Cashless to E-Cash

Bachelor's Thesis

Course of study	Bachelor of Science in Computer Science
Author	Joel Roman Häberli
Advisor	Prof. Dr. Benjamin Fehrensen
Co-advisor	Prof. Dr. Christian Grothoff
Expert	Dr. Alain Hiltgen, UBS

Version 1.0 of May 20, 2024

- Technic and Computer Science
- Institute for Cybersecurity and Engineering ICE

Abstract

In order to withdraw digital cash in GNU Taler, the *Taler Exchange* needs guarantees to legally secure the transaction. Withdrawing digital cash using Taler physically establishes direct trust, since cash can be used in order to withdraw digital cash and the transaction is completed. If you want to withdraw digital cash using cashless systems like credit cards, the *Taler Exchange* has no proof that the payment has succeeded. In order to fill this cap, this thesis proposes a framework allowing cashless withdrawals using GNU Taler. A reference implementation is created which establishes a trust relationship between the terminal manufacturer Wallee and the *Taler Exchange* through a newly created component called *C2EC*. This enables a trust relationship between the *Taler Exchange* and the terminal operator which allows withdrawing Taler without using cash. The liability for the transaction is on the side of the terminal operator and therefore establishes the guarantees for the *Taler Exchange*.

Contents

Abstract iii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Perspectives . . . 2
1.2.1 Taler Exchange (C2EC) . . . 2
1.2.2 Terminal Application . . . 2
1.2.3 Taler Wallet . . . 2
1.3 Goal . . . 2
1.3.1 C2EC . . . 3
1.3.2 Wallee POS Terminal . . . 3

2 Overview 5
2.1 Components . . . 5
2.2 Process . . . 5
2.2.1 The Terminal . . . 7
2.2.2 The C2EC . . . 8
2.2.3 The Wallet . . . 9

3 Architecture 11
3.1 C2EC . . . 11
3.1.1 C2EC Perspective . . . 11
3.1.2 Withdrawal-Operation state transitions . . . 11
3.1.3 Authentication . . . 12
3.1.4 The C2EC RESTful API . . . 13
3.1.5 Taler Wirewatch Gateway API . . . 15
3.1.6 C2EC Entities . . . 17
3.2 Payto wallee-transaction extension . . . 18
3.2.1 Payto refund using Wallee . . . 19
3.2.2 Extensibility . . . 19
3.3 Taler Wallet . . . 19
3.3.1 Taler Wallet Perspective . . . 19
3.4 Wallee . . . 20
3.4.1 Wallee Perspective . . . 20
3.4.2 Wallee Terminal . . . 20
3.4.3 Wallee Backend and API . . . 21

4 Implementation 23
4.1 C2EC . . . 23
4.1.1 Endpoints . . . 23
4.1.2 Abortion Handling . . . 25
4.1.3 Processes . . . 32
4.1.4 Providers . . . 33
4.2 Wallee POS Terminal . . . 36
4.2.1 Withdrawal flow . . . 36
4.2.2 Screens . . . 36
4.2.3 Abortion Handling . . . 38
4.3 Database . . . 41
4.3.1 Schema . . . 42
4.3.2 Triggers . . . 44
4.4 Wallet . . . 46
4.5 Security . . . 46
4.5.1 Withdrawal Operation Identifier (WOPID) . . . 46
4.5.2 Database Security . . . 46
4.5.3 Authenticating at the Wallee ReST API . . . 47
4.5.4 API access . . . 48
4.5.5 Registering Providers and Terminals . . . 48
4.5.6 Deactivating Terminals . . . 49
4.6 C2EC CLI . . . 49
4.6.1 Adding a Wallee provider . . . 50
4.6.2 Adding a terminal . . . 50
4.6.3 Deactivating the terminal . . . 50
4.6.4 Setting up the Simulation . . . 50
4.7 Testing . . . 50
4.8 Deployment . . . 51
4.8.1 Preparation . . . 51
4.8.2 Setup . . . 52
4.8.3 Deploy . . . 52
4.8.4 Migration and releases . . . 52

5 Results 53
5.1 Discussion . . . 53
5.2 Results . . . 53
5.3 Future Work . . . 53

Bibliography 57

List of Figures 59

List of Tables 61

Listings 63

Glossary 65

1 Introduction

1.1 Motivation

Which payment systems do you use in your daily live and why? Probably one you know it is universally accepted, reliable, secure and the payment goes through more or less instantly.

The **universal acceptance** was identified as one of the most important aspects in a report which was published on behalf of the ECB (European Central Bank) in march 2022 as result of a focus group concerning the acceptance of a digital euro [1] as new payment system. The universal acceptance was even identified as *the* most important property amongst the general public and tech-savvy people in the report [2].

In a world, where everything is connected and everything is accessible from everywhere (one might think), it is therefore very important to make it as easy as possible to on-board people on a product. This is also the case for Taler. For a wide acceptance of the payment system Taler, it is important that various ways exist to withdraw digital cash in Taler.

This is where this thesis hooks in. Currenlty it is possible to withdraw digital cash using Taler at a Bank which runs a *Taler Exchange* and integrates the respective API. At time of this writing only one Bank is in the process of running a *Taler Exchange*. At the Berner Fachhochschule an *Exchange* is operated and digital cash can be withdrawn at the secretariat using cash.

To make the access to digital cash using Taler easier and allow faster spreading of the payment system Taler, a framework for cashless withdrawal of digital cash is proposed and implemented in order to open new doors for the integration and adoption of the Taler payment system within the society.

To make the withdrawals using a credit card possible, various loose ends must be put together within the Taler ecosystem and the terminal provider.

Therefore a new component C2EC shall help, establishing a trustworthy relationship, which makes it possible for the *Exchange* to issue digital cash to a customer. Therefore the *Exchange* is not putting his trust on cash received but rather on the promise of a trusted third party (a terminal provider) to put the received digital cash in a location, controlled by the *Exchange* eventually (e.g. a bank account owned by the *Exchange*).

This enables a broader group of people to leverage Taler for their payments. Which eventually leads to a wider adoption of the payment system Taler.

1.2 Perspectives

During the initial analysis of the task, three areas of work were discovered. One is the *Taler Exchange*, one the Application for the terminal and the (Taler) *Wallet*. This led to different views on the system by two different players within it. To allow a more concise view on the system and to support the readers and implementer, two perspectives shall be kept in mind. They have different views on the process but need to interact with each other seamlessly.

1.2.1 Taler Exchange (C2EC)

The perspective of the *Taler Exchange* includes all processes within C2EC component and the interaction with the terminal application, terminal backend and the wallet of the user. The *Taler Exchange* wants to allow withdrawal of digital digital cash only to users who pay the equivalent value to the *Exchange*. The *Exchange* wants to stay out of any legal implications at all costs.

1.2.2 Terminal Application

The perspective of the terminal application includes all processes within the application which interacts with the user, their *Wallet* and credit card allowing the withdrawal of digital cash. The terminal application wants to conveniently allow the withdrawal of digital cash and charge fees to cover its costs and risks.

1.2.3 Taler Wallet

The *Wallet* holds the digital cash owned by the customer. The *Wallet* wants to eventually gather the digital cash from the *Taler Exchange*. The owner of the *Wallet* must therefore present their credit card at a *Terminal* of the terminal provider and pay the *Exchange* as well accept the fees of the provider.

1.3 Goal

The goal of this thesis is to propose a framework for cashless withdrawals and implement the process which allows withdrawing Taler using a credit card at a terminal of the terminal provider *Wallee*.

1.3.1 C2EC

Therefore a new component, named C2EC, will be implemented as part of the Taler Exchange. C2EC will mediate between the Taler Exchange and the terminal provider. This includes checking that the transaction of the debitor reaches the account of the Exchange and therefore the digital currency can be withdrawn by the user, using its Wallet.

1.3.2 Wallee POS Terminal

The Wallee payment terminal, also called Point of Sales (POS) terminal, interfaces with payment cards (Credit Cards, Debit Cards) to make electronic fund transfers, i.e. a fund transfer to a given GNU Taler Exchange. For our purpose, we will extend the functionality of the terminal to initiate the corresponding counter payment from the exchange to the GNU Taler wallet of the payee.

2 Overview

2.1 Components

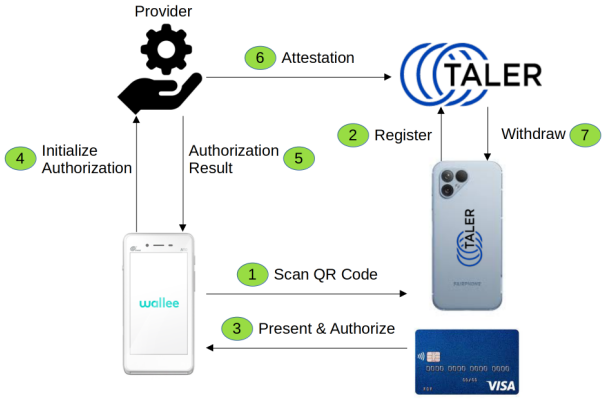

Figure 2.1: Involved components and devices

The component diagram shows the components involved by the withdrawal using the terminal. Besides the credit card owned by the user, two systems are involved and within each system two components are required to fulfill the task. The Taler ecosystem which represents the Taler Wallet and the Taler Exchange (C2EC is a part of the Exchange) involved in the withdrawal process. In the Terminal system, the terminal and the backend system of the terminal manufacturer are leveraged in the process.

2.2 Process

The figure 2.2 shows a high level overview of the components involved and how they interact. In an initial step (before the process is effectively started as depicted), the customer or owner of the terminal selects the *Exchange*, which shall be used for the withdrawal. The process is then started. The numbers in the diagrams are picked up by the description of the steps what is done between the different components:

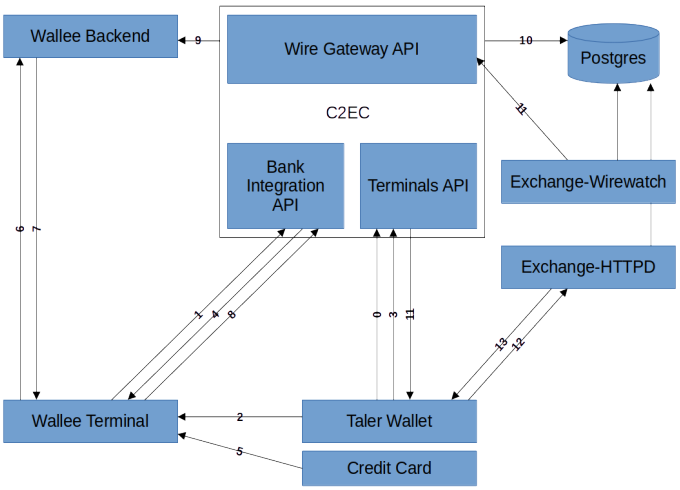

Figure 2.2: Diagram of included components and their interactions

1. Wallee Terminal requests to be notified when parameters are *selected* by C2EC.
2. The Wallet scans the QR code at the Terminal.
3. The Wallet registers a reserve public key and the *WOPID*.
4. The Bank-Integration API of C2EC notifies the Terminal, that the parameters were selected.
5. The POS initiates a payment to the account of the GNU Taler Exchange. For the payment the POS terminal requests a payment card and a PIN for authorizing the payment.
6. The Terminal triggers the payment at the Wallee Backend.
7. The Terminal receives the result of the payment.
   a) successful
   b) unsuccessful
8. The Terminal sends the payment notification to the Bank Integration API of C2EC.
9. The C2EC component approves the payment by requesting the transaction of the Wallee Backend.
10. C2EC updates the database by either setting the status of the withdrawal operation to *confirmed* or *abort* (depending on the response of the Wallee Backend).

11. Now decoupled from each other the Exchange-Wirewatch asks the Wire Gateway API of C2EC for a list of transactions and the Bank-Integration API sends a *confirmed* or *abort* message to the wallet.
12. The Wallet asks the Exchange to be notified, when a reserve with the reserve public key becomes available.
13. The Exchange can send the digital cash back to the Wallet.

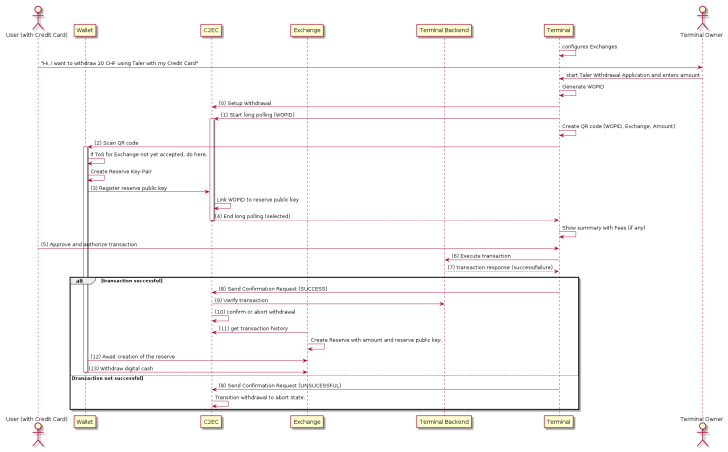

Figure 2.3: Process of a withdrawal using a credit card

The diagram in figure 2.3 shows the high level flow to withdraw digital cash using the credit card terminal and Taler. It shows when the components of figure 2.2 interact with each other. It shows the implementation of the flow. Terminal, Wallet and Exchange are linked leveraging a *WOPID* initially generated by the terminal and presented to the Exchange by the withdrawing Wallet accompanied by a reserve public key.

