      address from = Utils.recover(hashedTx, _signature);
7
8      // if hashedTx does not fit to _signature Utils.recover resp.
        Solidity's ecrecover returns another (random) address,
9      // if this returned address does have enough tokens, they would be
        transferred, therefor we check if the retrieved
10     // signature is equal the specified one
11     require(from == _from);
12     require(from != address(0));
13
14     doTransfer(from, _to, _value, _fee, msg.sender);
15     signatures[_signature] = true;
16
17     emit Transfer(from, _to, _value);
18     emit Transfer(from, msg.sender, _fee);
19     emit TransferAndCallPreSigned(from, _to, msg.sender, _value, _fee,
        _methodName, _args);
20
21     // call receiver

```

```

22     require(Utils.isContract(_to));
23
24     //call on behalf of from and not msg.sender
25     (bool success, bytes memory response) = _to.call(abi.encodePacked(
26         abi.encodeWithSelector(_methodName, from, _value), _args));
27     require(success);
28     return success;

```

Listing 4.2: The `transferAndCallPreSigned` method from the CHFT contract.

```

1 function transferAndCallPreSignedHashing(address _token, address _to,
2     uint256 _value, uint256 _fee, uint256 _nonce,
3     bytes4 _methodName, bytes memory _args) internal pure returns (
4     bytes32) {
5     /* "38980f82": transferAndCallPreSignedHashing(address,address,
6         uint256,uint256,uint256,bytes4,bytes) */
7     return keccak256(abi.encode(bytes4(0x38980f82), _token, _to, _value,
8         _fee, _nonce, _methodName, _args));
9 }

```

Listing 4.3: The hashing function which is saved in `Utils.sol` and used in the CHFT contract.

```

1 function recover(bytes32 hash, bytes memory signature) internal pure
2     returns (address) {
3     // Check the signature length
4     if (signature.length != 65) {
5         return (address(0));
6     }
7
8     // Divide the signature in r, s and v variables
9     bytes32 r;
10    bytes32 s;
11    uint8 v;
12
13    // ecrecover takes the signature parameters, and the only way to get
14    // them
15    // currently is to use assembly.
16    // solhint-disable-next-line no-inline-assembly
17    assembly {
18        r := mload(add(signature, 0x20))
19        s := mload(add(signature, 0x40))
20        v := byte(0, mload(add(signature, 0x60)))
21    }
22
23    // EIP-2 still allows signature malleability for ecrecover(). Remove
24    // this possibility and make the signature
25    // unique. Appendix F in the Ethereum Yellow paper (https://ethereum.github.io/yellowpaper/paper.pdf), defines
26    // the valid range for s in (281):  $0 < s < \text{secp256k1n} \cdot \frac{1}{2} + 1$ , and
27    // for v in (282):  $v \in \{27, 28\}$ . Most
28    // signatures from current libraries generate a unique signature
29    // with an s-value in the lower half order.
30
31    //
32    // If your library generates malleable signatures, such as s-values
33    // in the upper range, calculate a new s-value

```

```

27      // with 0
      xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAEDCE6AF48A03BBFD25E8CD0364141
      - s1 and flip v from 27 to 28 or
28      // vice versa. If your library also generates signatures with 0/1
      for v instead 27/28, add 27 to v to accept
29      // these malleable signatures as well.
30      if (uint256(s) > 0
      x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0
      ) {
31          return address(0);
32      }
33
34      if (v != 27 && v != 28) {
35          return address(0);
36      }
37
38      // If the signature is valid (and not malleable), return the signer
      address
39      return ecrecover(hash, v, r, s);
40 }

```

Listing 4.4: The recover method in `Utils.sol`. This is the official implementation and comments of OpenZeppelin [19].

4.2.3 The Service Contract: a Simple Translation Service

For demonstration purposes a simple service contract was implemented. A user can request a translation by calling the `transferAndCall` function of the CHF Token contract. The method signature of the `requestTranslation` method needs to be specified. The user can specify a reward of its choice. This amount is then sent to the translation service contract.

Listing 4.5 shows the translation service contract.

When the user requests a translation, he/she passes the URL where the original document is saved. In the `requestTranslation` method the keccak256 hash of the URL is computed and used as unique identifier. The `_requester` address, the `_reward` and the `requestBlockNumber` are then saved to the `TranslationRequest` object in the requests mapping. The `requestTranslation` function can only be called via the CHF Token contract since only with the `transferAndCall(PreSigned)` method it can be made sure that the transfer happened before the request.

Another user is handing in a translation by specifying the `_originalUrl` as well as the URL where the translation is saved (`_translationUrl`). The `translationSubmission` method can be called directly from the user or via the token contract because there is no required transfer beforehand.

The service contract contains also a `withdrawRequest` and a `requestImprovement` method, which can be called by the requester. The methods were removed from the listing below for the sake of simplicity.

Once the translator submitted the translation, the requester has three days to request an improvement if he is not accepting the translation. The translator can collect the reward (`collectReward` function on line 57) once his submission has been more than three days and no improvement request is outstanding. The blocks to wait before the translator can collect the reward are calculated with an average blocktime of 13 seconds. On the Ethereum BC, around 19938 blocks are mined in three days (line 23). If the reward is collected the collected boolean is set to true and the reward is transferred to the translator's address (line 67-68).

One can see that for the methods `translationSubmission` and `collectReward` (and also the ones not listed), the second parameter is always the `_value` parameter although it is not needed. This is because the generic `transferAndCall` and `transferAndCall-PreSinged` methods from the token contract (see line 25 in Listing 4.2) always pass the message sender and the transferred value as first and second parameter. Therefore, to make them callable by the token contract respectively by the service layer, the first two parameters need to be kept.

```

1  pragma solidity ^0.5.12;
2  import "./CHFT.sol";
3
4  contract TranlationService {
5
6      address contractOwner;
7      CHFT CHFTContract;
8
9      struct TranslationRequest {
10         address requester;
11         uint256 reward;
12         uint requestBlockNumber;
13         address translator;
14         string translation;
15         uint256 handinBlockNumber;
16
17         bool rewardCollected;
18     }
19
20     mapping (bytes32 => TranslationRequest) requests;
21
22     // the requester has around 3 days to request an improvement, after
23     // these 3 days, the translator can collect the reward
24     // in blocks, avg blocktime 13 seconds -> 60 * 60 * 24 * 3 / 13secs
25     // = 19938 blocks
26     uint256 timeForImprovementRequest = 19938;
27
28     constructor() public {
29         contractOwner = msg.sender;
30     }
31
32     function setCHFTContract(address _chftAddress) public {
33         require(msg.sender == contractOwner);
34         CHFTContract = CHFT(_chftAddress);
35     }
36
37     function requestTranslation(address _requester, uint256 _reward,

```

```

string memory _originalUrl) public {
36
37     // make sure that the amount was really transferred
38     // this function can only be called by the token contract to
        make sure that the _reward is always transferred
39     require(msg.sender == address(CHFTContract));
40
41     TranslationRequest storage request = requests[keccak256(bytes(
        _originalUrl))];
42     request.requester = _requester;
43     request.reward = _reward;
44     request.requestBlockNumber = block.number;
45 }
46
47 [...] // withdrawRequest and requestImprovement
48
49 function translationSubmission(address _translator, uint256 _value,
    string memory _originalUrl, string memory _translationUrl) public
    {
50     // this function can be called either via the token contract or
        directly since the _value should be always 0
51     require(msg.sender == address(CHFTContract) || msg.sender ==
        _translator);
52
53     TranslationRequest storage request = requests[keccak256(bytes(
        _originalUrl))];
54
55     require(request.translator == address(0) || request.translator
        == _translator, "already translated by someone else");
56
57     request.translator = _translator;
58     request.translation = _translationUrl;
59     request.handinBlockNumber = block.number;
60
61 }
62
63 function collectReward(address _collector, uint256 _value, string
    memory _originalUrl) public {
64     // this function can be called either via the token contract or
        directly since the _value should be always 0
65     require(msg.sender == address(CHFTContract) || msg.sender ==
        _collector);
66
67     TranslationRequest storage request = requests[keccak256(bytes(
        _originalUrl))];
68
69     require(_collector == request.translator, "only the translator
        can collect the reward");
70     require(!request.rewardCollected, "the reward is already
        collected");
71
72     require(request.requestBlockNumber < request.handinBlockNumber,
        "the reward can only be collected if there is no improvement
        request outstanding");
73     require(block.number - timeForImprovementRequest >= request.
        handinBlockNumber, "the reward can only be collected if the

```

