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 June 11, 2024

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

Acknowledgements

I would like to thank Prof. Dr. Benjamin Fehrensen and Prof. Dr. Christian Grothoff for their support during the thesis. Their knowledge and feedbacks helped me to work towards the thesis objectives and the reflection of my work. They pushed me to implement the product and motivated me to work hard.

The GNU Taler team deserves a big thank you to discuss, reflect and sharpen the Terminals API which was an important part of the thesis.

Also I thank my colleagues from the class who motivated me during the thesis. Especially I would like to thank Jan Fuhrer for the nice Friday night coding sessions, Christian Blättler for the valuable discussion about GNU Taler and Andy Bigler for the exchange about Android applications. They were crucial to gain a better understanding of how the components work and how I must do the implementation.

Additionally, I would like to thank Meret Staub for her critical thoughts during the proofreading of the thesis.

Thank you to Bruno Fauser who generously provided the title picture to me.

Last but not least I thank Flurina from all my heart. You were not mad at me when I cancelled dinner, because I wanted to write some code.

Abstract

This thesis develops and implements a framework that allows for cashless withdrawals using GNU Taler, with the objective of increasing the easy onboarding and acceptance of GNU Taler as payment system. Currently, the GNU Taler payment system permits the withdrawal of digital cash using different means of payment. However, GNU Taler currently lacks the possibility using cashless payment means such as credit cards to withdraw digital cash. To address this gap, this thesis introduces a novel component, called cashless2ecash (C2EC), which establishes a reliable connection between the Taler ecosystem and payment service provider's terminals. The reference implementation establishes the process between the payment service provider Wallee and the GNU Taler Exchange by implementing the new Terminals API in C2EC. The implemented process guarantees the finality of the transaction to the GNU Taler Exchange and the terminal operator. The finality enables the withdrawal of digital cash using GNU Taler without the use of cash. The liability for the transaction is borne by the payment service provider, which assumes the guarantees for the GNU Taler Exchange.

Contents

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.

In March 2022, the European Central Bank (ECB) found that an **easy onboarding** procedure is one of the most important factors influencing the acceptance of the Digital Euro as a new payment system [1]. If the process of onboarding new users is straightforward, this will have a positive effect on the universal acceptance of digital cash using GNU Taler. The ECB asserts that universal acceptance, or the ability to "pay anywhere," is the most significant attribute of an effective digital payment instrument for consumers across the EU, regardless of age [2]. Therefore, an easy onboarding procedure is a crucial feature for digital cash to be adopted by the public.

The findings of the European Central Bank also extend to the GNU Taler, the software-based microtransaction and electronic payment system. For the GNU Taler to be widely accepted as a payment system, it is of utmost importance that the onboarding process for new users be as straightforward and user-friendly as possible. For this reason, it is essential that a variety of methods exist for the withdrawal of digital cash in Taler.

This thesis develops an additional withdrawal method by implementing a framework that allows cashless withdrawals in GNU Taler. Currently, it is possible to withdraw digital cash from a bank that operates a Taler Exchange and integrates the respective API. At the time of writing, only two banks are engaged in the process of establishing a Taler Exchange; GLS bank [3] and MagNet bank [4]. Furthermore, at the Bern University of Applied Sciences, an exchange is operated allowing the withdrawal of digital cash at the secretariat using cash.

To make the access to digital cash using Taler easier and allow a faster uptake 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 society.

To make the withdrawal using a credit card or other means of payment possible, the GNU Taler facilities must be extended and integrated with established payment service providers. The integration must enable the communication between the Taler ecosystem and payment service providers and their terminals.

To address this communication gap, this thesis introduces a new component, called cashless2ecash (C2EC), which establishes a reliable connection between the Taler ecosystem and payment service provider's terminals. The C2EC component enables the Taler 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 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).

Designing the user-experience along established patterns will lead to a better uptake of GNU Taler by enabling money to flow from existing payment systems into GNU Taler's digital cash.

1.2 Perspectives

To support readers and implementers, three 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 including the interfaces for the terminal application, terminal backend and the wallet of the customer. The Taler Exchange wants to allow withdrawal of digital digital cash only to users who pay the equivalent value to the Exchange. For this the Taler Exchange must make sure the payment is final on the side of the payment service provider. Otherwise the Exchange is at risk of loosing money.