The process requires the Terminal, the Wallet, the C2EC component and the Exchange which interact with each other. In this section the highlevel process as showed in figure 2.3 is explained.

2.2.1 The Terminal

The Terminal initiates the withdrawal leveraging an application which works as follows:

1. At startup of the application, the terminal loads the C2EC configuration
2. When a user wishes to do a withdrawal, the owner of the terminal opens the application and initiates a new withdrawal. A withdrawal is basically a funds transfer to the IBAN account of the *Exchange*.
    a) Terminal sets up a withdrawal by aksing C2EC to setup a *WOPID*

   b) The application starts long polling at the C2EC and awaits the selection of the reserve parameters mapped to the *WOPID*. The parameters are sent by the Wallet to C2EC.

   c) *WOPID* is packed into a QR code (with Exchange and amount entered by the terminal owner)

   d) Terminal calculates fees and shows summary and the Terms of Service (ToS) of Taler.

   e) The user accepts the offer, agrees with the ToS

   f) QR code is displayed

3. The user now scans the QR Code using his Wallet.
4. The application receives the notification of the C2EC, that the parameters for the withdrawal were selected.
5. The Terminal executes the payment (after user presented their credit card, using the Terminal Backend).
6. The terminal initiate the fund transfer to the *Exchange*. The customer has to authorize the payment by presenting his payment card and authorizing the transaction with his PIN. The terminal processes the payment over the an available connector configured on the *Wallee Backend*. Possible connectors are Master Card, VISA, TWINT, Maestro, Post Finance, and others [3].

   a) It presents the result to the user.

   b) It tells the C2EC, that the payment was successful.

2.2.2 The C2EC

The C2EC component manages the withdrawal using a third party provider (e.g. Wallee) and seeks guarantees in order to create a reserve containing digital cash which can be withdrawn by the Wallet.

1. C2EC retrieves setup request for withdrawal which will lead to generation of the *WOPID*.
2. C2EC retrieves a long polling request for a *WOPID* (from the Terminal).
3. C2EC retrieves a request including a *WOPID* and a reserve public key.
4. C2EC validates the request and adds the key to the mapping. This establishes the *WOPID* to reserve public key mapping.
5. C2EC ends the long polling from the terminal.
6. C2EC receives confirmation request of the terminal.

7. C2EC verifies the notification by asking the provider backend for confirmation.
8. C2EC responds to an incoming transaction request of the Taler Wirewatch of the Exchange with the reserve public key of the withdrawal (which will eventually create a withdrawable reserve).

2.2.3 The Wallet

The Wallet must attest its presence to the terminal by registering a *WOPID* and belonging reserve public key which will hold the digital cash that can eventually be withdrawn by the Wallet.

1. The Wallet scans the QR Code (*WOPID*, Exchange information and amount) on the Terminal
2. It creates a reserve key pair
3. The Wallet sends the reserve public key and the scanned *WOPID* to the C2EC
4. The Wallet can withdraw digital cash from the created reserve.

3 Architecture

3.1 C2EC

The C2EC (**c**ashless**2ec**ash) component is the central coordination component in the cashless withdrawal of digital cash using Taler. It initializes the parameters and mediates between the different stakeholders of a withdrawal, which finally allows the customer to withdraw digital cash from a reserve owned by the *Exchange*. Therefore C2EC provides API which can be integrated and used by the *Terminal*, *Wallet* and the *Exchange*.

The API of the C2EC (cashless2ecash) component handles the flow from the creation of a C2EC mapping to the creation of the reserve. For the integration into the Taler ecosystem, C2EC must implement the Taler Wirewatch Gateway API [4] and the Taler Bank Integration API [5].

The exact specification can be found in the official Taler docs repository as part of the core specifications of the bank integration [5] and wire gateway [4]

3.1.1 C2EC Perspective

From the perspective of C2EC, the system looks as follows:

- Is requested by the *Taler Wallet* to register a new *wopid* to reserve public key mapping.
- Is notified by the *Terminal* (e.g. a Wallee terminal) about a payment.
- Attests a payment by requesting the payment proof at the *Provider Backend* (e.g. Wallee backend)
- Supplies the Taler Wire Gateway API that the respective *Taler Exchange* can retrieve fresh transactions and create reserves which are then created and can be withdrawn by the *Taler Wallet*.

3.1.2 Withdrawal-Operation state transitions

Basically C2EC mediates between the stakeholders of a withdrawal in order to maintain the correct state of the withdrawal. Therefore it decides when a withdrawal's status can be transitioned. The diagram in figure 3.1 shows the transi-

tions of states in which a withdrawal operation can be and which events will trigger a transition. The term attestation in this context means, that the backend of the provider was asked and the transaction was successfully processed (or not). So if a transaction was successfully processed by the provider, the final state is the success case *confirmed*, where the *Exchange* will create a reserve and allow the withdrawal. If the attestation fails, thus the provider could not process the transaction successfully, the failure case *aborted*, is reached as final state.

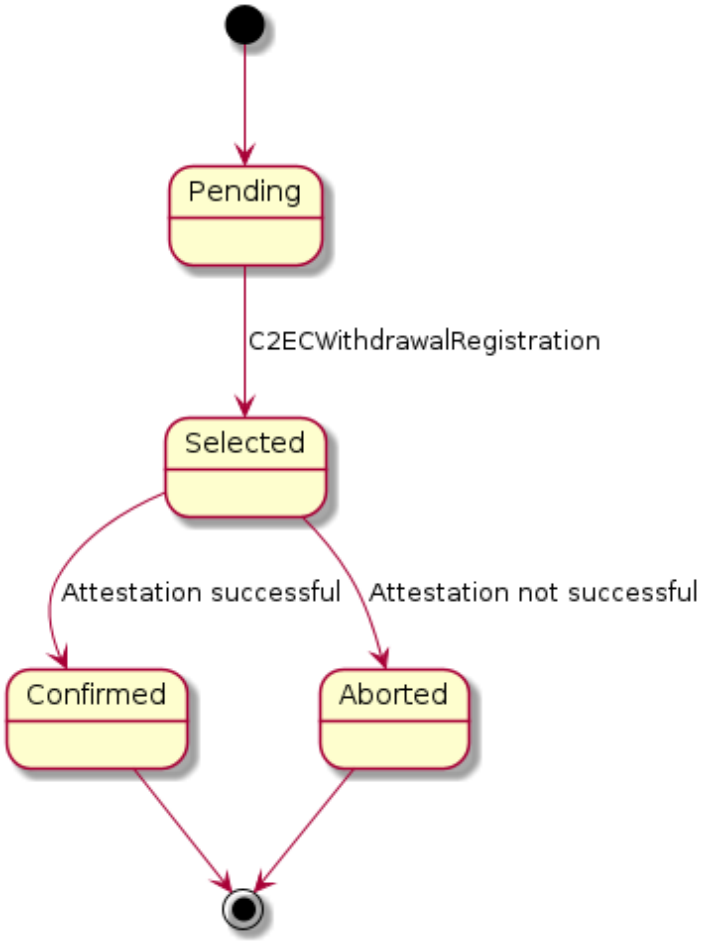

Figure 3.1: Withdrawal Operation state transition diagram

3.1.3 Authentication

Terminals and the Exchange Wire Watch which authenticate against the C2EC API using Basic-Auth [6] must provide their respective access token. Therefore, they provide a `Authorization: Basic $ACCESS_TOKEN` header, where `$ACCESS_TOKEN` is a basic-auth value configured by the operator of the exchange consisting of the

terminal username and password. The header value must begin with the prefix specified in RFC 7617 [6]: *Basic*.

3.1.4 The C2EC RESTful API

All components involved in the withdrawal process must interact with the C2EC component. Therefore this section describes the various API implemented in the C2EC component. The description contains a short list of the consumers of the respective API. Consumer in this context does not necessarily mean that data is consumed but rather that the consumer uses the API to either gather data or send reqeusts or data to C2EC.

Terminals API

That terminal can initiate and serve withdrawals in Taler, the Terminals API [7] is implemented, which mirrors all actions of a terminal at the C2EC component.

Config

- **Method:** GET
- **Endpoint:** /config
- **Description:** Return the protocol version and configuration information about the C2EC API.
- **Response:** HTTP status code 200 OK. The exchange responds with a `TerminalsConfig` object.
- **Consumers:** Terminals who want to use the API and therefore want to load the config of the instance.

Withdrawals

- **Method:** POST
- **Endpoint:** /withdrawals
- **Description:** Register a withdrawal operation at C2EC.
- **Response:** HTTP status code 200 OK. The *WOPID* generated by C2EC. On any other status code, terminate the withdrawal.
- **Consumers:** Terminals who want to initiate a withdrawal operation.

Status of the withdrawal operation

- **Method:** GET
- **Endpoint:** /withdrawals/$WOPID

- **Description:** Query information about a withdrawal operation, identified by the `WOPID`.
- **Response:** HTTP status code 200 OK and body containing a `BankWithdrawalOperationStatus` object or 404 Not found.
- **Consumers:** The API is used by the *Terminal* and *Taler Wallet* to retrieve information about the current state of the withdrawal operation. The API allows long-polling and can therefore be used by the consumer to be updated if the status of the withdrawal operation changes.

Check transaction

- **Method:** POST
- **Endpoint:** /withdrawals/$WOPID/check
- **Description:** Notifies C2EC about an executed payment for a specific withdrawal.
- **Request:** The request body contains a `WithdrawalConfirmationRequest` object.
- **Response:** HTTP status code 204 No content, 400 Bad request, 404 Not found, or 500 Internal Server error.
- **Consumers:** The API is used by the *Terminal* to notify the C2EC component that a payment was made and to give the C2EC component information about the payment itself (e.g. the provider specific transaction identifier or optional fees).

Fees

An important aspect of the withdrawal flow using third party providers are the fees. When the withdrawal operation is not supplied by some exchanges as standard service, the provider possibly wants to charge fees to the customer in order to make a profit. The provider might decide to delegate those fees to the customer and therefore fees can be sent to the C2EC component through the *check* API specified above. It's also possible that the service of withdrawing cash through a third party is causing costs to the merchant, which it does not want to cover on their own and therefore charge a fee to cover their costs. For example cashback is causing a lot of fees to the merchants supporting it.

> Die Händler zahlen jedoch für den Cashback-Service bereits Gebühren an die Banken. Aktuell sind es laut EHI im Schnitt 0,14 Prozent, insgesamt waren das 2023 rund 17,2 Millionen Euro. [8, Crefeld, ZEIT]

Terminal abortion

- **Method:** DELETE

- **Endpoint:** /withdrawals/$WOPID/abort
- **Description:** Aborts the withdrawal specified by the `WOPID`.
- **Request:** The request body contains a `WithdrawalConfirmationRequest` object.
- **Response:** HTTP status code 204 No content, 400 Bad request, 404 Not found, or 500 Internal Server error.
- **Consumers:** The API is used by the *Terminal* to notify the C2EC component that a payment was made and to give the C2EC component information about the payment itself (e.g. the provider specific transaction identifier or optional fees).

Taler Bank Integration API

Withdrawals by the *Wallet* with a C2EC are based on withdrawal operations which register a reserve public key at the C2EC component. The provider must first create a unique identifier for the withdrawal operation (the `WOPID`) to interact with the withdrawal operation (as described in sous-sous-section 3.1.4) and eventually withdraw digital cash using the *Wallet*. The withdrawal operation API is an implementation of the *Bank Integration API* [5].

Config

- **Method:** GET
- **Endpoint:** /config
- **Description:** Return the protocol version and configuration information about the C2EC API.
- **Response:** HTTP status code 200 OK. The exchange responds with a `C2ECConfig` object.
- **Consumers:** Wallets who want to use the API and therefore want to load the config of the instance.

3.1.5 Taler Wirewatch Gateway API

The Taler Wirewatch Gateway [4] must be implemented in order to capture incoming transactions and allow the withdrawal of digital cash. The specification of the Taler Wirewatch Gateway can be found in the official Taler documentation [4].

The wirewatch gateway helps the Exchange communicate with the C2EC component using a the API. It helps the Exchange to fetch guarantees, that a certain

transaction went through and that the reserve can be created and withdrawn. This will help C2EC to capture the transaction of the Terminal Backend to the Exchange's account and therefore allow the withdrawal by the customer. Therefore the wirewatch gateway API is implemented as part of C2EC. When the wirewatch gateway can get the proof, that a transaction was successfully processed, the exchange will create a reserve with the corresponding reserve public key.

For C2EC not all endpoints of the Wire Gateway API are needed. Therefore the endoints which are not needed will be implemented but always return http status code 400 with explanatory error details as specified by the specification.

Config

- **Method:** GET
- **Endpoint:** /config
- **Description:** Returns a `WireConfig` object with configuration information about the Wirewatch Gateway API of the C2EC component.
- **Response:** HTTP status code 200 OK and the wirewatch gateway configuration
- **Consumers:** Components who want to use the API and therefore want to load the config of the instance.