```

74         time for a request has expired");
75         request.rewardCollected = true;
76         require(CHFTContract.transfer(request.translator, request.reward
77             ));
78     }
79 }

```

Listing 4.5: The recover method in Utils.sol. This is the official implementation and comments of OpenZeppelin [19].

4.3 Service Configuration

The service includes a configuration file which is shared between frontend and backend. The configuration file is important for the usability of the service. It allows the service to be adaptable without a new installation and the frontend to be user friendly, since the user does not need to specify token addresses for example. The configuration file is saved in the data storage and specifies which ERC-865 tokens are served by the service. Moreover, one can specify which SCs can be called from the service. Most of the configuration information is used by the frontend to extract the methods which can be called and to show a dropdown of different transferable ERC-865 tokens. But also the backend uses the configuration file to know the destination of the Ethereum node and which account to send the transactions from for example.

Listing 4.6 shows the configuration JavaScript object with one callable token contract and one service contract which can be called with the "Call a function" feature. Note that the `acceptedTokens` and the `serviceContracts` array are shortened to one object for readability. It is obviously meant to be, that the arrays have more than one object.

```

1  {
2    "unlockedServiceAccount": "0x8d68e1ae44513b895ba3a430e4d60c305269f22a"
3    ,
4    "ethereumNetwork": "rinkeby",
5    "ethereumFullNodeAddress": "127.0.0.1:8546",
6    "currentEthUsdPrice": 145.22,
7    "acceptedTokens": [
8      {
9        "address": "0x64B957C8912Edd9315a6Da36aEd04585777cB08E",
10       "name": "CHF Token",
11       "symbol": "CHFT",
12       "feeTransfer": 1,
13       "feeTransferAndCall": 2,
14       "defaultTokenToEthPrice": 0.0001,
15       "abi": [...],
16       "signMethod": "secp256k1Sign"
17     }
18   ],
19   "serviceContracts": [
20     {
21       "address": "0x2bfEcE4263bce0cbF2364D3c26b60ee153a440c1",
22       "name": "TranslationService",

```

```
22     "abi": [...]  
23   },  
24 ]  
25 }
```

Listing 4.6: A sample configuration JavaScript object shared between the frontend and the backend of the service.

The configuration file starts with some global service settings in line 2 to 5. The service uses one specific address which the delegated transactions are sent from. This address is specified with the `unlockedServiceAccount` field. As the name states, this address needs to be unlocked already. For future usage of the service, it should be made sure, that the account does not always need to be unlocked (see Chapter 6 for more information).

Line 3 and 4 are related to the Ethereum network. The field `ethereumNetwork` is used within code to differentiate the networks and handle it accordingly. `ethereumFullNodeAddress` specifies the destination of the Ethereum full node used for the service connecting to the Ethereum BC.

Line 5 (`currentEthUsdPrice`) specifies the ETH/USD price when last updated. This is used by the frontend to calculate and tell the user the fee price in US dollar instead of tokens or Ether. In future versions of the service, this price can be pulled dynamically from Etherscan.io or other BC explorers or cryptocurrency exchanges.

The `acceptedTokens` array specifies which ERC-865 tokens are accepted with this service. For each object the following fields need to be specified:

- **address:** The SC address where the token contract is deployed
- **abi:** The ABI (application binary interface) of the token contract. It is used for creating the contract object in the Ethereum interface to eventually send the transactions to the BC.
- **name:** The name of the token. This is used for displaying it in the frontend.
- **symbol:** The token symbol, also used in the frontend.
- **feeTransfer:** This specifies the fee which the service receives for calling the `transferPreSigned` method.
- **feeTransferAndCall:** This specifies the fee which the service receives for calling the `transferAndCallPreSigned` method.
- **defaultTokenToEthPrice:** This field is only for showing the user in the frontend an estimated Ether or USD equivalent of the paid fee.
- **signMethod:** The `signMethod` attribute is necessary for the frontend to know which method the data should be signed with. Different ERC-865 tokens could be recovering the signature with different methods. That's why it is important to sign the data with the correct method. More information about the signing the transaction in Section 4.4.2.

On line 18 of Figure 4.6, the `serviceContracts` array is specified. It includes all SCs which the frontend supports for the "Call a function" feature. That does not mean that other SCs cannot be called. The frontend can just only support with callable function names and method parameter input fields, when it knows the ABI of the SC.

Objects in the `serviceContracts` array need to include the address of the SC and the name to display in the dropdown menu. Also the ABI need to be specified. With the `abi` field being provided, the frontend is able to display the callable methods and its parameters.

4.4 Frontend

The frontend is implemented in ReactJS. It is a web application accessible from most browsers which supports JavaScript. The application has all requirements to be accessible from mobile phone browsers as well. However, the current state of the frontend application is optimised for desktop browsers meaning not everything is tested and optimised for mobile devices.

The frontend is deployed to Heroku and is reachable under <https://pay-gas-service.herokuapp.com/>.

4.4.1 User Interface

The first goal of the user interface was its usability and the simplicity of the application. But it was also assumed that some future use cases will only use the service layer and implement their own frontend, therefore it was made sure, that the functionalities which a frontend needs to implement are clear and can easily be copied by others.

The web application can be used without any login. When the web application is started, one is right where the action happens, on the transfer token form. The user interface has three navigation pills: "Transfer tokens", "Call a function" and "My transactions".

Transfer tokens

Figure 4.2 shows the start page. The "Transfer tokens" tab is selected and the user can just start filling the form. The user enters the amount and chooses from the dropdown what tokens the service layer should transfer. After that, one can fill in the from-address and the to-address. Below the two input fields, the transaction fee is displayed and converted into Ether and US dollar. Not only the fee but also the total amount of tokens, which the service will deduct from the token balance are reported.

If the user is logged into MetaMask, a popup appears which asks for permission to access the MetaMask accounts. If the user clicks "Connect", the account address is automatically added to the from-address.

ERC-865 pay gas service

Transfer tokens

Call a function

My transactions

SCAN QR CODE INSTEAD OF FILLING THE FORM

400

CHF Token

From:

From Address

To:

To Address

Transaction fee:

50 CHFT

≈ 0.3381345 ETH

≈ 50.19 USD

Token total:

450 CHFT

≈ 3.0432 ETH

≈ 451.7 USD

Private key of the from address

Your private key is only used to sign the entered transaction data. It is neither stored nor sent somewhere.

SEND

Figure 4.2: The "Transfer tokens" tab empty, just after startup of the application

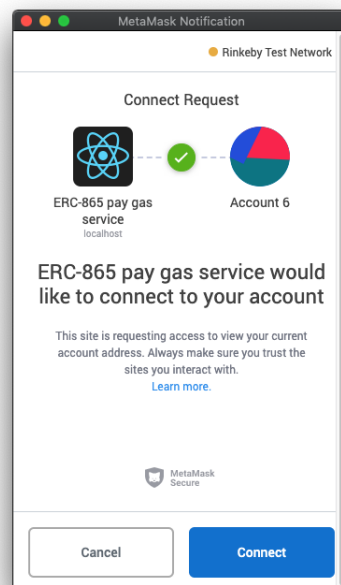


Figure 4.3: The popup which is prompted when the user is logged into MetaMask.

To sign and send the transaction, the user needs to provide its private key. The first and easiest way to do that is entering it into the input field. Below the input field is stated that the private key is only used to sign the transaction data and that it is not stored nor send somewhere. While this is very important information for the user, it does not give any certainty about the trustworthiness of such a service.

A second possibility to add the private key is to scan the QR code representation of the private key. It will add the private key value to the state of the application component. As soon as the component is unmounted (refreshing the page or changing the tab), the private key value is lost again.

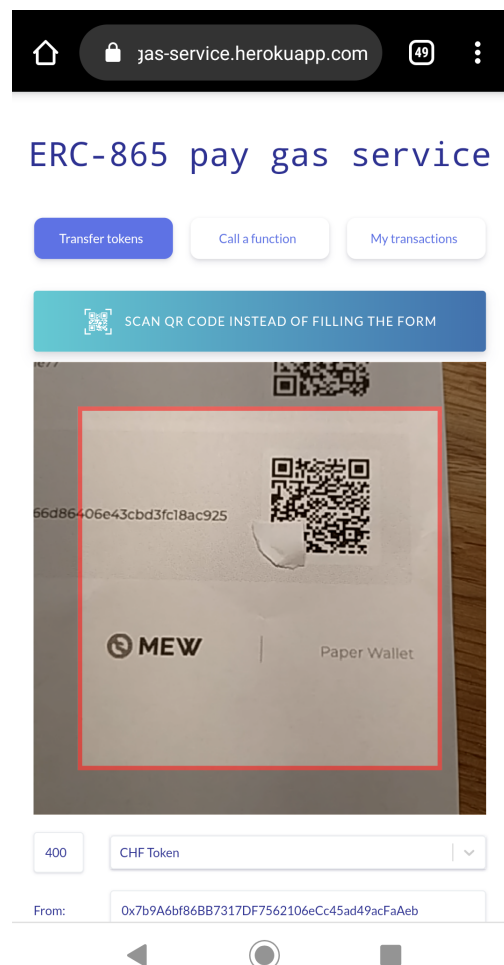


Figure 4.4: The QR code section opens by clicking on the "Scan QR code..." button.

Figure 4.4 shows the QR code feature. One can see that the application can also be used on mobile phones. As soon as the browser has permission to use the camera, it opens and the user can hold the QR code against the camera. The QR code can either contain only the private key (for example from MyEtherWallet, as it is on the figure) or it can include the data to fill all the fields in the form.

Listing 4.7 shows a JavaScript object which can be encoded in a QR code. When scanning

a QR code with such an object, the service frontend will detect the information and automatically fill the fields. The private key field will be hided.