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. Fees must be considered, since the withdrawal process is a service which costs the payment service provider money in form of integration and maintenance efforts. To cover its costs, the provider might want to add some fees on the withdrawal.

1.2.3 Taler Wallet

The wallet holds the digital cash owned by the customer. The wallet wants to eventually collect the digital cash from the Taler Exchange.

1.3 Goal

The objective of this thesis is to develop and implement a framework for the cashless withdrawal of digital cash in GNU Taler. The framework implements the process that digital cash in GNU Taler can be withdrawn at a terminal of an established payment service provider. The withdrawal process on the side of the provider terminal is implemented on the Paydroid platform, which is supplied by the payment provider *Wallee*.

The framework aims to achieve the following key objectives:

1. Finality: The operator of the Taler Exchange is not liable for any losses incurred in connection with the payment.
2. Convenience: The user experience adheres to established patterns.
3. Abort: The payment flow is robust and secure, and the option to abort transactions without the loss of money is available.

1.3.1 cashless2ecash (C2EC)

To achieve these goals C2EC is implemented as part of GNU Taler. C2EC mediates between the Taler Exchange and the terminal provider. This includes checking that the transaction of the debitor reaches the account of the Exchange and the digital cash can be withdrawn by the user using their wallet.

1.3.2 Paydroid Payment Terminal

The Wallee payment terminal, interfaces with payment cards (credit cards, debit cards) and other means of payment (e.g. Twint) to make electronic fund transfers, i.e. a fund transfer to a given GNU Taler Exchange. For our purpose, we 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

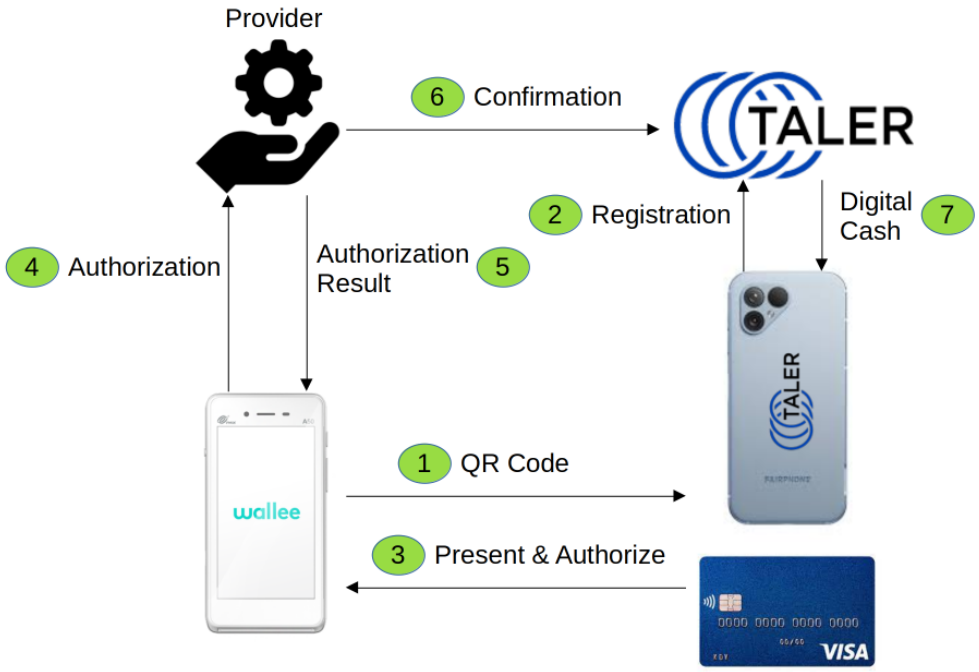

Figure 2.1: Involved components and devices

The component diagram in Figure 2.1 shows the components involved by the withdrawal using the terminal. Besides the mean of payment owned by the user, the Taler payment system and a payment service provider (such as Wallee) is involved.

To initiate the withdrawal, the wallet scans the QR code (1) and registers a reserve public key (2). After authorizing (3) the transaction using a credit card or other supported

payment means, the transaction is authorized (4) via the payment service provider backend. The payment service provider sends back an authorization result (5) before the C2EC component receives the confirmation of payment from the payment service provider (6). As soon the payment was confirmed the wallet can withdraw the digital cash from the Exchange (7).