Transfer

- **Method:** POST
- **Endpoint:** /transfer
- **Description:** Allows the *Exchange* to make a transaction. This API is used in case of a refund. The transfer will therefore be pointed towards a provider specific payto-address (`payto://wallee-transaction` in case of Wallee).
- **Request:** The request contains a `TransferRequest` object.
- **Response:** HTTP status code 200 OK. The exchange responds with a `TransferResponse` object.
- **Consumers:** The *Exchange Wirewatch* who wants to transfer money using C2EC.

History of incoming transactions

- **Method:** GET
- **Endpoint:** /history/incoming
- **Description:** Returns a list of transactions which were recently created in the C2EC component. In case of C2EC, this are withdrawal operations which are confirmed and a reserve can therefore be created by the exchange.

- **Response:** HTTP status code 200 OK. The exchange responds with a `C2ECConfig` object.
- **Consumers:** The *Exchange Wirewatch* who will create the reserve which then can be withdrawn by the *Taler Wallet*.

History of outgoing transactions

- **Method:** GET
- **Endpoint:** /history/outgoing
- **Description:** Returns a list of transfers which were executed by the C2EC component.
- **Response:** HTTP status code 200 OK and a list of outgoing transfers.
- **Consumers:** The *Exchange Wirewatch* who will create the reserve which then can be withdrawn by the *Taler Wallet*.

3.1.6 C2EC Entities

The entities of the C2EC component must track two different aspects. The first is the mapping of a nonce (the `WOPID`) to a reserve public key to enable withdrawals and the second aspect is the authentication and authorization of terminals allowing withdrawals owned by terminal providers like *Wallee*.

The detailed explanation and ERD can be found in section 4.3.

Terminal Provider

Entity displayed in figure 4.11 describing providers of C2EC compliant terminals. The name of the provider is important, because it decides which flow shall be taken in order to attest the payment. For example will the name *Wallee* signal the terminal provider to trigger the attestation flow of *Wallee* once the payment notification for the withdrawal reaches C2EC.

Terminal

Entity displayed in figure 4.12 contains information about terminals of providers. This includes the provider they belong to and an access-token, which is generated by the operator of the C2EC component. It allows authenticating the terminal. A terminal belongs to one terminal provider.

Withdrawal

Entity displayed in figure 4.13 represents the withdrawal processes initiated by terminals. This table therefore contains information about the withdrawal like the amount, which terminal the withdrawal was initiated from and which reserve public key is used to create a reserve in the Exchange.

Relationships

The structure of the three entities forms a tree which is rooted at the terminal provider. Each provider can have many terminals and each terminal can have many withdrawals. The reverse does not apply. A withdrawal does always belong to exactly one terminal and a terminal is always linked to exactly one provider. These relations are installed by using foreign keys, which link the sub-entities (Terminal and Withdrawal) to their corresponding owners (Provider and Terminal). A provider owns its terminals and a terminal owns its Withdrawals.

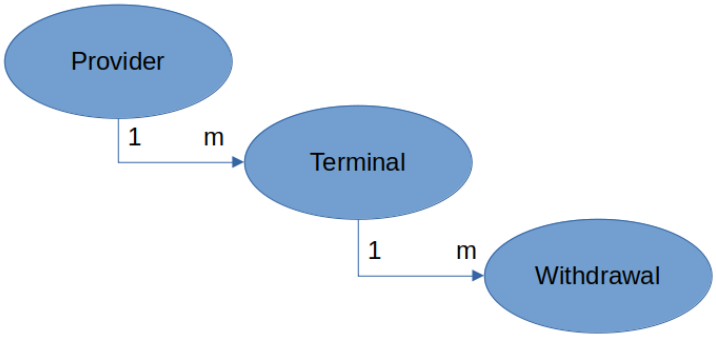

Figure 3.2: Relationships of the entities.

3.2 Payto wallee-transaction extension

RFC 8905 [9] specifies a URI scheme (complying with RFC 3986 [10]), which allows to address a creditor with theoretically any protocol that can be used to pay someone (such as IBAN, BIC etc.) in a standardized way. Therefore it introduces a registry which holds the specific official values of the standard. The registry is supervised by the GANA (GNUnet Assigned Numbers Authority) [11].

In case a refund becomes necessary, which might occur if a credit card transaction does not succeed, a new *target type* called *wallee-transaction* is registered. It takes a transaction identifier as *target identifier* which identifies the transaction for which a refund process shall be triggered. The idea is that the handler of the payto URI is able to deduct the transaction from the payto-uri and trigger the refund process.

3.2.1 Payto refund using Wallee

Wallee allows to trigger refunds using the Refund Service of the Wallee backend. The service allows to trigger a refund given a transaction identifier. Therefore the C2EC component can trigger the refund using the refund service if needed. The payto-uri retrieved as debit account by the wire gateway API, is leveraged to delegate the refund process to the Wallee Backend using the Refund Service and parsing the transaction identifier of the payto-uri.

3.2.2 Extensibility

The flow is extensible and other providers like Wallee might be added. They must therefore register their own refund payto-uri (if needed) with the GANA and then the refund process can be implemented likewise.

3.3 Taler Wallet

The *Taler Wallet* is responsible to create a reserve key pair which will allow him the withdrawal using the *Exchange* using the reserve public key of the key pair.

The reserve public key is created by the *Taler Wallet* and sent to C2EC to establish the mapping between the *wopid* and the reserve public key. The reserve public key is used to eventually create a reserve at the exchange which contains the digital cash. The *Taler Wallet* can then withdraw the digital cash from this reserve using the withdrawal process of the wallet [12]. The process for the case of C2EC is slightly different from the present processes because the requests to the Bank-Integration API contain different properties than the currently supported. This means the *Taler Wallet* must be extended in order to allow the withdrawal using C2EC.

3.3.1 Taler Wallet Perspective

From the perspective of the Wallet, the system looks as follows:

- Uses the QR Code displayed on the *Wallee Terminal* to identify nonce and read exchange information.
- Uses the Bank-Integration API of *C2EC* to register the reserve public key and retrieve information about the confirmation of the withdrawal.
- Uses the *Exchange* to withdraw the digital cash.

3.4 Wallee

Wallee offers level 1 PCI-DSS [13] compliant payment processes to its customers [14] and allows an easy integration of its process into various kinds of merchant systems (e.g. websites, terminals, etc).

3.4.1 Wallee Perspective

From the perspective of Wallee, the system looks as follows:

- Uses the Bank-Integration API of *C2EC* to get notified about parameter selection and inform *C2EC* about the payment.
- Needs the credit card of the customer in order to execute the payment.
- Uses the *Wallee Backend* to execute the payment using the supplied Android Till SDK sous-sous-section 3.4.2

3.4.2 Wallee Terminal

Wallee Terminals are based on android and run a modified, certified android version as operating system. Thus they can be used for payments and establish strong authentication in a trusted way.

Withdrawal Operation Identifier

The **W**ithdrawal-**OP**eration-**ID**entifier (*WOPID*) is leveraged by all components to establish the connection to an entry in the withdrawal table (figure 4.13) of C2EC. The *WOPID* is therefore crucial and every participant of the withdrawal must eventually gain knowledge about the value of the *WOPID* in order to process the withdrawal. The *WOPID* is created by the Terminal and advertised to the Exchange by requesting notification, when the reserve public key belonging to the *WOPID* was received and the mapping could be created. The Wallet gains the *WOPID* value when scanning the QR code at the Terminal and then sends the *WOPID* (and the other parameters) to the Exchange.

Creation of the WOPID

The creation of the *WOPID* is a crucial step in the process. The *WOPID* must be cryptographically sound. Therefore a cryptographically secure PRNG must be leveraged. Otherwise a *WOPID* might be guessed by an attacker. This would open the door for attacks as described in sous-section 4.5.1.

Wallee Till API

Wallee supplies the Wallee Android Till SDK [15] which allows the implementation of custom application for their android based terminals. The API facilitates the integration with the Wallee backend and using it to create payments.

3.4.3 Wallee Backend and API

Terminals of Wallee are used to communicate with the customer at the shop of the merchant. The payment and processing of the transaction is run on the *Wallee Backend*. The *Wallee Backend* is used by C2EC to attest a payment, when a `C2ECPaymentNotification` message reaches C2EC. The *Wallee Backend* is also used in order to do refunds, in case something goes wrong during the payment. Therefore the API of *Wallee Backend* is used to collect this information or process a refund. Wallee structures its API using *Services*. For C2EC this means that the *Transaction Service* [16] and *Refund Service* [17] must be implemented.

Transaction Service

The *Transaction Service* is used by C2EC to attest a transaction was successfully processed and the reserve can be created by the *Exchange*. Therefore the `GET /api/transaction/read` API of the *Transaction Service* is used. If the returned transaction is in state *fulfill*, the transaction can be stored as *completion_proof* for the withdrawal as specified in the withdrawal table figure 4.13 and the withdrawal status can be transitioned to *confirmed*. This will tell the *Exchange* to create the reserve which can eventually be withdrawn by the wallet.

Refund Service

The *Refund Service* is used by C2EC in case of a refund. Therefore the C2EC gets notified by the *Exchange* that the transaction shall be refunded. To trigger the refund process at the Wallee backend, the `POST /api/refund/refund` is used.

Wallee Transaction State

In order to decide if a transaction was successful, the states of a transaction within Wallee must be mapped to the world of Taler. This means that a reserve shall only be created, if the transaction is in a state which allows Taler not having any liabilities regarding the transaction and that Wallee could process the payment successfully. The documentation states that *only* in the transaction state *fulfill*, the delivery of the goods (in case of withdrawal this means, that the reserve can

be created) shall be started [18]. For the withdrawal this means, that the only interesting state for fulfillment is the *fulfill* state. Every other state means, that the transaction was not successful and the reserve shall not be created.

4 Implementation

The implementation is documented per component (C2EC, Terminal, Database). This means that each feature is documented from the perspective of the respective component in another section. Remarkable interactions with other components are linked with and shall support the reader to jump between the different components explanations interacting with each other. The reader is therefore advised to read the document on a digital device for easier reading. Also diagrams might be hard to see when reading the documentation from a hard copy.

4.1 C2EC

This section treats the implementation of the C2EC component. C2EC is the core of the withdrawal using a third party. Besides different API for different client types such as the Terminal, Wallet or the Exchange, it must also deal with background tasks as described in sous-section 4.1.3. The component also implements a framework to extend the application to accept withdrawals through other providers than Wallee. In sous-section 4.1.4 the requirements for the integration of other providers is explained and shown at the example of Wallee.

4.1.1 Endpoints

The diagram in figure 4.1 shows the perspective of the C2EC component in the withdrawal flow. The numbers in brackets represent can be mapped to the diagram in figure 2.3 of the process description in the architecture chapter at section 2.2. The requests represented in figure 4.1 only show the requests of the succesful path. In case of an error in the process, various other endpoints are implemented as described per client type in sous-section 4.1.1

The implementation of the terminals API can be found in sous-sous-section 4.1.2, the bank integration API is documented in sous-sous-section 4.1.2 and the wire gateway API implementation is documented in sous-sous-section 4.1.2

Decoupling steps using Events

The concept of publishers and subscribers is used heavily in the implementation. It allows decoupling different steps of the process and allows different steps to

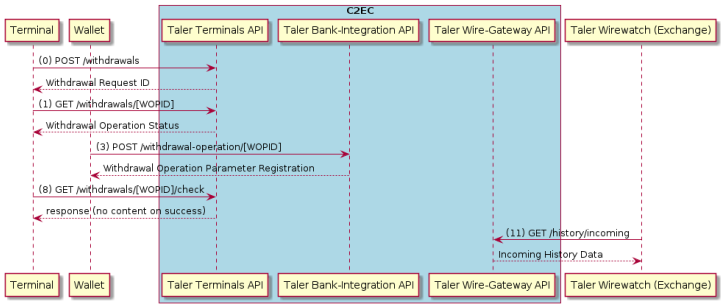

Figure 4.1: C2EC and its interactions with various components

be handled and executed in their own processes. Publishers can also be called notifiers or similar, while the subscriber can also be called listener or similar.

The communication of publishers and subscribers happends through channels. A publisher will publish to a certain channel when a defined state is reached. The subscriber who is subscribed or listens to this channel will capture the message sent through the channel by the publisher and start processing it.

The publish-subscribe scheme enables loose coupling and therefore helps to improve the performance of individual processes, because they cannot be hindered by others.

To decouple different steps in the withdrawal process an event based architecture is implemented. This means that every write action to the database will represent an operation which will trigger an event. The applications processes are listening to those events. The consumer of the API can wait to be notified by the API, by registering to those events via a long polling request at the API. This long-polling will then wait until the listener receives the event and return the received event to the consumer.

Following a short list of events and from whom they are triggered and who listens to them:

- Registration of the withdrawal operation parameters.
  - Registered by: Wallet
  - Listened by: Terminal
- Payment confirmation request sent to the Bank-Integration API of C2EC.
  - Registered by: Terminal
  - Listened by: Attestor
- Payment attestation success will send a withdrawal operation status update event.