```

1 {
2   pk: "95FE3783808009AFDA9A614D46511E304FD435C7E0ECE24A52E20D0A16C50C8F",
3   tokenAddress: "0xB5C9Fad1b0b108116c3b35311480f3d77cA4A5aC",
4   to: "0x8203DE77aaA67ae7fA79b332Cb15A7826B611999",
5   methodName: "0xdd5c6fb9",
6   value: 500
7 }

```

Listing 4.7: The JavaScript object which can be encoded in a QR code. By decoding it with the QR code scanner feature of the frontend, the data is filled into the form.

When scanning such a QR code, the frontend will automatically switch to the "Call a function" view. It checks (also by entering the to-address manually), if the to-address is a SC accepted by this service layer. If so, it passes the data to the second view.

Call a function

The "Call a function" view can either be opened by the user itself or the user can be forwarded to it by entering (or scanning a QR code with) a to-address that is a SC address supported by the service layer. Supported is a SC when it is mentioned in the `serviceContracts` array of the configuration file (see section 4.3).

The user can also choose from all supported respectively callable SCs via the drop-down menu at "Contract Name". In the "Contract Function" drop-down, all callable function of the chosen contract will be listed and the user can select one of them.

As soon as the contract name is selected, the to-address field is filled too. When entering a known SC address, the same happens with the contract name selection field.

Figure 4.5 shows the Contract Function drop-down. The methods callable via the `transferAndCall` function of the token contract are extracted from the SC ABI of the configuration file.

Note that this functionality is only available when the token contract has a generic `transferAndCall` method as specified in the example token contract in section 4.2.2. In other token contracts where the `transferAndCall` method calls a fixed defined `tokenFallback` method for example (see listing 2.5 in section 2.2.2 as reference), the function selection feature is obviously not needed.

Figure 4.5: The "Call a function" tab with the function drop-down of the Translation Service contract.

Listing 4.8 shows how the callable SC functions are extracted: The ABI from the contract object is parsed. All non-constant objects with type "function" are methods which can be called. In addition to that, only the ones with the first method parameter of type "address" and the second parameter of type "uint256" are possible to call from the generic transferAndCall function. In listing 4.2 on line 25, one can see, that the first two parameters are always the `_from`-address and the `_value` passed from the transferAndCall method.

For SCs with non-generic transferAndCall methods or for not supported SCs (when the contract ABI and therefore the callable methods are unknown) the user needs to enter the method input data as a hex formatted string.

```

1  getCallableMethods(selectedServiceContract) {
2    let methods = [];
3    selectedServiceContract.value.contractObj._jsonInterface.forEach(e
    => {
4      if (this.isCallableMethod(e))
5        methods.push({
6          value: e,
7          label: e.name
8        });
9    });
10   return methods;
11 }
12

```

```

13  isCallableMethod(e) {
14      return (
15          e.type === "function" &&
16          e.constant === false &&
17          e.inputs.length >= 2 &&
18          e.inputs[0].type === "address" &&
19          e.inputs[1].type === "uint256"
20      );
21  }

```

Listing 4.8: The extraction of the methods which can be called from the generic `transferAndCall` function of a modern ERC-865 token.

Contract Name: TranslationService

Contract Function: translationSubmission

Method Parameters

_originalUrl: https://files.axelra.com/original_text.txt

_translationUrl: https://csg.uzh.ch/translated_text.txt

Transaction fee: 2 CHFT
 ≈ 0.01352538 ETH
 ≈ 0.20 USD

Private key of the from address

Your private key is only used to sign the entered transaction data. It is neither stored nor sent somewhere.

SEND

Figure 4.6: The input fields for method parameters of the calling functions.

My transactions tab

The My transactions tab shows the transactions which were done from the user's browser. Technically it can show every transaction done by the service layer but as soon as a user has done a transaction it saves the address of the user to the browser's local storage and the transactions are filtered to the transactions of this user only.

The table shows basically a subset of the information available on Etherscan [12] or other BC explorers. It displays the transaction hash, the transaction status, the age of the transaction, from and to address, the amount of transferred tokens and the method which was called from the service layer.

Figure 4.7 shows a screenshot of the My transactions tab and its table. One can see, that it shows when a transaction was not successful or when it was confirmed or still pending.

ERC-865 pay gas service

Transfer tokens
Call a function
My transactions

Txn Hash	Status	Age	From	To	Quantity	Method
0x5a860f...351a	PENDING	3 secs	0x7b9A6bf86BB7317...aAeb	0x25fc28613d205f3...e53a	400 CHFT	transfer
0xdc4b3a...8edf	CONFIRMED	28 secs	0xD12dc33628FF06e...538c	0x2bfEcE4263bce0c...40c1	0 CHFT	transferAndCall
0x26d44e...6a89	REVERTED	6 hours	0xD12dc33628FF06e...538c	0x2bfEcE4263bce0c...40c1	0 CHFT	transferAndCall
0xcf40bf...06b1	REVERTED	6 hours	0xD12dc33628FF06e...538c	0x2bfEcE4263bce0c...40c1	0 CHFT	transferAndCall
0x55ee3f...f90f	REVERTED	6 hours	0xD12dc33628FF06e...538c	0x2bfEcE4263bce0c...40c1	0 CHFT	transferAndCall
0x409d8b...7f9f	CONFIRMED	6 hours	0xD12dc33628FF06e...538c	0x2bfEcE4263bce0c...40c1	0 CHFT	transferAndCall
0xb00098...2d42	CONFIRMED	6 hours	0xD12dc33628FF06e...538c	0x2bfEcE4263bce0c...40c1	20 CHFT	transferAndCall

Figure 4.7: The "My transaction" tab and the table with the sent transactions.

4.4.2 Signing the Transaction

The SEND button can be clicked as soon as the form is filled or a QR code with the necessary data is scanned. It is only clickable if all fields are filled and if both addresses are real Ethereum addresses.

Clicking the SEND button triggers the signing process. The signed transaction is then sent to the backend together with the transaction parameters. In addition to that, the sender address is saved to the local storage that "My transactions" table can filter for the relevant addresses.

Listing 4.9 presents the most important method implemented in the frontend, i.e., the `signTransactionData` function of the "Call a function" component. It encodes the parameters and signs the data before it is send to the backend. If other services only want to use the API of the service layer, they need to implement this signing process on their own.

First, in line 2 to 22, the parameters of the service contract function are encoded. As stated in the code, the first two parameters have to be ignored, because they are added by the token contract when doing the service contract call (see line 25 in listing 4.2. For

using the `encodeParameters` function of `web3.eth.abi`, the parameter types and values need to be passed split in two arrays. This generates a hex string which represents a bytes array.

Whereas line 2 to 22 are only used when signing a `transferAndCall` transaction the next lines are almost identically to the ones for a `transfer` function.

Line 24 to 49 generates the input for the keccak256 hash which is done in line 51. It is important, that the input is encoded the exact same way as it is done in the `transferAndCallPreSignedHashing` method in `Utils.sol` which is used by the token contract. The comment in line 26 shows what is called in the contract. The first parameter is the method signature of `transferAndCallPreSignedHashing` (0x38980f82), followed by the token contract address, the receiver address which is the service contract address in this case, the value to be transferred, the fee and the nonce. As one can see in line 26, for `transferAndCall` the method signature and the in line 2 to 22 encoded parameters of the to be called function are added to the input. Again, types and values need to be passed to `encodeParameters` in two separate arrays.

After the keccak hash is computed in line 51 and the "0x"-prefix is removed from the `privateKey`, the signature object is generated in line 57 to 63. Depending on which sign method is used in the token contract and therefore specified in the configuration file (see listing 4.6 in section 4.3), the `inputHash` is signed with the corresponding signing method.

Last but not least, the signature is transformed to a hex string. The recovery value is added; if it is 0 or 1 it is changed to 27 or 28 since only these are accepted by the token contract (line 65 to 68).