  - Registered by: Attestor
  - Listened by: Consumers (via Bank-Integration-API)
- Payment attestation failure will trigger a retry event.
  - Registered by: Attestor
  - Listened by: Retrier
- Transfers which represent refunds in C2EC.
  - Registered by: Exchange (through the wire gateway API)
  - Listened by: Transfer

4.1.2 Abortion Handling

A withdrawal might be aborted through the terminal or the wallet. These cases are implemented through the respective *abort* endpoint in the bank-integration API figure 4.1.2 and terminals API figure 4.1.2. If in doubt whether to abort the withdrawal or not, it should be aborted. In case of abortion and failure cases, the security of the money is weighted higher than the user-experience. If the user must restart the withdrawal in case of a failure in the process, it is less severe than opening possible security holes by somehow processing the withdrawal anyway. On the other hand the system must be as stable as possible to make this error cases very rare. If they occur too often, the customer might not use the technology and therefore would make it worthless.

The withdrawal can only be aborted, when it is not yet confirmed by the attestation process (described in sous-sous-section 4.1.3).

Terminal API

This section describes the Implementation of the Terminal API [7].

The C2EC terminal API implements following endpoints:

- GET /config
- POST /withdrawals
- GET /withdrawals/[WOPID]
- GET /withdrawals/[WOPID]/check

The C2EC component does not implement the `/quotas/*` endpoints, since those are not relevant for the withdrawal using a payment terminal.

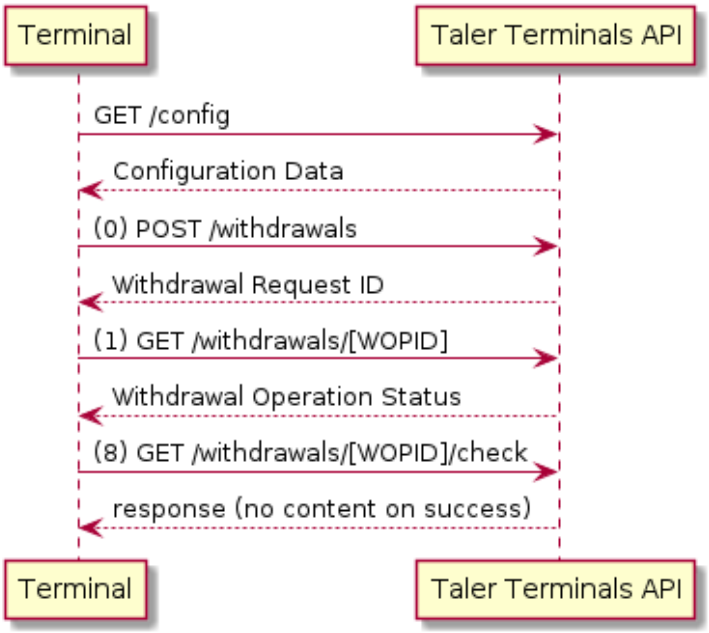

Figure 4.2: Terminals API endpoints

Configuration (/config)

This endpoint returns the configuration for the respective terminal. To support multi-provider setup, the respective provider is read from the basic-auth credentials sous-section 4.5.4. This means that the configuration response will be different when requesting the endpoint using a terminal from provider A than requesting from a terminal of provider B.

Setting up a withdrawal (/withdrawals)

The setup of a withdrawal generates the *WOPID* which is a cryptographically sound 32-Byte nonce and will be encoded using the base 32 crockford encoding [19]. The

cryptographical strength is crucial, because otherwise risks as described in sous-section 4.5.1 can materialise themself.

Terminals are advised to always set the *amount* field of the request, if they can define a fixed amount. This will force the Wallet to withdraw this exact amount and cannot be overwritten by it. The *suggested amount* field should only be used when the terminal cannot know how much money will be withdrawn (such as an ATM).

Status of withdrawal (/withdrawals/[WOPID])

When the terminal setup the withdrawal successful and received the *WOPID*, the terminal wants to wait before effectively authorizing the transaction until the Wallet has registered the parameters for the withdrawal. This endpoint allows this and supports long-polling such that the terminal may directly ask for the status after setting up the withdrawal. The endpoint is an exact replication of the Bank-Integration API status endpoint as described in figure 4.1.2

Trigger Attestation (/withdrawals/[WOPID]/check)

Once the terminal authorized the transaction at the providers backend and received the notification, that the transaction was processed at the providers backend, the terminal can trigger the attestation of the transaction by calling this endpoint. This is also the point where the terminal can know the fees of the provider (if any) and send them to the C2EC component.

Trigger Attestation (/withdrawals/[WOPID]/abort)

As long as the withdrawal was not authorized, it can be aborted by the terminal through this API. If the withdrawal was already authorized, the abortion will not work and the refund process might be needed to gain back the authorized money.

Taler Integration (/taler-integration/*)

Under the */taler-integration/* sub-path the Bank-Integration API is reachable. Endpoints under this subpath are used by the Wallet to register parameters of a withdrawal and ask for the status of a withdrawal operation. The endpoints of the Bank-Integration API are described in sous-sous-section 4.1.2

Taler Integration (/taler-wire-gateway/*)

The sub-path */taler-wire-gateway/* defines the location of the wire-gateway API used by the Taler Wirewatch component of the Exchange. It is used by the exchange to allow creation of withdrawable reserves. Therefore the wire gateway API was implemented as described in section sous-sous-section 4.1.2

Bank-Integration API

The Bank Integration API was implemented according to the specification [5].

Namely this are the following endpoints:

- GET /config
- GET /withdrawal-operation/[WOPID]
- POST /withdrawal-operation/[WOPID]
- POST /withdrawal-operation/[WOPID]/abort

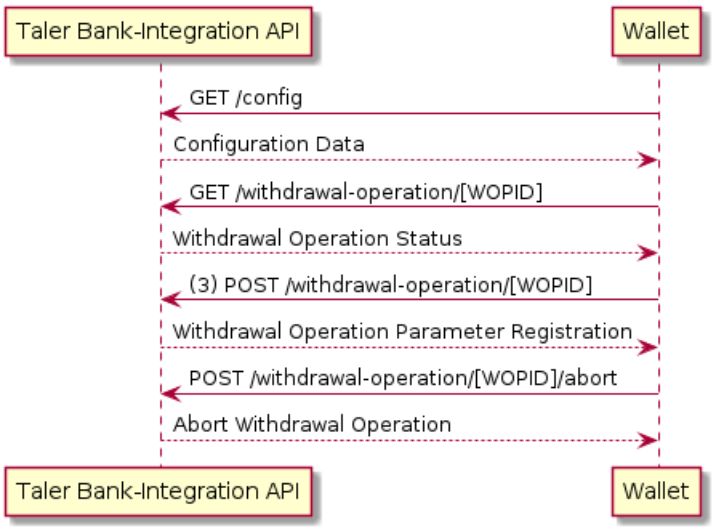

Figure 4.3: Bank-Integration API endpoints

Configuration (/config)

The configuration of the Bank-Integration endpoint is important for Wallets to check their compatibility and readiness. Also the currency specification can be retrieved by this endpoint, which allows the

Status of withdrawal (/withdrawal-operation/[WOPID])

The */withdrawal-operation/[WOPID]* endpoint returns the status of withdrawal operation. The endpoint enables long-polling through request parameters. This allows clients (the Wallet) to ask the Bank about a status before the status was reached. C2EC will then simply keep the connection open and either send a respond when a status change was registered or when the long-poll time exceeds.

Registering withdrawal parameters (/withdrawal-operation/[WOPID])

This endpoint is used by the Wallet to register the reserve public key generated by the Wallet, which will eventually hold the digital cash at the Exchange. This

reserve public key is unique and the API will return a conflict response if a withdrawal with the reserve public key specified in the request already exists. This is also the case if a mapping for the given *WOPID* was already created.

Aborting a withdrawal (/withdrawal-operation/[WOPID]/abort)

This endpoint simply allows the abortion of the withdrawal. This will change the status of the withdrawal to the *aborted* state.

Wire-Gateway API

The Wire-Gateway API [4] delivers the transaction history to the exchange which will create reserves for the specific public keys and therefore allow the customers to finally withdraw the digital cash using their wallet. Additionally it allows the Exchange to trigger transfers and keep track of executed transfers.

Following endpoints are implemented by the wire gateway API implementation:

- GET /config
- GET /history/incoming
- POST /transfer
- GET /history/outgoing

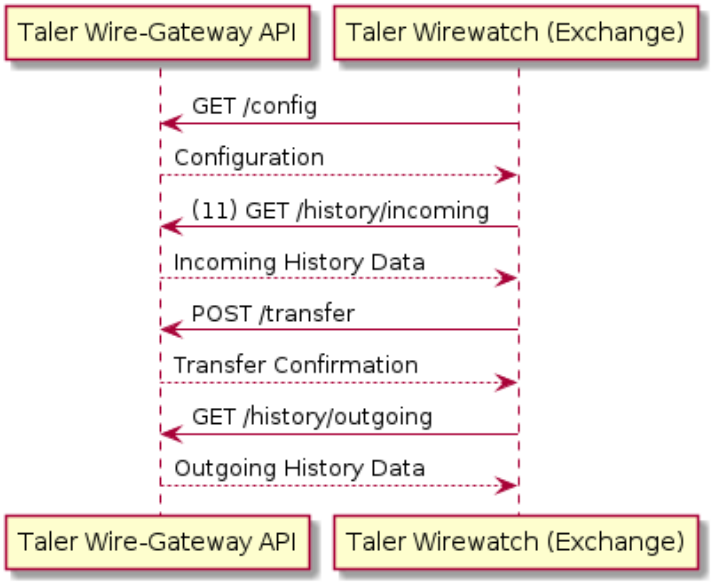

Figure 4.4: Wire-Gateway API endpoints

Configuration (/config)

The wire gateway configuration is used by the Exchange wirewatch component to check the compatibility of the component. This includes the check of the supported currency and the version.

Incoming transactions (/history/incoming)

The C2EC component needs to return incoming transactions by providers through the */history/incoming* endpoint. This will eventually create the reserve at the Exchange and therefore allow the customer to withdraw the digital cash using their Wallet. The

Transfers (/transfer)

The specification [4] requires the implementor of the API to keep track of incoming transfer requests. In order to guarantee the idempotence of the API, the implementation keeps track of all transfers in the database table *transfers*. It stores the transfer data in the database. If a request with the same UID is sent to the transfer-API, first it is checked that the incoming request is exactly the same as the previous one by comparing the request to the values stored in the database. Only if the values are the same, the transfer request is processed further. Otherwise the API responds with a conflict response.

Outgoing transactions (/history/outgoing)

The */history/outgoing* endpoint works in the same fashion as the */history/incoming* endpoint. But it will not return a list of confirmed withdrawals, but rather the list of successfully executed transfers registered using the */transfer* endpoint.

4.1.3 Processes

This section describes the different processes running in the background transitioning the state of a withdrawal. These transitions are triggered by the because of requests received by one of the components through the respective API.

Attestation

The attestation of a transaction is crucial, since this is the action which allows the exchange to create a reserve and can proof to the provider and customer, that the transaction was successful and therefore can put the liability for the money on the provider. The attestation process is implemented using a provider client interface and a provider transaction interface. This allows the process to be the same for each individual provider and new providers can be added easily by providing a specific implementation of the interfaces.

Attestation Retrier

If the attestation fails, but the transaction is not in the refund state as specified by the provider's transaction, the problem could simply be that the service was not available or the transaction was not yet processed by the provider's backend. In order to not need to abort the transaction directly and give the system some robustness, a retry mechanism was implemented which allows retrying the attestation step. This retry mechanism is run in a separate process started through the main process.

The retry will only be executed, when the transaction attestation failed because the transaction was not in the abort state or if for some reason the transaction information could not have been retrieved.

Transfer Retrier

The Exchange may send a transfer request to the C2EC component, due to the closing of a reserve or an issue. This will trigger a refund process at the providers backend. This refund process may fail and therefore like in the attestation case to increase the robustness of the system, a retry mechanism is implemented, which will retry the transfer before ultimatively failing the transfer.

Randomizing delays due to self synchronization

4.1.4 Providers

This section treats the integration of providers into the system by explaining the generic structures and showing how they were implemented for Wallee. It is also explained, what must be done in order to integrate other third parties into the system therefore showing the extensibility of the system.

Provider Client

The provider client interface is called by the attestation process depending on the notification received by the database upon receiving a payment notification of the provider's terminal. The specific provider clients are registered at the startup of the component and the attestation process will delegate the information gathering to the specific client, based on the notification received by the database.

The provider client interface defines three functions:

1. SetupClient: The setup function is called by the startup of the application and used to initialize the client. Here it makes sense to check that everything needed for the specific client is in place and that properties like access credentials are available.
2. GetTransaction: This function is used by the attestation process to retrieve the transaction of the provider system. It takes the transaction identifier supplied with the withdrawal confirmation request and loads the information about the transaction. Based on this information the decision to confirm or abort the transaction is done.
3. Refund: Since the transaction of the money itself is done by the provider, also refunds will be unwind by the provider. This functions mean is to trigger this refund transaction at the provider.

Provider Transaction

Since the attestation process is implemented to support any provider, also the transaction received by the provider clients *GetTransaction* function is abstracted using an interface. This interface must be implemented by any provider transaction which belongs to a specific provider client.

The provider client interface defines following functions:

1. AllowWithdrawal: This function shall return true, when the transaction received by the provider enters a positive final state. This means that the provider accepted the transaction and could process it. This means that the *Exchange* can create the reserve and allow the customer the withdrawal of the digital cash.

2. AbortWithdrawal: It doesn't mean that if a transaction does not allow to do the withdrawal, that the transaction shall be cancelled immediately. It could also be that the transaction was not yet processed by the provider. In this case we need means to check if the provider transaction is in an abort state if it is not ready for withdrawal, before aborting it. AbortWithdrawal shall therefore answer the question if the provider transaction is in a negative final state, which means the transaction is to be aborted.

3. Bytes: This function shall return a byte level representation of the transaction which will be used as proof of the transaction and stored in the exchanges database.

Wallee Client

The Wallee client is the first implementation of the provider client interface and allows the attestation of transactions using the Wallee backend system. The backend of Wallee provides a ReST-API to their customers, which allows them to request information about payments, refunds and so on. To access the API, the consumer must authenticate themself to Wallee by using their own authentication token as explained in sous-section 4.5.3.

As indicated by the provider client interface, two services of the Wallee backend are leveraged:

- Transaction service: The transaction service aims to provide information about a transaction registered using a Wallee terminal.

- Refund service: The refund service allows to trigger a refund for a given transaction using the transaction identifier. The refund will then be executed by the Wallee backend, back to the Customer.

To integrate Wallee as provider into C2EC, the provider client interface as described in sous-sous-section 4.1.4 was implemented. The transaction received by Wallee's transaction service implement the provider transaction interface as described in sous-sous-section 4.1.4.

Simulation Client

Additionally to the Wallee Client a Simulation Client was implemented which can be used for testing. It allows end-to-end tests of the C2EC component by stubbing the actions performed against a provider and returning accurate results.

Adding a new provider

To add a new provider, the client- and transaction-interfaces must be implemented as described in sous-sous-section 4.1.4 and sous-sous-section 4.1.4. The `SetupClient` function of the client interface must make sure to register itself to the global map of registered providers. Additionally, to the newly added provider implementation, the provider must also be registered in the database (section 4.6 describes how to achieve this). When the client adds itself to the registered providers clients, the application will load the provider client at startup of C2EC. If C2EC fails to find the specified provider in the database, it won't start. This behaviour makes sure, that only needed providers are running and that if a new provider was added, it is effectively registered and configured correctly (the setup function of the provider interface is responsible to check the provider specific configuration and do readiness or liveness checks if needed). If the new added provider requires a new payto target type, a new entry is to be created with the GANA in order to prevent conflicts in the future.

4.2 Wallee POS Terminal

4.2.1 Withdrawal flow

The process (figure 4.5) starts by first selecting the exchange and loading the configuration of the respective terminals-api. When this is successful, we will switch to the amount screen. Otherwise the withdrawal will be terminated.

On the amount screen the terminal operator enters the amount to withdraw and clicks on the "withdraw" button. If the operator clicks on the abort button, the withdrawal is terminated. When the user clicks the "withdraw" button, the terminal sets up the withdrawal at the exchanges terminals api and retrieves the wopid. When this step is unsuccessful, the withdrawal operation is aborted and terminated. Otherwise the terminal navigates to the register parameters screen.

In the register parameters screen, a QR code is displayed, which must be scanned by the withdrawer using their wallet app. The Terminal starts a long polling the terminals api to be notified, when the withdrawal operation is in state 'selected' which means, the wallet has successfully registered its withdrawal parameters. In this case the terminal application changes to the authorize payment screen in which the withdrawing person must authorize the transaction using their credit card. In any other case, the withdrawal operation is aborted and terminated. When the terminals backend sends the response of authorization, it sends the terminals api the check notification which tells the terminals api, that it can verify the payment at the providers backend now.

If this request is successful, the terminals shows a summary of the transaction and a button to leave the withdrawal activity. The wallet of the user should eventually be able to withdraw the amount authorized from the exchange.

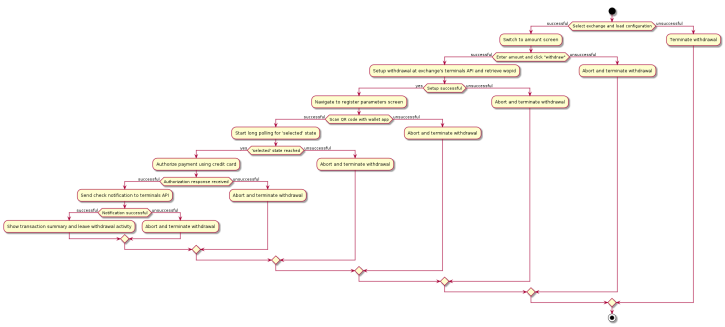

Figure 4.5: The flow of the terminal app

4.2.2 Screens

The Application is implemented using jetpack compose [20] and each of the screens described in sous-section 4.2.1 is implemented as composable screen. This allows

to handle the entire withdrawal flow in one single activity and therefore makes state handling easier, because the state of the withdrawal can be bound to the activity and also will be removed when the activity finishes or is terminated due to an error. It also prevents illegal states and that different withdrawals interfere each other. The state is maintained in a view model as described by androids documentation [21]. The withdrawal activity handles the lifecycle of the view model instance and initializes the routing of the screens using androids navigation controller as documented [22]. The navigation integration of android allows the declarative definition of the in-app routing and is defined at the creation of the withdrawal activity.

Choose Exchange Screen

On the screen figure 4.6 the user chooses the exchange to withdraw from. This allows the terminal to support withdrawals from various exchanges and therefore enhances the flexibility. When the user selected the exchange, the configuration of the exchange is loaded. This will define the currency of the withdrawal and tell the terminal where to reach the Terminals API of the C2EC server.

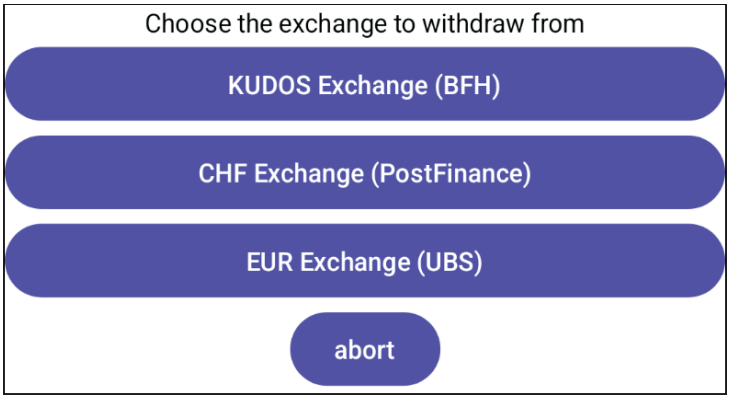

Figure 4.6: Terminal: Select the exchange to withdraw from

Amount Screen

The amount screen in figure 4.7 is used to ask the user what amount they would like to withdraw. When the amount was entered and the *withdraw*-button was clicked, the terminal sets up the withdrawal using the Terminal API. The Terminals API will send the *WOPID* to the terminal, which allows the terminal to generate the taler withdraw URI according to [23].

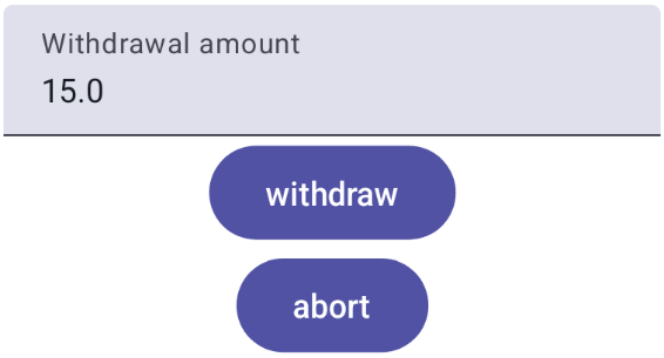

Figure 4.7: Terminal: Enter the desired amount to withdraw

Parameter Registration Screen

This screen in figure 4.8 displays a QR code which contains the taler withdraw URI of the withdrawal. This allows the customer to scan it using their Taler Wallet app and register the parameters for the withdrawal (namely the reserve public key). The withdrawal can be aborted on the screen. This step is important to make sure, that the customer has a working Taler Wallet installed and allows them to accept the terms of service for the respective exchange, if they did not yet registered the exchange on their wallet.

Authorization Screen

The authorization screen will use Wallee's *Android Till SDK* [15] to authorize the amount at the Wallee backend. The response handler of the SDK will delegate the response to the implementation of the terminal, which allows triggering the attestation of the payment by C2EC using the Terminals API. When the authorization process is not started and the transaction therefore is created at the backend system of Wallee, the screen figure 4.9 will be displayed. This signals the user, that the payment authorization must still be done and is about to be started. The user can abort the transaction at this point.

When the transaction was processed successfully, the summary of the transaction will be displayed on this screen as can be seen in figure 4.10.

4.2.3 Abortion Handling

During the flow various steps can fail or lead to the abortion of the withdrawal. Therefore these edge cases must be considered and handled the right way. Generally we can split the abortion handling on the terminal side into two different phases. The implementation of the Wallee POS Terminal therefore follows a strict

Figure 4.8: Terminal: Register withdrawal parameters

abort on failure strategy. This means that if anything goes wrong the withdrawal is aborted and must be started again. Generally the abortion handling strategy is to abort the withdrawal when in doubt and values security (of the money) over user-experience.

Abortion before authorization

The first phase are abortions *before* the payment is authorized. In this case the withdrawal operation can be aborted using the *abort* operation described in sous-sous-section 4.1.2. Every problem which cannot be recovered or not further processed must therefore lead to the abortion of the withdrawal.

Figure 4.9: Terminal: Waiting to start the authorization of the android till SDK

Abortion after authorization

When the transaction was authorized, the process is a little bit more complex. The customer has two possibilities. The first one is automatically covered with the given implementation, while the second is not guaranteed and needs manual interaction of the customer with the Taler Exchange operator.

Wait for automatic refund due to closing of the reserve The Taler Exchange configures a duration for which a reserve is kept open (and therefore can be withdrawn). When the configured duration exceeds the reserve is closed autmatically and the money transferred back to the customer. In the case of Wallee payments, this is realized through a refund request at the provider backend upon receiving a transfer request at the wire-gateway API sous-sous-section 4.1.2 of the C2EC component.

Manual request to refund money Depending on the operator of the Taler Exchange it might be possible to somehow manually trigger a refund and get back the money spent for the withdrawal.

Figure 4.10: Terminal: Payment authorized

4.3 Database

The Database is implemented using Postgresql. This database is also used by other Taler components and therefore is a good fit.

Besides the standard SQL features to insert, update and select data, Postgres also comes with handy features like LISTEN and NOTIFY.

This allows the implementation of neat pub/sub models allowing better performance, separation of concerns and loose coupling.

4.3.1 Schema

For the C2EC component the schema c2ec is created. It holds tables to store the entities described in sous-section 3.1.6. Additionally it contains the table for transfers which is used to capture refunds requested by the *Exchange*.

Terminal Provider

The *terminal provider* table holds information about the provider. It contains the information, which payto target type is used to make transactions by the provider. This information is needed in the refund case where the *Exchange* sends a transfer request. It also holds information about the attestation endpoint. Namely the base url and the credentials to authenticate the attestation process against the API of the providers backend. When adding the provider using the cli, the credentials are formatted in the correct way and also encrypted.

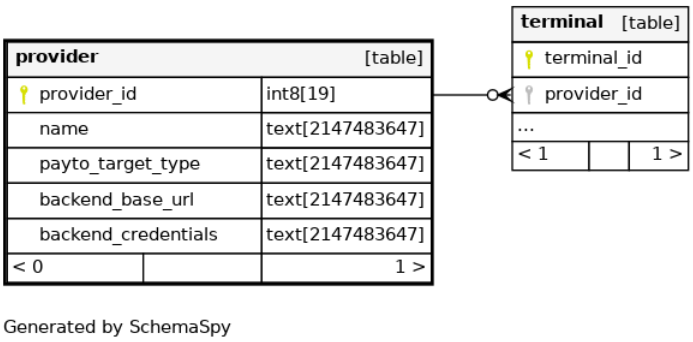

Figure 4.11: Terminal Provider Table

Terminal

Each Terminal must register before withdrawals are possible using the terminal. Therefore this table holds the information needed for withdrawals. A terminal can be deactivated by setting the *active* field accordingly. The terminals are authenticated using an access token generated during the registration process. Like adding the provider through the cli also the terminal access tokens will be encrypted using a PBKDF (namely argon2). The terminal is linked through the *provider_id* as foreign key to its provider. The *description* field can hold any information about the terminal which might be useful to the operator and help identify the device (location, device identifier, etc.). The operator will be asked for the respective values, when using the cli for the registration of the terminal.

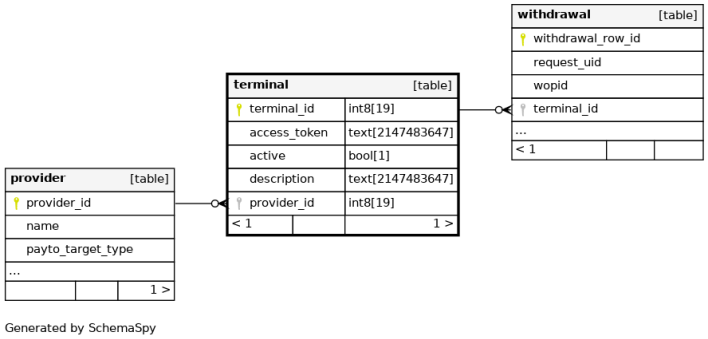

Figure 4.12: Terminal Table

Withdrawal

The withdrawal table is the heart of the application as it captures the information and state for each withdrawal. besides the obvious fields like *amount*, *wopid*, *reserve_pub_key* or *terminal_fees* (which all are directly related to one of the API calls described in sous-sous-section 4.1.2 or sous-sous-section 4.1.2), the table also holds the *terminal_id* which identifies the terminal which initiated the withdrawal. The *registration_ts* indicates, when the parameters of a withdrawal were registered. The field is mainly thought for manual problem analysis and has no direct functional impact. The *withdrawal_status* holds the current state of the withdrawal and is transitioned as described in sous-section 3.1.2. The *request_uid* is a unqiue identifier supplied by the terminal setting up a withdrawal. It is used to support idempotence of the API.