```

1  async signTransactionData() {
2      let callParameterTypes = this.state.callParameters.map(e => {
3          return e.type;
4      });
5      // we ignore the first two parameters of the service contract method
        since _from and _value are added by the token contract when the
        service contract is called.
6      callParameterTypes.slice(2);
7
8      let callParameterValues = this.state.callParameters.map(e => {
9          if (e.type === "bytes32" && e.value) {
10             if (!this.context.web3.utils.isHexStrict(e.value))
11                 return this.context.web3.utils.utf8ToHex(e.value);
12             }
13             return e.value;
14         });
15         // we ignore the first two parameters of the service contract method
            since _from and _value are added by the token contract when the
            service contract is called.
16         callParameterValues.slice(2);
17
18         const callParametersEncoded = this.context.web3.eth.abi.
            encodeParameters(
19             callParameterTypes,
20             callParameterValues
21         );

```

```

22   this.setState({ callParametersEncoded });
23
24   const nonce = Date.now();
25   // transferAndCallPreSignedHashing from Utils.sol
26   // keccak256(abi.encode(bytes4(0x38980f82), _token, _to, _value, _fee,
    _nonce, _methodName, _args));
27   const input = this.context.web3.eth.abi.encodeParameters(
28     [
29       "bytes4",
30       "address",
31       "address",
32       "uint256",
33       "uint256",
34       "uint256",
35       "bytes4",
36       "bytes"
37     ],
38     [
39       "0x38980f82",
40       this.state.selectedTokenContract.value.contractObj.options.address
41       ,
42       this.state.to,
43       this.state.value.toString(),
44       this.state.selectedTokenContract.value.feeTransferAndCall.toString
45       (),
46       nonce.toString(),
47       // call parameters
48       this.state.selectedMethod.value.signature,
49       callParametersEncoded
50     ]
51   );
52   const inputHash = this.context.web3.utils.keccak256(input);
53   const privateKey =
54     this.state.privateKey.substring(0, 2) === "0x"
55     ? this.state.privateKey.substring(2)
56     : this.state.privateKey;
57   const signObj =
58     this.state.selectedTokenContract.value.signMethod === "personalSign"
59     ? await web3.eth.personal.sign(inputHash.substring(2), this.state.
60       from)
61     : secp256k1.sign(
62       Buffer.from(inputHash.substring(2), "hex"),
63       Buffer.from(privateKey, "hex")
64     );
65   const signatureInHex =
66     "0x" + signObj.signature.toString("hex") + signObj.recovery <= 1
67     ? (signObj.recovery + 27).toString(16)
68     : signObj.recovery.toString(16);
69
70   this.setState({
71     signature: signatureInHex,
72     nonce
73   });

```

74 }

Listing 4.9: The `signTransactionData` method, which encodes the parameters and signs the transaction data entered in the frontend.

4.4.3 API Calls and Notifications

The frontend includes a simple notification system. When the transaction data was signed it is sent to the backend. Depending which functionality it is desired, either the endpoint `/api/transfer` or `/api/transferAndCall` is called. As soon as the call to the backend successfully returns the transaction hash, a polling is started which is listening to a transaction status change on the transaction object stored in the backend data storage. Every two second the new status is fetched from the backend.

Listing 4.10 shows the polling method and the different messages displayed depending on the transaction status.

In Figure 4.8, two notification can be seen. This screenshot was taken in the development environment, where the transaction is mined immediately on the Parity development chain. Therefore it is possible to show both notifications at once.

Currently the polling is only active, as long as the page is not refreshed. In the future, one could work with the local storage to keep the notification system alive in every case (see Chapter 6).

```

1 export const polling = async txHash => {
2   await timeout(2000);
3   let tx = await getTransaction(txHash)
4     .then(response => response.json())
5     .then(response => {
6       if (response.success) {
7         return response.data;
8       }
9     });
10  switch (tx.status) {
11    case TRANSACTION_STATUS.PENDING:
12      polling(txHash);
13      break;
14    case TRANSACTION_STATUS.CONFIRMED:
15      toast.success("Your transactions has been successfully mined.");
16      break;
17    case TRANSACTION_STATUS.REVERTED:
18      toast.error(
19        "Your transactions has been mined but the action was not
20          successful."
21      );
22      break;
23    case TRANSACTION_STATUS.ERROR:
24      toast.error("Something went wrong with your transaction.");
25      break;
26    case TRANSACTION_STATUS.OUT_OF_GAS:
27      toast.error(

```