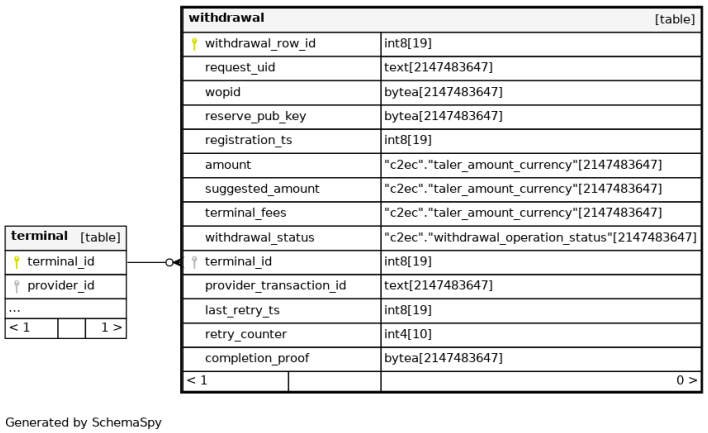

Figure 4.13: Withdrawal Table

Transfers

The transfer table is maintained through the transfer endpoint as described in sous-sous-section 4.1.2. A transfer in case of C2EC is constrained with a refund activity. The besides the fields indicated by the Wire Gateway API *request_uid, row_id, amount, exchange_base_url, wtid, credit_account* and *transfer_ts* which are all used to store information about the transfer, the fields *transfer_status* and *retries* are stored which allow retry behavior and help to make the system more robust. The *credit_account* is the refund payto URI which allows the refund process to be provider specific through a custom payto target type.

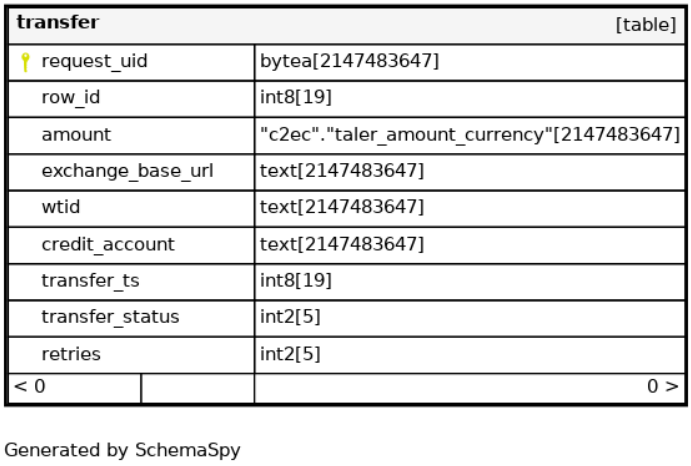

Figure 4.14: Transfer Table

Relationships

The relationships of the tables are created as described in sous-sous-section 3.1.6. A withdrawal belongs to a terminal and a terminal belongs to a provider. These relationships are implemented using foreign keys. The are specified to be non-null and therefore make sure, the chain of provider, terminal and withdrawal is always complete. The *transfer* table is unattached and lives by himself.

4.3.2 Triggers

Triggers are used to decouple the different sub processes in the withdrawal flow from one another.

The trigger runs a Postgres function which will execute a NOTIFY statement using Postgres built-in function *pg_notify*. Listeners in the application will capture those notifications and process them.

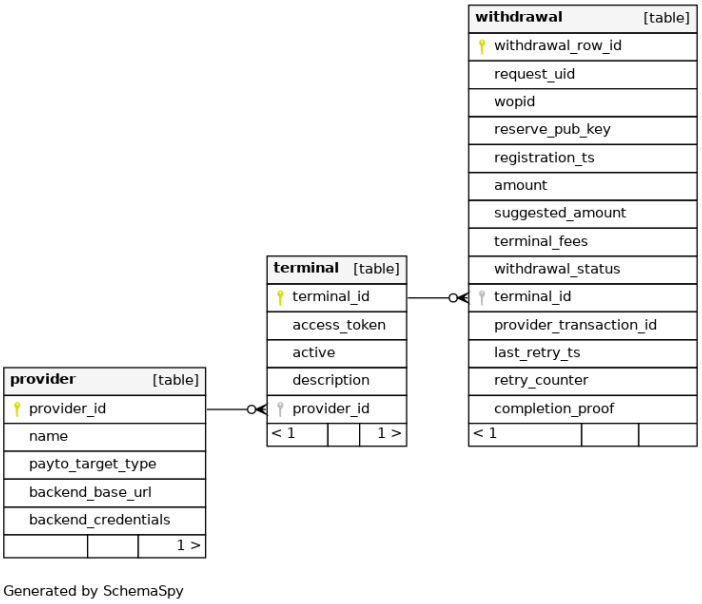

Figure 4.15: Relationships of the entities.

Withdrawal Status Trigger

The withdrawal status trigger emits the status of a withdrawal when the status is changed or the withdrawal is generated (inserted). The notification is sent through a channel which is named after the withdrawal using the *WOPID* in base64 encoded format. This allows a listener to specifically be notified about one specific withdrawal. This feature is used by the long poll feature of the status requests described in sous-sous-section 4.1.2 or sous-sous-section 4.1.2. By specifically listening to the withdrawal status to be changed for a *WOPID* the API can directly return, when a status change is received through the withdrawals channel.

Payment Trigger

The payment trigger is triggered through the withdrawal confirmation request of the Terminals API (described in sous-sous-section 4.1.2). It will start the attestation of the transaction at the providers backend, through the provider specific attestation process.

Attestation Retry Trigger

If the attestation for a withdrawal fails, this trigger is responsible to notify the retry listener of the application to retry the attestation. Therefore the trigger calls

the *emit_retry_notification* function which will notify its listener and the retry will eventually be executed.

Transfer Trigger

The transfer trigger is responsible triggering the transfer process by the application emiting the *request_uid* of the respective transfer, through the *emit_transfer_notification*.

4.4 Wallet

4.5 Security

4.5.1 Withdrawal Operation Identifier (WOPID)

The *WOPID* is the achiles heel of the withdrawal operation and therefore needs great care when generated. When the *WOPID* becomes somehow foreseeable, it opens the door for attackers allowing them to hijack the withdrawal from a remote location. Therefore the *WOPID* needs to leverage high entropy sources to be generated. This is achieved by using the crypto random library of Go. The library is part of the standard library and gains entropy through the entropy sources of the device running the application (in case of linux it is *getrandom(2)* which takes its entropy from */dev/urandom*, according to the documentation [24]).

4.5.2 Database Security

The database is very important as it decides wether to allow a withdrawal or not and it manages terminals and providers which hold sensitive credentials. Therefore two important aspects need to be considered.

Storing credentials

Even if a database leak occurs, it shall be very hard for the attacker to access the API using the credentials stored in the database. This is why credentials are stored using the PBKDF *argon2* [25]. *Argon2* is the winner of the password hashing competition initiated by the cryptographer Jean-Philippe Aumasson [25]. It is a widely adopted best practice approach for hashing passwords. Storing the hash of the credentials makes stealing credentials very hard and therefore prevents the abuse of credentials gathered through a database leak. The CLI described in section 4.6 implements operations which will register providers and terminals also hashing the credentials using *argon2*.

Access data through correct user

The database user executing a database query must have enough rights to execute its duties but not more. Therefore different database users are created for different tasks within the database. The described setup and installation process in section 4.8 will automatically generate the users and grant them the correct rights, when the respective variables are specified.

Table 4.1: Database users

Username	Component	Description
c2ec_admin	None	This user is possibly never to be used but during maintenance of the database itself (adding database users doing backups adding and granting users or others)
c2ec_api	C2EC	This user has all rights it needs to manage a withdrawal
c2ec_operator	CLI	This user shall be used by an operator of the C2EC component to add providers and terminals. It has no access to withdrawals

4.5.3 Authenticating at the Wallee ReST API

The Wallee API specifies four Wallee specific headers which are used to authenticate against the API. It defines its own authentication standard and flow. The flow builds on a message authentication code (MAC) which is built on a version, user identifier, and a timestamp. For the creation of the MAC the hash based message authentication code (HMAC) SHA-512 is leveraged which takes the so called *application-user-key* (which is basically just an access-token, which the user receives when creating a new API user) as key and the above mentioned properties plus information about the requested http method and the exactly requested path (including request parameters) as message [26]. The format of the message is specified like:

```
Version|User-Id|Unix-Timestamp|Http-Method|Path
```

- Version: The version of the algorithm
- User-Id: The user-id of the requesting user
- Unix-Timestamp: A unix timestamp (seconds since 01.01.1970)
- Http-Method: one of `HEAD`, `GET`, `POST`, `PUT`, `DELETE`, `TRACE`, `CONNECT`
- Path: The path of the requested URL including the query string (if any)

The resulting string must then be UTF-8 encoded according to RFC-3629 [27].

Wallee User Access rights

In order for Wallee to successfully authorize the user's requests, the API user must have the corret access rights. The C2EC Wallee API user must be able to access the transaction service for reading transactions and the refund service to write create refunds at the Wallee backend. Therefore following rigths must be assigned to the API user:

1. Refund-service
2. Transaction-Service

These rights can be assigned on Wallee's management interface by creating a role and assigning the rights to it. The role must then be added to the API user. The assignment of the roles must be done for the space context (Three different contexts are available. The relevant context is the space context, since requests are scoped to a space).

4.5.4 API access

Terminals API

The terminal API is accessed by terminals and the authentication mechanism is based on a basic auth scheme as specified by RFC-7617 [6] an specified in the terminals API specification [7]. Therefore a generated access-token used as password and a username which is generated registering the terminal using the cli explained in sous-section 4.5.5 are leveraged. Currently the terminal id and the provider name of the requesting terminal is included in the username part of the basic auth scheme.

Bank-Integration API

The Bank-Integration API is accessed by Wallets and specified to be unauthenticated.

Wire-Gateway API

The wire gateway specifies a basic authentication scheme [28] as described in RFC-7617 [6]. Therefore the C2EC component allows the configuration of a username and password for the exchange. During the request of the exchange at the wire gateway API, the credentials are checked.

4.5.5 Registering Providers and Terminals

A provider may want to register a new Terminal or maybe even a new provider shall be registered for the exchange. To make this step easier for the exchange

operators, a simple cli program (command line interface) was implemented (section 4.6). The cli will either ask for a password or generate an access token in case of the terminal registration. The credentials are stored has hashes using a PBKDF (password based key derivation function) so that even if the database leaks, the credentials cannot be easily read by an attacker.

4.5.6 Deactivating Terminals

A Terminal can be stolen, hijacked or hacked by malicious actors. Therefore it must be possible to disable a terminal immediately and no longer allow withdrawals using this terminal. Therefore the *active* flag can be set to *false* for a registered terminal. The Terminals-API which processes withdrawals and authenticates terminals, checks that the requesting terminal is active and is allowed to initiate withdrawals. Since the check for the *active* flag must be done for each request of a terminal, the check can be centralized and is implemented as part of the authentication flow. A Wallee terminal can be deactivated using the cli mentioned in sous-section 4.5.5.

4.6 C2EC CLI

The management of providers and terminals is not part of the thesis but since writing and issueing SQL statements is cumbersome and error-prone a small cli was implemented to abstract managment tasks. The cli tool was also shows the concepts a future implementation of the provider managment can use to integrate with the present features. The cli can be extended with more actions to allow the management of other providers and its terminals. Also the cli allows to setup the simulation terminal and provider which can be used for testing. Before commands can be executed, the user must connect the tool to the database which can be done throught the *db* command. With the aim to not introduce security risks by storing configuration state of the cli, the credentials must be entered after each startup of the cli. This can be surpassed by specifying postgres specific environment variables `PGHOST`, `PGPORT`, `PGUSER` and `PGPASSWORD` but remember that these environment variables might leak database credentials to others if not cleaned properly or set for the wrong users shell.

The cli was implemented to be usable and as it was out of scope of the thesis, the focus was on the functionality and tasks needed for the thesis and to allow an easy management of the terminals. This included features to manage wallee provider and terminals and the simulation. Additionally the tool implements commands to activate and deactivate a terminal, which makes the task much easier than writing and executing SQL by hand. Also it eliminates mistakes by reducing problems to bugs in the implementation of the cli.

4.6.1 Adding a Wallee provider

Adding the Wallee provider is as easy as calling *rp* (register-provider). It will then ask for properties like the base url and the credentials of the API user. Since the payto target type in case of Wallee will always be *wallee-transaction,* this is hard coded. The credentials supplied are encrypted using argon2 and stored as hash. Like this if the database leaks for some reason the credentials are still not easy to crack, when no standard password was used. Since Wallee generates those access tokens for their API user, this can be assumed to be the case.