```

27         "Your transaction could not be mined because it ran out of gas."
28     );
29     break;
30     default:
31         toast.info("Your transaction is not pending anymore.");
32     }
33 };

```

Listing 4.10: The polling method which is started as soon as the transaction is successfully sent to the backend.

The service successfully sent the transaction to the blockchain. TxHash: 0x3f9c...2a0d

Your transactions has been successfully mined.

Transfer tokens Call a function My transactions

SCAN QR CODE INSTEAD OF FILLING THE FORM

400 CHF Token

From: 0x7b9A6bf86BB7317DF7562106eCc45ad49acFaAeb

To: 0x25fc28613d205f3c9ae0937827Ff6Ab07754e53a

Transaction 50 CHFT
 fee: ≈ 0.005 ETH
 ≈ 0.73 USD

Token total: 450 CHFT
 ≈ 0.045 ETH
 ≈ 6.53 USD

95FE3783808009AFDA9A614D46511E304FD435C7E0ECE24A52E20D0A1c

Your private key is only used to sign the entered transaction data. It is neither stored nor sent somewhere.

SEND

Figure 4.8: The notifications displayed when a transaction object is sent to the service layer.

4.5 Service Layer

The service layer consists of three parts. The REST API which is the interface between the interface and the service layer. The REST API distributes the request to the corresponding services and is therefore connected to the data storage as well as to the Ethereum interface.

The service layer is implemented as a Node.js server. The application is deployed to Heroku and the API can be accessed under <https://pay-gas-service.herokuapp.com/api>.

4.5.1 REST API

For implementing the REST API the web framework ExpressJS is used. It is a fast, open and simple web framework for Node.js [13].

Table 4.1 shows the implemented REST API. The API consists of four endpoints: Two POST methods which are the center of the service layer API, and the GET methods which are auxiliary endpoints to improve the usability.

The `/transfer` and `/transferAndCall` endpoint require a transaction object as request body. The transfer and the transferAndCall transaction object slightly differ. Listing 4.11 explains the differences. The request bodies of both endpoints need to include the keys signature, from, to, value, tokenContract and nonce. The `/transferAndCall` endpoint additionally requires the keys methodName and callParametersEncoded. Both endpoint return the transaction hash (txHash) of the sent transaction. The returned txHash can then be passed for example as path parameter to the `/transactions/:txHash` endpoint.

The `/transactions` and `/transactions/:txHash` obviously don't require a request body, since they are GET methods. The `/transactions` endpoint is able to process a fromAddresses list as query parameter to filter the transaction according to the specified sender addresses. The list of the addresses needs to be comma delimited. The following link shows how the API needs to be called to filter the transactions according some addresses:

```
https://pay-gas-service.herokuapp.com/api/transactions?fromAddresses=0x7b9A6bf86BB7317DF7562106eCc45ad49acFaAeb,0xD12dc33628FF06eA9c22BC28E9Ba0Ed201f4538c
```

The `/transactions/:txHash` endpoint returns one single transaction object whereas the `/transactions` endpoint is returning a list of transaction objects. Listing 4.12 shows how an object looks like in the case of a confirmed transaction. It contains basically everything which can be saved during the process (besides the private key). The transactionInput contains all the input passed to the contract. Either the receipt or the error is filled with an object received from the Ethereum client. The status and created and last modified timestamps are returned within the response too.

Method	Endpoint	Path/Query Parameters	Request Body	Response Data
POST	/transfer	-	transfer transaction object	txHash (string): the transaction hash
POST	/transferAndCall	-	transferAndCall transaction object	txHash (string)
GET	/transactions	fromAddresses (comma delimited list of addresses)	-	list of transaction objects
GET	/transactions/:txHash	txHash: the transaction hash	-	transaction object

Table 4.1: The REST API endpoints of the service layer

```

1
2 req.body = {
3   signature: "",
4   from: "",
5   to: "",
6   value: "",
7   tokenContract: "",
8   nonce: "",
9   // the following to keys are only needed in the transferAndCall object
10  methodName: "",
11  callParametersEncoded: ""
12 };
13
14 const router = express.Router();
15
16 router.post(
17   '/transfer',
18   requestHandler(async req => {
19     return sendTransferPreSignedTransaction(req.body);
20   })
21 );

```

Listing 4.11: The registration of the /transfer endpoint

```

1 {
2   "_id": "0
3     x7e6130db1dff7abfc08ca1955cce18c7c1b5cda2c61f93c5380a1ae831d57b9c
4     ",
5   "transactionInput": {...}
6   "from": "0x7b9A6bf86BB7317DF7562106eCc45ad49acFaAeb",
7   "status": "CONFIRMED",
8   "created": "2019-10-14T06:36:07.840Z",
9   "modified": "2019-10-14T06:36:07.849Z"
10  "serviceAccountAddress": "0x00a329c0648769A73afAc7F9381E08FB43dBEA72
11  ",

```

```

9   "type": "transferAndCall",
10  "error": {...} | null,
11  "receipt": {...} | null,
12 }

```

Listing 4.12: The transaction object which is returned by the `/transactions/:txHash` endpoint.

4.5.2 Data Storage

For the data storage the free cloud data storage mongoDB Atlas is used. mongoDB is a document database which allows to save any JavaScript object to its collections. Collections are same as tables in a relational database, they store multiple JavaScript objects. Listing 4.13 shows the reason why a cloud database is used and no local one is set up. Only this code snippet needs to be added to have a fully functional database ready to use.

```

1  import mongodb from "mongodb";
2  const MongoClient = mongodb.MongoClient;
3
4  export let db;
5
6  const startApplication = async () => {
7
8      await MongoClient.connect(process.env.MONGODB_URI, {
9          useNewUrlParser: true,
10         useUnifiedTopology: true
11     })
12     .then(client => {
13         db = client.db(process.env.DB_NAME);
14         console.log("Connected to database.");
15     })
16     .catch(err => {
17         console.log(err);
18         process.exit(1);
19     });

```

Listing 4.13: This code snippet initialises the database and starts the application.

The data storage currently has two collections. One is the configurations collection where the shared configuration, described in section 4.3, for the service layer is saved. This was set up manually and currently no endpoint exists to generate a new configuration file.

The second collection is the transactions collection. Every made transaction, no matter if failed or confirmed, is saved to this collection. Listing 4.12 not only shows the JavaScript object returned to the client when calling `/transactions/:txHash`, it is also the exact same object which is saved in the transactions collection.

Listing 4.14 shows how the transactions collection is queried after the `/transactions` endpoint calls the `getTransactions` functions of the database service: A simple `".find in"` and `".sort"` brings the desired functionality.

```

1  router.get(
2    "/transactions",
3    requestHandler(async req => {
4      if (req.query.fromAddresses)
5        return getTransactions(req.query.fromAddresses.split(","));
6      else return getTransactions();
7    })
8  );
9
10 export const getTransactions = fromAddresses => {
11   if (fromAddresses) {
12     return db
13       .collection("transactions")
14       .find({ from: { $in: fromAddresses } })
15       .sort({ created: 1 })
16       .toArray();
17   } else
18     return db
19       .collection("transactions")
20       .find()
21       .sort({ created: 1 })
22       .toArray();
23 };

```

Listing 4.14: The /transactions endpoint and how the data storage is queried.

4.5.3 Ethereum Interface

The Ethereum interface is responsible for the communication between the Node.js application and the Ethereum BC. This is done with Web3.js. Web3.js is a JavaScript framework which includes functions which translate into JSON-RPC calls as well as numerous util functions. Web3.js also takes care of establishing the connection to the Ethereum full node.

Perform send transaction call

The Ethereum interface is used to do the actual delegated call to the BC. The frontend calls the API endpoint and the request is forwarded for example to the `sendTransferAndCallPreSignedTransaction` service seen in listing 4.15. The Web3.js interface frameworks allows to make JSON-RPC calls by just calling `tokenContractObject.methods."methodName".send()`.

The `send` function returns a so called `PromiEvent` [1]. A `PromiEvent` is an event based promise. One can handle it like a promise but it also emits events to listen to and to react on. In line 19 of the listing, the `transferAndCallPreSigned` call is triggered with the keyword `send` and on line 23, 32 and 41 three event listeners are registered. On `"transactionHash"` the transaction information that is already know is saved to the database and the promise is resolved with a response. On `"receipt"` or on `"error"` (only one of the events occurs), the service reacts on another event and updates the transaction with the final transaction status and the receipt or the error object.

```

1 import { tokenContracts } from "../../helpers/get-contracts.mjs";
2
3 export const sendTransferAndCallPreSignedTransaction = transactionObject
4   => {
5     const tokenContract = tokenContracts.find(
6       contract => contract.address === transactionObject.tokenContract
7     );
8     return tokenContract.contractObj.methods
9       .transferAndCallPreSigned(
10         transactionObject.signature,
11         transactionObject.from,
12         transactionObject.to,
13         transactionObject.value.toString(),
14         tokenContract.feeTransferAndCall.toString(),
15         transactionObject.nonce.toString(),
16         transactionObject.methodName,
17         transactionObject.callParametersEncoded
18       )
19       .send({
20         from: config.unlockedServiceAccount,
21         gas: 8000000
22       })
23       .on("transactionHash", async tx => {
24         const saveToDbSuccess = await saveTransaction(
25           tx,
26           transactionObject,
27           config.unlockedServiceAccount,
28           "transferAndCall"
29         );
30         return { txHash: tx, saveToDbSuccess };
31       })
32       .on("receipt", async receipt => {
33         updateTransaction(
34           receipt.transactionHash,
35           receipt.status === true
36             ? TRANSACTION_STATUS.CONFIRMED
37             : TRANSACTION_STATUS.REVERTED,
38           receipt
39         );
40       })
41       .on("error", errorReceipt => {
42         console.log(errorReceipt);
43         const errorObj = JSON.parse(
44           errorReceipt.message.substring(
45             "Transaction has been reverted by the EVM:\\n".length - 1
46           )
47         );
48         updateTransaction(
49           errorObj.transactionHash,
50           TRANSACTION_STATUS.ERROR,
51           null,
52           errorObj
53         );
54       });

```

55 };

Listing 4.15: The `sendTransferAndCallPreSignedTransaction` which does the call to the Ethereum BC.

For being able to call token contract methods as it is done in listing 4.15, the `tokenContract` list needs to be first initialised with contract objects. This is done in `get-contracts.mjs` by iterating over the `acceptedTokens` list of the configuration. For every accepted token an object is generated with the information from the configuration object as well as the `contractObj` where a new `web3.eth.Contract` object is created (see line 11 and 17 in listing 4.16). The creation requires the contract address as well as the ABI of the token contract. The `tokenContract` list is implemented as a singleton such that the contract objects are only created once.

```

1 import web3 from "../web3Instance.mjs"
2 export let tokenContracts = [];
3 export let serviceContracts = [];
4
5 const setContracts = () => {
6   if (tokenContracts.length === 0) {
7     if (config.ethereumNetwork !== 'ganache' && config.ethereumNetwork
8       !== 'parity') {
9       tokenContracts = config.acceptedTokens.map(c => {
10         ...c,
11         contractObj: new web3.eth.Contract(c.abi, c.address)
12       });
13     };
14     serviceContracts = config.serviceContracts.map(c => {
15       return {
16         ...c,
17         contractObj: new web3.eth.Contract(c.abi, c.address)
18       };
19     });
20
21     [...]
```

Listing 4.16: `setContracts` is a singleton which populates the arrays with contract objects.

The `web3` instance imported in the first line of listing 4.16 is very central for the connection to the Ethereum BC. The `web3` singleton is initialised once and is used for doing every transaction on the BC. A quick look at how it is initialised shows (listing 4.17), that the same method can also be used for the frontend. Since this service layer makes it possible to use the Ethereum BC without connecting to a `web3` provider, the `getFrontendWeb3` method can be ignored.

Depending on the configuration file, in which the `ethereumFullNodeAddress` can be specified, the corresponding protocol is used. Then, the created web-socket provider is passed to the "new Web3" call, where a new `web3` instance is created.

```

1 let web3;
2 const getWeb3Instance = () => {
3   if (!web3) {
```

```

4   web3 =
5       typeof window !== "undefined"
6       ? getFrontendWeb3()
7       : new Web3(getBackendProvider());
8   }
9 };
10
11 const getFrontendWeb3 = () => {...};
12
13 const getBackendProvider = () => {
14     let protocol = "ws://";
15     let fullNodeAddress = config.ethereumFullNodeAddress;
16
17     if (!fullNodeAddress) {
18         if (config.ethereumNetwork === "main") {
19             fullNodeAddress = "infura.io/ws";
20             protocol = "wss://";
21         } else if (config.ethereumNetwork === "ganache") {
22             fullNodeAddress = "127.0.0.1:7545";
23         } else if (config.ethereumNetwork === "parity") {
24             fullNodeAddress = "127.0.0.1:8546";
25         }
26     }
27     return new Web3.providers.WebsocketProvider(protocol + fullNodeAddress
28 );
29 };

```

Listing 4.17: The singleton for the web3 instance initialisation.

4.6 Ethereum Blockchain

This section describes what has been done according to the lowest level of the service, the Ethereum blockchain. Some example smart contracts have been implemented and deployed. An Ethereum full node has been set up and a script which does automatic deployment has been improved and upgraded such that the newest Solidity compiler can be used to compile the smart contracts and the newest Web3.js version to do the continuous deployment. This script is recommended to use for further development and testing of this application or of any other smart contract based applications.

4.6.1 Example Smart Contract

The smart contracts written in Solidity were moved to the beginning of this chapter to describe the use case and to make sure that the further descriptions are understandable. See section 4.2.

The CHF Token contract address is deployed to Rinkeby and the token address is 0x64B957-C8912Edd9315a6Da36aEd04585777cB08E.

The Translation Service contract address is deployed to Rinkeby too and the contract address is 0x2bfEcE4263bce0cbF2364D3c26b60ee153a440c1

4.6.2 Full Node Specification

The smart contracts are deployed on the Ethereum Rinkeby Testnet. Therefore, the service layer needs an Ethereum Rinkeby full node to connect to. With the following command a working Geth client can be started.

```
$ geth --syncmode=fast --rinkeby \
--ws --wsorigins="*" \
--unlock <ADDRESS>
--allow-insecure-unlock
```

The Geth client is started in fast sync mode. The flag `-allow-insecure-unlock` needs to be added because this is deprecated. This is also something which can be improved. The node is open for web socket connections with the flag `-ws` and all web socket origins are allowed (not recommend for main network).

The system administrator will be prompted to enter the password to unlock the account.

For development and local testing or try out the Parity development chain can be used. The chain can be started with

```
$ sudo parity --config dev --geth \
--unlock 0x00a329c0648769a73afac7f9381e08fb43dbea72 \
--password ./parity-empty-pw.txt \
--jsonrpc-hosts any --jsonrpc-apis eth,net,private,parity,personal \
--ws-apis eth,net,private,parity,personal \
--jsonrpc-cors https://remix.ethereum.org
```

It takes the default Parity dev chain configurations, unlocks the specified default address with an empty password saved in `parity-empty-pw.txt`. It allows any JSON-RPC host and enables all APIs for JSON-RPC and web socket APIs. It also allows CORS for `remix.ethereum.org`.

4.6.3 Deployment Script for Development and Testing

During the development of the service layer and previous projects, a deployment script was developed. It was existing already before the project, but was not working with Solidity 0.5. This deployment script was refactored and improved that it works with the newest Solidity compiler and the newest Web3.js version.

The Solidity SCs which should be compiled can be specified in `src/smartcontracts/deployment/get`. Also libraries used in these contracts are specifiable in the `findImports` method in this file.

```
$ yarn deploy-contracts
```

or

```
$ BC_NETWORK=parity NODE_ENV=development node --experimental-modules \
src/smartcontracts/deployment/deployment-script.mjs
```

runs the deployment script. The deployment script first compiles the contracts with `solc.compile` and creates for all compiled contracts a web3 contract object. This object is then used to deploy the contracts to the Parity development chain automatically. After the deployment the contract ABIs and addresses are saved to the file system from where it is read from the server during the startup.

Listing 4.18 presents the beginning of the deploy call. The contract ABI and the bytecode object are used for that.