4.6.2 Adding a terminal

Adding a Wallee terminal can be achieved by using the *rt* (register-terminal) command. It will ask the user to enter the description of the terminal and will then generate a 32-byte access token using Go's crypto random library which must be supplied to the owner of the terminal through a secure channel with the *terminal-user-id* (which is just the name of the operator and the id of the terminal separated by a dash '-')

4.6.3 Deactivating the terminal

To deactivate the terminal, the command *dt* must be issued. It will ask for the *terminal-user-id* of the terminal and then deactivate the specified terminal. The deactivation will be immediately and therefore helps to increase the security by allowing immediate action, when a terminal is come to be knwon hijacked, stolen or other fraud is detected specific to the terminal.

4.6.4 Setting up the Simulation

The Simulation provider and terminal allow to simulate transactions and interactions of the terminal with the API of C2EC. Therefore the command *sim* will setup the needed provider and terminal including the credentials of the simulation terminal, which must be saved and supplied to the operator through a secure channel. These credentials allow to test the Terminals API using the simulation terminal. The simulation client will not be available in productive environments to reduce the attack surface due to unnecessaty features.

4.7 Testing

Since the program leverages concurrency and operates in a distributed way, it is difficult to test besides unit testing. Therefore a simulation client and simulation

program was implemented which allows to test the C2EC component while simulating the different involved parties like the terminal, wallet and the providers backend system. This setup allows to test and therefore proof the functionality of the system.

The Simulation can be used for regression testing and therefore can be run before introducing new features in order to check, that existing functionality will not be broken.

Besides the automated tests, using the above mentioned simulation, unit tests were implemented for parsing, formatting and encoding functions. Additionally nanual test were fulfilled to ensure the system behaves correctly and without problems. To test the wire-gateway API, the *taler-exchange-wire-gateway-client* facility was used supplied by GNU Taler to verify the correct functioning of the API.

4.8 Deployment

4.8.1 Preparation

For the deployment the it is recommended to use a Debian Linux machine. To prepare the deployment of C2EC following steps must be done:

1. Machine which has bash, go and postgres installed must be prepared.
2. Three *different* passwords (each must be different and be stored in a secure location, like a password manager for example)
3. For the setup the username and password of postgresql superuser must be known.
4. The name for the database must be known and the database must exist at the target database system.
5. The installation location of C2EC must be created
6. The *setup* script in the root directory of cashless2cash must be altered with the values mentioned above.

For the deployment of the Wallee POS Terminal app, the following steps are necessary to prepare the usage of the cashless withdrawals leveraging Wallee:

1. A running deployment of C2EC must be accessible.
2. Wallee must be a registered provider at the C2EC instance.
3. The Terminal must be registered at C2EC.

4.8.2 Setup

Once the steps from the preparation were succesfully done, the *setup*-script can now be run. It will initiate the database and setup the users (as described in sous-sous-section 4.5.2) with the correct permissions. It will further generate the executables for C2EC, the cli and the simulation inside the specified `C2EC_HOME`. The setup script contains sensitive credentials and shall be deleted after using it. Maybe it can be stored in a save location like a password manager. Like this it will be still available in the future but will not lie around on the filesystem unencrypted.

Setting up Wallee as provider

To allow withdrawals using Wallee as provider, the correct access tokens must be created at the Wallee backend. Therefore a new application user must be created and the *application user key* must be saved to a password manager. Then Wallee must be registered at C2EC using the cli (described in section 4.6) and the *rp* command. There the space-id, user-id of the application user and the *application user key* must be provided. The cli will register the provider using these values.

When Wallee was registered as provider, one must register a terminal to allow access to the Taler Terminals API of C2EC. Therefore also the cli with its *rt* command can be used. It will generate the terminal user id and the access token. Both these values should be stored in a save location like the password manager

Setting up the simulation

When the simulation shall be installed the *prod*-flag in the C2EC configuration should be disabled, in order to allow the simulation provider to be registered at startup. This is a security measure, that testing facilities are not reachable in productive use of the system.

4.8.3 Deploy

When the provider and the terminal was successfully registered, the configuration located inside the `C2EC_HOME` must be adjusted to the correct values. Once this is done, the C2EC process can be started using `./c2ec [PATH-TO-CONFIGFILE]`.

4.8.4 Migration and releases

When a new version of the system shall be installed, the new executable can be built by issueing `make build`. After migrating the database using `make migrate` the newly built executable can be started.

5 Results

5.1 Discussion

What is the significance of your results? – the final major section of text in the paper. The Discussion commonly features a summary of the results that were obtained in the study, describes how those results address the topic under investigation and/or the issues that the research was designed to address, and may expand upon the implications of those findings. Limitations and directions for future research are also commonly addressed.

5.2 Results

What did you find? – a section which describes the data that was collected and the results of any statistical tests that were performed. It may also be prefaced by a description of the analysis procedure that was used. If there were multiple experiments, then each experiment may require a separate Results section.

5.3 Future Work

- Integrate other providers - Management interface for Terminals and Operators
- Automate registration of Terminals

Declaration of Authorship

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those acknowledged.

All statements taken from other writings, either literally or in essence, have been marked as such.

I hereby agree that the present work may be reviewed in electronic form using appropriate software.

May 20, 2024

J. Häberli

Bibliography

[1] Fabio Panetta. A digital euro that serves the needs of the public: striking the right balance. https://www.ecb.europa.eu/press/key/date/2022/html/ecb.sp220330_1~f9fa9a6137.en.html, March 2022.

[2] on behalf of ECB Kantar Public (Verian since November 2023). Study on new digital payment methods. https://www.ecb.europa.eu/euro/digital_euro/investigation/profuse/shared/files/dedocs/ecb.dedocs220330_report.en.pdf, March 2022.

[3] Wallee. Payment connectors. https://app-wallee.com/connectors.

[4] Taler. Taler wire gateway http api. https://docs.taler.net/core/api-bank-wire.html.

[5] Taler. Taler bank integration api. https://docs.taler.net/core/api-bank-integration.html.

[6] Julian Reschke. The 'Basic' HTTP Authentication Scheme. RFC 7617, September 2015.

[7] Taler. Terminal api. https://docs.taler.net/core/api-terminal.html.

[8] Sven Crefeld. Supermärkte zahlen immer mehr geld an kunden aus. *Zeit*, 04 2024.

[9] Florian Dold and Christian Grothoff. The 'payto' URI Scheme for Payments. RFC 8905, October 2020.

[10] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, January 2005.

[11] GNUnet Project. The gnunet assigned numbers authority (gana). https://gana.gnunet.org/.

[12] Taler. Withdrawal. https://docs.taler.net/taler-wallet.html#withdrawal.

[13] PCI Security Standards Council. Pci data security standard. https://docs-prv.pcisecuritystandards.org/PCI%20DSS/Standard/PCI-DSS-v4_0.pdf.

[14] Wallee. Transaction states. https://app-wallee.com/de-de/doc/payment.

[15] Wallee. Android till sdk. https://github.com/wallee-payment/android-till-sdk.

[16] Wallee. Transaction service. https://app-wallee.com/de-de/doc/api/web-service#transaction-service.

[17] Wallee. Refund service. https://app-wallee.com/de-de/doc/api/web-service#refund-service.

[18] Wallee. Transaction states. https://app-wallee.com/de-de/doc/payment/transaction-process.

[19] Douglas Crockford. Base 32. https://www.crockford.com/base32.html.

[20] Developer-Android. Build better apps faster with jetpack compose. https://developer.android.com/develop/ui/compose.

[21] Developer-Android. Viewmodel overview. https://developer.android.com/topic/libraries/architecture/viewmodel.

[22] Developer-Android. Navigation. https://developer.android.com/guide/navigation.

[23] Christian Grothoff and Florian Dold. The 'taler' URI scheme for GNU Taler Wallet interactions. Internet-Draft draft-grothoff-taler-01, Internet Engineering Task Force, November 2022. Work in Progress.

[24] Golang Doc. rand. https://pkg.go.dev/crypto/rand.

[25] Jean-Philippe Aumasson. Password hashing competition. https://www.password-hashing.net.

[26] Wallee. Authentication. https://app-wallee.com/en-us/doc/api/web-service#_authentication.

[27] François Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, November 2003.

[28] Taler. Taler wire gateway http api. https://docs.taler.net/core/api-bank-wire.html#authentication.

List of Figures

2.1 Involved components and devices 5
2.2 Diagram of included components and their interactions 6
2.3 Process of a withdrawal using a credit card 7

3.1 Withdrawal Operation state transition diagram 12
3.2 Relationships of the entities. 18

4.1 C2EC and its interactions with various components 24
4.2 Terminals API endpoints 26
4.3 Bank-Integration API endpoints 28
4.4 Wire-Gateway API endpoints 30
4.5 The flow of the terminal app 36
4.6 Terminal: Select the exchange to withdraw from 37
4.7 Terminal: Enter the desired amount to withdraw 38
4.8 Terminal: Register withdrawal parameters 39
4.9 Terminal: Waiting to start the authorization of the android till SDK 40
4.10 Terminal: Payment authorized 41
4.11 Terminal Provider Table 42
4.12 Terminal Table 43
4.13 Withdrawal Table 43
4.14 Transfer Table 44
4.15 Relationships of the entities. 45

List of Tables

4.1 Database users 47

Listings

1 C2EC API specification 67

Glossary

This document is incomplete. The external file associated with the glossary 'main' (which should be called `thesis.gls`) hasn't been created.

Check the contents of the file `thesis.glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

If you don't want this glossary, add `nomain` to your package option list when you load `glossaries-extra.sty`. For example:

```
\usepackage[nomain]{glossaries-extra}
```

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "thesis"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "thesis"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

Appendix A

API

```
..
  This file is part of GNU TALER.

  Copyright (C) 2014-2024 Taler Systems SA

  TALER is free software; you can redistribute it and/or modify it under the
  terms of the GNU Affero General Public License as published by the Free Software
  Foundation; either version 2.1, or (at your option) any later version.

  TALER is distributed in the hope that it will be useful, but WITHOUT ANY
  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR
  A PARTICULAR PURPOSE.  See the GNU Affero General Public License for more details.

  You should have received a copy of the GNU Affero General Public License along with
  TALER; see the file COPYING.  If not, see <http://www.gnu.org/licenses/>

  @author Joel Haeberli

============================
The C2EC RESTful API
============================

.. note::

  **This API is experimental and not yet implemented**

```

```
27 This chapter describe the APIs that third party providers need to
       integrate to allow
28 withdrawals through indirect payment channels like credit cards or
       ATM.
29
30 .. contents:: Table of Contents
31
32 --------------
33 Authentication
34 --------------
35
36 Terminals which authenticate against the C2EC API must provide
       their respective
37 access token. Therefore they provide a ‘‘Authorization: Bearer
       $ACCESS_TOKEN‘‘ header,
38 where ‘$ACCESS_TOKEN‘‘ is a secret authentication token configured
       by the exchange and
39 must begin with the RFC 8959 prefix.
40
41 ----------------------------
42 Configuration of C2EC
43 ----------------------------
44
45 .. http:get:: /config
46
47   Return the protocol version and configuration information about
        the C2EC API.
48
49   **Response:**
50
51   :http:statuscode:‘200 OK‘:
52     The exchange responds with a ‘C2ECConfig‘ object. This request
         should
53     virtually always be successful.
54
55   **Details:**
56
57   .. ts:def:: C2ECConfig
58
59     interface C2ECConfig {
60       // Name of the API.
61       name: "taler-c2ec";
62
```

```
      // libtool-style representation of the C2EC protocol version, see
      // https://www.gnu.org/software/libtool/manual/html_node/Versioning.html#Versioni
      // The format is "current:revision:age".
      version: string;
    }

----------------------------
Withdrawing using C2EC
----------------------------

Withdrawals with a C2EC are based on withdrawal operations which register a withdrawal identifier
(nonce) at the C2EC component. The provider must first create a unique identifier for the withdrawal
operation (the ``WOPID`') to interact with the withdrawal operation and eventually withdraw using the wallet.

.. http:post:: /withdrawal-operation

  Register a `WOPID` belonging to a reserve public key.

  **Request:**

  .. ts:def:: C2ECWithdrawRegistration

    interface C2ECWithdrawRegistration {
      // Maps a nonce generated by the provider to a reserve public key generated by the wallet.
      wopid: ShortHashCode;

      // Reserve public key generated by the wallet.
      // According to TALER_ReservePublicKeyP (https://docs.taler.net/core/api-common.html#cryptographic-primitives)
      reserve_pub_key: EddsaPublicKey;

      // Optional amount for the withdrawal.
      amount?: Amount;

      // Id of the terminal of the provider requesting a withdrawal by nonce.
      // Assigned by the exchange.
      terminal_id: SafeUint64;
```

```
    }

  **Response:**

  :http:statuscode:`204 No content`:
    The withdrawal was successfully registered.
  :http:statuscode:`400 Bad request`:
    The ``WithdrawRegistration`` request was malformed or
       contained invalid parameters.
  :http:statuscode:`500 Internal Server error`:
    The registration of the withdrawal failed due to server side
       issues.

.. http:get:: /withdrawal-operation/$WOPID

  Query information about a withdrawal operation, identified by
     the ``WOPID``.

  **Request:**

  :query long_poll_ms:
    *Optional.*  If specified, the bank will wait up to
       ``long_poll_ms``
    milliseconds for operationt state to be different from
       ``old_state`` before sending the HTTP
    response.  A client must never rely on this behavior, as the
       bank may
    return a response immediately.
  :query old_state:
    *Optional.* Default to "pending".

  **Response:**

  :http:statuscode:`200 Ok`:
    The withdrawal was found and is returned in the response body
       as ``C2ECWithdrawalStatus``.
  :http:statuscode:`404 Not found`:
    C2EC does not have a withdrawal registered with the specified
       ``WOPID``.

  **Details**

    .. ts:def:: C2ECWithdrawalStatus