```
1      // create web3 contract object
2      const web3Contract = new web3.eth.Contract(compiledContract.abi);
3      // deploy the contract
4      return web3Contract.deploy({ data: "0x" + compiledContract.evm.
        bytecode.object })
5      .send({
6        from: accounts[0],
7        gas: 8000000
8      })
9      .on("receipt", receipt => {
```

Listing 4.18: The deploy call to the Ethereum BC.

Chapter 5

Evaluation and Discussion

In this chapter the implemented service layer is evaluated and discussed. Moreover, improvements which can be made are proposed.

5.1 Evaluation

The System Usability Scale [22] is a testing strategy oriented towards the user for measuring the usability of a system. During tests, participants are asked to use the application and to answer question on a scale between Strongly Agree to Strongly Disagree. The scoring system and the questionnaire are outlined in the System Usability Scale (SUS) Template [23].

Since the main goal of the service layer is to improve the usability of token based applications by reducing the need for Ether to pay the gas the System Usability Scale fits as a basic evaluation.

Due to the fact that token based applications are currently very limited, it is difficult to put such a service layer in a real use case to test it properly. Although the imagination and the technical know how of a participant must be relatively high, it was decided to also take users not familiar with blockchain technologies into consideration.

Although the amount of test participants is very limited, it is still worth mentioning it: The software was used by four people and they were quickly interviewed afterwards. Important to say is that two of them are not familiar with blockchain technologies and cryptocurrencies, one test person already bought Bitcoin once and partially knows how a blockchain works, and one is a smart contract developer.

The result of the SUS are listed in table 5.1. The list below describes which questions were asked.

Question	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
1		4	3	1 2	
2	1 2 3 4				
3				2 4	1 3
4	1		2	4	3
5				4	1 2 3
6	1 3 4	2			
7					1 2 3 4
8	2 3	1 4			
9		3 4		2	1
10	1	2	4	3	

Table 5.1: The results of the System Usability Scale, interviewing four participants: 1 = smart contract developer, 2 = basic knowledge in Bitcoin and blockchain, 3 & 4 = no knowledge at all.

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that i would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The outcome of the test was positive towards the user experience and usability of the service layer. The questions 2, 5, 6 and 7 were answered with the most match between the participants. In these questions unnecessary complexity and inconsistency was denied and good integration and easiness was confirmed. In addition to that some comments on the first question were for example from participant 4: "I won't use that frequently but this is only because I don't use token based applications". In another comment participant 3 mentioned the following: "I would need support for getting such a QR code and to get the tokens in the first place. But as soon as I have them, I probably won't need support for this [the service layer]".

5.2 Discussion

The service layer is live on the Ethereum Testnet and can be used. The functionalities are simple and understandable. This is also the opinion of the four test users while only a subset of it was familiar with blockchain technologies.

The service layer serves the basic functionality but also one additional core feature (the `transferAndCall` functionality) and some auxiliary features such as the transactions endpoint, the notification system and the configuration file. In the authors opinion, the auxiliary features have a big impact on the usability. Let's imagine a service layer with just the core functionality. It would still feed the purpose, since the main goal of the service is to enable the user to only have tokens in its wallet instead of being forced to pay the gas in Ether additionally. Only the core feature would decrease the level of complexity already, but users would ask themselves, how the outcome of their transaction was, firstly, if it was mined, and secondly, if the Ethereum state was successfully updated.

Also the additional features in the frontend bring a lot of value to the service. Although it was assumed that other parties who will use the service might implement their own frontend or even use the service from within an already existing platform, it was focused on creating a simple user experience. And in the authors opinion this goal was reached. The notification feature does not leave the user in the dark and last but not least the QR code scanning possibility brings the user very fast towards its goal of spending the money without having Ether.

Nevertheless, there is still room for improvement. While implementing the service layer, several tasks and ideas were discovered which can improve the service even more. The following list provides a subset of these ideas:

- More generic way of calling the token contract: At the moment every token contract needs to be specified in the backend. A token contract not specified in the configuration file cannot be called. The advantages of an interface are that you don't have to know the underlying object in detail. This advantage is not used in the current service layer. Since every token contract which the service wants to serve, is at least implementing the `transferPreSigned` method, the call to that function could be more generic.

The reason for specifying every token contract in the configuration, is on one hand the usage of the `web3.eth.Contract.methods` approach from the Web3.js framework to call the functions (it requires the token ABI), and on the other hand the conversion of the fee paid in tokens towards Ether and US dollar. The conversion price currently needs to be specified somewhere.

The token contract call could be more generic by using plain JSON-RPC calls. One would need to specify method signature (for example the keccak256 hash of `transferPreSigned` and its parameters) and send the transaction data with `eth_sendTransaction` to the RPC. This would improve the usability because the service would accept automatically all contracts with the method implemented.

- Especially when the previous point is implemented, but also if not, there is need for a look up system. The service should be able to look up the entered addresses and check if it is a smart contract address, and to be able to distinguish if it is a token contract with corresponding transfer functions or if it is a callable service contract. Once figured out that the address belongs to a valid token contract, the service could look up the current prices for this token and the current ETH/USD price. The fee estimation could be done dynamic depending on the price of the token.

The same applies for a callable service contract. The service' user experience would increase a lot if one could enter an address, and if it is a known smart contract address, the contract ABI, where the callable methods are extracted from, would be fetched from a service like Etherscan. This would remove the requirement to configure the accepted service contracts in the configuration object and will open up the service layer for basically every smart contract.

- While a fixed token fee for the `transferPreSigned` call is totally fine, since the transfer of tokens only will result always in more or less the same amount of gas, for the `transferAndCallPreSigned` call a static token fee is not suitable. Preferably the gas is estimated by the backend and past via an additional endpoint to the frontend. The frontend signs the transaction with this value. The difficulty in such a functionality is that the frontend and the backend need to have the exact same fee (otherwise the hash will be different). To make sure that the client does not alter the fee value, the backend needs to either cache the fee or estimate it again when the call is actually made. For the second approach, one could introduce a threshold and if the frontend value is differing more than this threshold the transaction is aborted. If the client's fee value is close enough to the one calculated by the backend, the client's fee is accepted.
- An additional improvement is possible in the notification system. As already described, the user currently only receives the notifications, when it is stayed in the same component. As soon as it is changed to another component (e.g. another tab) the notification is lost. This definitely can be improved and will improve the user experience too. One could think about working with the local storage to keep track of the pending transactions. Another possibility is working with web sockets and emitting the message as soon as the transaction is mined.

As a conclusion, one can say that the service layer has a big impact on the user experience of token based applications. The usability without such a service layer is very poor. Depending on how well the service is integrated in an application, one can imagine that it could be a very good experience to use a decentralised application and paying with tokens by only scanning a QR code.

Chapter 6

Summary and Future Work

In this project a service layer for token based applications was implemented. The service layer supports normal ERC-865 tokens as specified in the official EIP (see Section 2.2.3) as well as ERC865Plus677ish tokens. ERC865Plus677ish token combines the advantages of the ERC-865 (the possibility to delegate the sending of the transactions and therefore paying the gas to a third party) with the ones from ERC-677 (the ability to transfer tokens to a SC and trigger a function of the SC after the transfer was successful). The service sends so called "pre-signed" transaction to the BC and pays for the gas. A fee is transferred to the service address.

The service layer consists of a frontend communicating with a REST API. The REST API endpoints are connected to a cloud data storage and to the Ethereum BC interface. The user signs the transaction data with its private key in the frontend and sends the signature together to the backend. The backend then does the actual call. The status of the BC transaction is saved to the cloud data storage and is propagated to the frontend with a simple notification system.

The goal of this work was to make ERC-865 token based applications user friendly. Without such a service, the users of token based applications always need to have Ether to pay the gas and tokens to use the application.

The result of the project is a simple and understandable service which does not leave the user in the dark when it is about the outcome of their transactions. This could also be approved by a limited number of test users with a various understanding of blockchain or technology in general. Nevertheless, there are some ideas for the future to improve the service in terms of usability as well as functionality.

What will affect the user the most, is a dynamic fee pricing and dynamic fee estimation. The current state works with fixed fee in tokens and without gas estimation SC calls. One could think about changing that to a dynamic way building the fee by estimating the gas first and also taking the token respectively the Ether/US dollar price into consideration. A second task could be the handling of unknown service contracts: Currently the token and the service contracts need to be specified in the configuration file. Whereas the token contract implements the ERC-865 or the ERC865Plus677ish interface and therefore actually no further specification of the contract ABI is needed, the service contract

methods are completely unknown. A service which is trying to fetch the callable methods from other resources would be beneficial. For more (detailed) information about future improvement possibilty, see Section 5.2.

Bibliography

- [1] Callbacks Promises Events. Available at: <https://github.com/OpenZeppelin/openzeppelin-contracts/pull/1622>, Accessed on 23.11.2019.
- [2] CoinMarketCap. All Tokens. Available at: <https://coinmarketcap.com/tokens/views/all/>, Accessed on 23.04.2019.
- [3] DEZOS GmbH. Dezos Smart Contract. Available at: <https://github.com/DEZOS-IC0/SmartContract>, Accessed on 30.11.2018.
- [4] Digital Signature Standard. Available at: <https://csrc.nist.gov/publications/detail/fips/186/4/final>, Accessed on 22.11.2019.
- [5] Ethereum Community. ERC20 Token Standard. Available at: https://theethereum.wiki/w/index.php/ERC20_Token_Standard, Accessed on 23.04.2019.19.10.2019.
- [6] Ethereum Community. ERC223. Available at: <https://github.com/ethereum/eips/issues/223>, Accessed on 23.04.2019.19.10.2019.
- [7] Ethereum Community. ERC677: transferAndCall Token Standard Available at: <https://github.com/ethereum/EIPs/issues/677>, Accessed on 23.04.2019.
- [8] Ethereum Community. ERC865: Pay transfers in tokens instead of gas, in one transaction. Available at: <https://github.com/ethereum/EIPs/issues/865>, Accessed on 23.04.2018.
- [9] Ethereum Foundation: Ethereum Blockchain App Platform. Available at: <https://www.ethereum.org/>, Accessed on 30.11.2018.
- [10] Ethereum Improvement Proposal. EIP 20: ERC-20 Token Standard. Available at: <https://eips.ethereum.org/EIPS/eip-20>, Accessed on 19.10.2019.
- [11] Ethereum's Yellowpaper. Available at: <https://ethereum.github.io/yellowpaper/paper.pdf>, Accessed on 24.11.2019.
- [12] Etherscan.io. Ethereum Blockchain Explorer. ERC-20 Tokens. Available at: <https://etherscan.io/tokens>, Accessed on 19.10.2019.
- [13] Express.js. Available at: <https://expressjs.com>, Accessed on 24.11.2019.

- [14] go-ethereum's SignatureValues function. Available at: https://github.com/ethereum/go-ethereum/blob/master/core/types/transaction_signing.go#L195, Accessed on 23.11.2019.
- [15] JSON-RPC. A light weight remote procedure call protocol. Available at: <https://www.jsonrpc.org/>
- [16] LinkToken/ERC677Token.sol at master - smartcontractkit/LinkToken. Available at: <https://github.com/smartcontractkit/LinkToken/blob/master/contracts/ERC677Token.sol>, Accessed on 20.10.2019.
- [17] OpenZeppelin. Available at: <https://openzeppelin.com>, Accessed on 19.10.2019.
- [18] OpenZeppelin. ERC 865. Available at: <https://github.com/OpenZeppelin/openzeppelin-solidity/pull/741/commits/67ca9e7a4249184e642ac097b41be08a958fd302>, Accessed on 23.04.2019.
- [19] OpenZeppelin: Signature Malleability. Available at: <https://github.com/OpenZeppelin/openzeppelin-contracts/pull/1622>, Accessed on 23.11.2019.
- [20] ReactJS: A JavaScript library for building user interfaces. Available at: <https://reactjs.org/>, Accessed on 10.11.2019.
- [21] Token ERC Comparison for Fungible Tokens. Available at: <http://blockchainers.org/index.php/2018/02/08/token-erc-comparison-for-fungible-tokens/>, Accessed on 22.11.2019.
- [22] System Usability Scale (SUS). Available at: <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>, Accessed on February 2019.
- [23] System Usability Scale Template. Available at: <http://www.uxforthemasses.com/wp-content/uploads/2011/04/SUS-System-Usability-Scale.doc>, Accessed on February 2019.

List of Figures

2.1	The sequence diagram of transferring ERC20 tokens to a smart contract and do something [6].	6
3.1	The designed service architecture	16
3.2	The process flow model of the application and its components.	17
4.1	The implemented service architecture	20
4.2	The "Transfer tokens" tab empty, just after startup of the application . . .	32
4.3	The popup which is prompted when the user is logged into MetaMask. . .	32
4.4	The QR code section opens by clicking on the "Scan QR code..." button. .	33
4.5	The "Call a function" tab with the function drop-down of the Translation Service contract.	35
4.6	The input fields for method parameters of the calling functions.	36
4.7	The "My transaction" tab and the table with the sent transactions.	37
4.8	The notifications displayed when a transaction object is sent to the service layer.	41

List of Tables

2.1	Comparison of the ERC token interfaces 20, 223, 677 and 865.	13
4.1	The REST API endpoints of the service layer	43
5.1	The results of the System Usability Scale, interviewing four participants: 1 = smart contract developer, 2 = basic knowledge in Bitcoin and blockchain, 3 & 4 = no knowledge at all.	52

Appendix A

Installation Guidelines

The following sections describe how the service is installed locally:

A.1 Pre-Requisites

- **Node.js:** Install the newest version of node.js on your local computer. <https://nodejs.org/en/download/>
- **Yarn:** Install yarn on your local computer. <https://yarnpkg.com/lang/en/docs/install/>
- **Parity Ethereum Client:** Install Parity on your local computer. <https://www.parity.io/ethereum/#download>
- **Chrome Browser** Install Chrome Browser on your local computer. <https://www.google.com/chrome/>
- **Meta Mask (Chrome extension):** *optional*, Install Meta Mask as Chrome extension. <https://metamask.io/>

A.2 Service Layer Frontend

The following steps are needed for starting the frontend:

- Make sure the Node.js and Yarn is installed.
- Install the dependencies

```
$ yarn
```
- Start the frontend

```
$ yarn start
```

A.3 Ethereum Full Node

- Make sure Parity is installed.
- Check your Parity dev account address and adjust it in the command below if it is not the same.
- Start the Parity dev-chain

```
$ yarn devchain
```

or

```
$ sudo parity \  
--config dev \  
--geth \  
--unlock 0x00a329c0648769a73afac7f9381e08fb43dbea72 \  
--password ./parity-empty-pw.txt \  
--jsonrpc-hosts any \  
--jsonrpc-apis eth,net,private,parity,personal \  
--ws-apis eth,net,private,parity,personal \  
--jsonrpc-cors https://remix.ethereum.org
```

A.4 Smart Contracts

- Make sure Parity is installed and started in dev-chain mode.
- Compile and deploy the development contracts

```
$ yarn deploy-contracts
```

A.5 Service Layer Backend

The following steps are needed for starting the backend:

- Make sure the Node.js and Yarn is installed.
- Create a mongoDB Atlas account, cluster and database (<https://www.mongodb.com/cloud/atlas>) or start a local mongoDB instance.
- Create a `.env` file in the root directory (erc865-pay-gas-service) with the following content:

```
1 MONGODB_URI=<MONGODB CONNECTION STRING>  
2 DB_NAME=<DATABASE NAME>
```

- Install the dependencies

```
$ yarn
```

- Make sure in `config.json` the `ethereumNetwork` is specified as `parity` and the `ethereumFullNodeAddress` is either set to `127.0.0.1:8546` or left empty.
- Start the server

```
$ yarn server
```


Appendix B

Contents of the CD

The CD which is added to this report contains the following:

- The Abstract in `abstract.txt`
- The german translation of the abstract in `zusammenfassung.txt`
- This report in PDF format: `report.pdf`
- The source code (also open-sourced on GitHub <https://github.com/severinwullschleger/erc865-pay-gas-service>) in ZIP format: `sourcecode.zip`