```

```
      interface C2ECWithdrawalStatus {
        // Current status of the operation
        // pending: the operation is pending parameters selection
            (exchange and reserve public key)
        // selected: the operations has been selected and is
            pending confirmation
        // aborted: the operation has been aborted
        // confirmed: the transfer has been confirmed and
            registered by the bank
        // Since protocol v1.
        status: "pending" | "selected" | "aborted" | "confirmed";

        // Amount that will be withdrawn with this operation
        // (raw amount without fee considerations).
        amount: Amount;

        // A refund address as ``payto`` URI. This address shall
            be used
        // in case a refund must be done. Only not-null if the
            status
        // is "confirmed" or "aborted"
        sender_wire?: string;

        // Reserve public key selected by the exchange,
        // only non-null if ``status`` is ``selected`` or
            ``confirmed``.
        // Since protocol v1.
        selected_reserve_pub?: string;
      }


.. http:post:: /withdrawal-operation/$WOPID

   Notifies C2EC about an executed payment for a specific
       withdrawal.

  **Request:**

  .. ts:def:: C2ECPaymentNotification

    interface C2ECPaymentNotification {

      // Unique identifier of the provider transaction.
      provider_transaction_id: string;
```

```
172
173       // Specifies the amount which was payed to the provider
              (without fees).
174       // This amount shall be put into the reserve linked to by
              the withdrawal id.
175       amount: Amount;
176
177       // Fees associated with the payment.
178       fees: Amount;
179     }
180
181   **Response:**
182
183   :http:statuscode:`204 No content`:
184     C2EC received the ``C2ECPaymentNotification`` successfully and
          will further process
185     the withdrawal.
186   :http:statuscode:`400 Bad request`:
187     The ``C2ECPaymentNotification`` request was malformed or
          contained invalid parameters.
188   :http:statuscode:`404 Not found`:
189     C2EC does not have a withdrawal registered with the specified
          ``WOPID``.
190   :http:statuscode:`500 Internal Server error`:
191     The ``C2ECPaymentNotification`` could not be processed due to
          server side issues.
192
193
194 --------------
195 Taler Wire Gateway
196 --------------
197
198 C2EC implements the wire gateway API in order to check for
      incoming transactions and
199 let the exchange get proofs of payments. This will allow the C2EC
      componente to add reserves
200 and therefore allow the withdrawal of the digital cash. C2EC does
      not entirely implement all endpoints,
201 because the it is not needed for the case of C2EC. The endpoints
      not implemented are not described
202 further. They will be available but respond with 400 http error
      code.
203
204 .. http:get:: /config
```

```
205
206   Return the protocol version and configuration information about
          the bank.
207   This specification corresponds to ``current`` protocol being
          version **0**.
208
209   **Response:**
210
211   :http:statuscode:`200 OK`:
212     The exchange responds with a `WireConfig` object. This request
            should
213     virtually always be successful.
214
215   **Details:**
216
217   .. ts:def:: WireConfig
218
219     interface WireConfig {
220       // Name of the API.
221       name: "taler-wire-gateway";
222
223       // libtool-style representation of the Bank protocol
              version, see
224       //
              https://www.gnu.org/software/libtool/manual/html_node/Versioning.html#Versioni
225       // The format is "current:revision:age".
226       version: string;
227
228       // Currency used by this gateway.
229       currency: string;
230
231       // URN of the implementation (needed to interpret 'revision'
              in version).
232       // @since v0, may become mandatory in the future.
233       implementation?: string;
234     }
235
236 .. http:post:: /transfer
237
238   This API allows the exchange to make a transaction, typically to
          a merchant.  The bank account
239   of the exchange is not included in the request, but instead
          derived from the user name in the
240   authentication header and/or the request base URL.
```

```
241
242   To make the API idempotent, the client must include a nonce.
          Requests with the same nonce
243   are rejected unless the request is the same.
244
245   **Request:**
246
247   .. ts:def:: TransferRequest
248
249     interface TransferRequest {
250       // Nonce to make the request idempotent.  Requests with the
              same
251       // ``request_uid`` that differ in any of the other fields
252       // are rejected.
253       request_uid: HashCode;
254
255       // Amount to transfer.
256       amount: Amount;
257
258       // Base URL of the exchange.  Shall be included by the bank
              gateway
259       // in the appropriate section of the wire transfer details.
260       exchange_base_url: string;
261
262       // Wire transfer identifier chosen by the exchange,
263       // used by the merchant to identify the Taler order(s)
264       // associated with this wire transfer.
265       wtid: ShortHashCode;
266
267       // The recipient's account identifier as a payto URI.
268       credit_account: string;
269     }
270
271   **Response:**
272
273   :http:statuscode:`200 OK`:
274     The request has been correctly handled, so the funds have been
          transferred to
275     the recipient's account.  The body is a `TransferResponse`.
276   :http:statuscode:`400 Bad request`:
277     Request malformed. The bank replies with an `ErrorDetail`
          object.
278   :http:statuscode:`401 Unauthorized`:
279     Authentication failed, likely the credentials are wrong.
```

```
  :http:statuscode:`404 Not found`:
    The endpoint is wrong or the user name is unknown. The bank
        replies with an `ErrorDetail` object.
  :http:statuscode:`409 Conflict`:
    A transaction with the same ``request_uid`` but different
        transaction details
    has been submitted before.

  **Details:**

  .. ts:def:: TransferResponse

    interface TransferResponse {
      // Timestamp that indicates when the wire transfer will be
          executed.
      // In cases where the wire transfer gateway is unable to
          know when
      // the wire transfer will be executed, the time at which the
          request
      // has been received and stored will be returned.
      // The purpose of this field is for debugging (humans trying
          to find
      // the transaction) as well as for taxation (determining
          which
      // time period a transaction belongs to).
      timestamp: Timestamp;

      // Opaque ID of the transaction that the bank has made.
      row_id: SafeUint64;
    }

.. http:get:: /history/incoming

  **Request:**

  :query start: *Optional.*
    Row identifier to explicitly set the *starting point* of the
        query.
  :query delta:
    The *delta* value that determines the range of the query.
  :query long_poll_ms: *Optional.*  If this parameter is specified
      and the
    result of the query would be empty, the bank will wait up to
        ``long_poll_ms``
```

```
314     milliseconds for new transactions that match the query to
            arrive and only
315     then send the HTTP response.  A client must never rely on this
            behavior, as
316     the bank may return a response immediately or after waiting
            only a fraction
317     of ``long_poll_ms``.
318
319   **Response:**
320
321   .. ts:def:: IncomingReserveTransaction
322
323     interface IncomingReserveTransaction {
324       type: "RESERVE";
325
326       // Opaque identifier of the returned record.
327       row_id: SafeUint64;
328
329       // Date of the transaction.
330       date: Timestamp;
331
332       // Amount transferred.
333       amount: Amount;
334
335       // Payto URI to identify the sender of funds.
336       debit_account: string;
337
338       // The reserve public key extracted from the transaction
              details.
339       reserve_pub: EddsaPublicKey;
340
341     }
```

Listing 1: C2EC API specification

Appendix B

Project Management

Gant Chart

Iterative approach

During the project, each week a plan is made which described the tasks for the week. The plan is made on paper and hanged above my desk so I can see it. I inform the thesis advisors during the weekly synch call and change them if needed. For the prioritisation of work, the project plan in section 5.3 was used. This iterative approach helps to adapt to changing requirements and environment fast. Since I am working alone on the project, there is no need for more methodological overhead or to implement some alibi project organisation. Requirements are captured as specifications within the Taler documentation repository or in the architecture section (chapitre 3). As part of the weekly planning I reflect the past work and therefore can change what I think is necessary. Questions and impediments are directly addressed through the channel and/or person I think can help me with it.

Appendix C

Meeting notes

17.01.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Kickoff
- Understanding the Task
- Device
- Taler

Questions

- What am I going to do?
- Which components are roughly involved?

Action points

- Setup Thesis Document
- GNU Taler Copyright Assignment
- SSH-Public Key for git
- Inspect taler-exchange-wirewatch

Decisions

- Implement process 'cashless2ecash' as part of Taler-Exchange
- Wallet initializes process by scanning QR code like in the 'cash2ecash' showcase

- cash2ecash was implented by the guy named "windfisch" on mattermost

20.02.2024

Participants

- Jung Florian
- Häberli Joel

Topics

- Introduce each other and explain ideas
- Discuss nonce2ecash draft
- Discuss who wants to do what

Action points

- I send Flo a plan of what I'm going to do until when (approximately)
- I update the sequence diagram as discussed and send the openapi spec to Flo for review.

Decisions

- We can establish a generic approach for both our cases. Therefore the abstraction of *Providers* will be implemented. The *Providers* abstract and generalize some endpoint which can accept digital cash in any form (Credit Card, Cash, and so on) and give the Exchange the guarantee, that the digital cash will eventually be transferred to the Exchange.
- The verification at the provider from the perspective of the exchange must be optional (withdrawing at an ATM will not get any better than the amount the ATM sends to the Excahnge in the payment notification). Therefore an additional request to the provider will not bring any benefit.

Notes

- Flo wants to create a Reserve containing digital cash from the ATM. He then wants to trigger a peer to peer transaction. And therefore this reserve deals as guarantee to the Exchange. This flow is possible if the provider is controlled, which in my case is not given (Wallee is a company and I cannot easily alter their source code to open a reserve)

22.02.2024

Participants

- Hiltgen Alain
- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Task description
- Deeper understanding of the topic established?
- I contacted Florian Jung (alias Windfisch) and we bespoke his work on cash2ecash.

Questions

- Repository of Wallee Application will be different than 'cashless2ecash'? No
- Wallee: Master Password? Password received by Ben
- Wallee: Which SDK to use? Till-SDK (API to Wallee-Backend)
- How do we want to handle different currencies? How about currencies like Bitcoin? Currency is determined by the currency of the exchange.

06.03.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- API Spec nonce2ecash
- Database Spec nonce2ecash

Questions

- How can I create a reserve from the mapping table?
- Taler / Wallee : Which nonce to use? How to generate the nonce? Is there a preferred kind to generate nonces within taler?
- Do we add a maximal limit amount for a withdrawal on the side of the Taler Exchange?

Action points

- write API specification in .rst format (see /docs/core/api-*.rst in taler docs git)
- use Bank integration API
- write SQL schema and generate UML using schema-spy instead of writing UML.

13.03.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- SQL Schema of nonce2ecash.

Action points

- Add rst file to official docs Repository
- Add proper versioning to the SQL script.

20.03.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Payto Specification.

Action points

- Specify payto-uri scheme in GANA repo

20.03.2024 - 2

Participants

- Grothoff Christian

- Häberli Joel

Topics

- Architecture
- Payto

Action points

- Look at Wire Gateway and Bank Integration API as specification of an API and not as individual components of Taler. C2EC must implement those specification in order to integrate into the Taler ecosystem.

27.03.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Discussion of the Architecture documentation
- Feedback of Ben and Christian

Action points

- Integrate Feedback into documentation
- Use official docs repo to specify the API (e.g. Bank-Integration API and Wire Gateway API specification)
- No meeting next week.

10.04.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Discussion of the C2EC code.

Action points

- Use ini-format to parse config
- Add support for PGHOST environment variable
- Rename config properties to be compliant with other Taler repositories.
  - serve
  - bind
  - unix-path-mode
  - etc.
- For the attestation there is the additional case that neither confirm nor abort is an option and instead retries are required.
- Remove doubled abstractions (Abstracting attestation is not necessary)

17.04.2024

Participants

- Hiltgen Alain
- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Midterm Meeting with Expert Alain Hitlgen.
- Sequence diagram

Action points

- Fix Bank-Integration API
- Fees must be shown during the payment on the terminal
- The Wire Gateway API must implement "/history/outgoing" and return entries of the transfer table.

24.04.2024

Participants

- Fehrensen Benjamin

- Grothoff Christian
- Taler App Team
- BFH Guests and Students
- Häberli Joel

Topics

- New Terminals API
- Exponential Backoff, Self-Synchronization

Action points

- Integrate new API
- The Book entry

01.05.2024

Participants

- Fehrensen Benjamin
- Häberli Joel

Topics

- Wallee Terminal Version
- Completion Behavior of the transaction

Action points

- Use Version 0.9.20 (not 0.9.12)

08.05.2024

Participants

- Fehrensen Benjamin
- Häberli Joel

Topics

- Submit APK to Wallee
- Server is online running C2EC
- The Book entry

Action points

- Supply Wallee and APK (as soon as possible)
- Poster

TEMPLATE

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

-

Questions

-

Action points

-

Decisions

-