2.2 Process

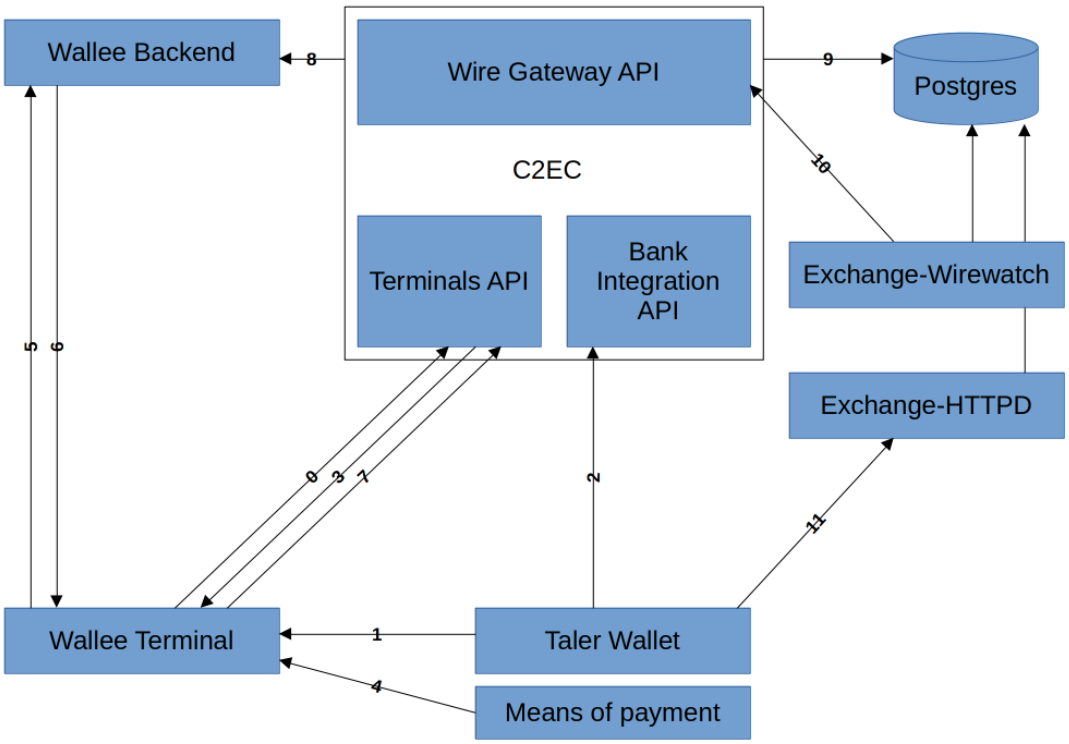

Figure 2.2: Diagram of included components and their interactions

Figure 2.2 shows the interactions of the components. In a initial step (before the process is effectively started as depicted), the customer or owner of the terminal selects the Exchange, which is to be used for the withdrawal. The process is then started and goes through the following steps:

0. The Wallee terminal requests to be notified when parameters are *selected* by C2EC.
1. The wallet scans the QR code at the terminal.

2. The wallet registers a reserve public key and initializes the mapping to the withdrawal operation identifier (*WOPID*).
3. The Terminals API of C2EC notifies the terminal, that the parameters were selected.
4. The payment terminal initiates a payment to the account of the GNU Taler Exchange. For the payment the payment terminal requests a payment mean and the verification such as a pin code to authorize the payment.
5. The terminal triggers the payment through the Wallee backend.
6. The terminal receives the result of the payment, which is either successful or not.
7. The terminal sends a payment confirmation request to the Bank Integration API of C2EC.
8. The C2EC component seeks confirmation for the payment by requesting the transaction of the Wallee backend.
9. The 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.
10. The Exchange-Wirewatch asks the Wire Gateway API of C2EC for a list of transactions. Confirmed transaction will lead to the generation of a reserve at the Exchange.
11. The wallet asks the Exchange to be notified, when a reserve with the reserve public key becomes available. The digital cash is then withdrawn by the wallet.

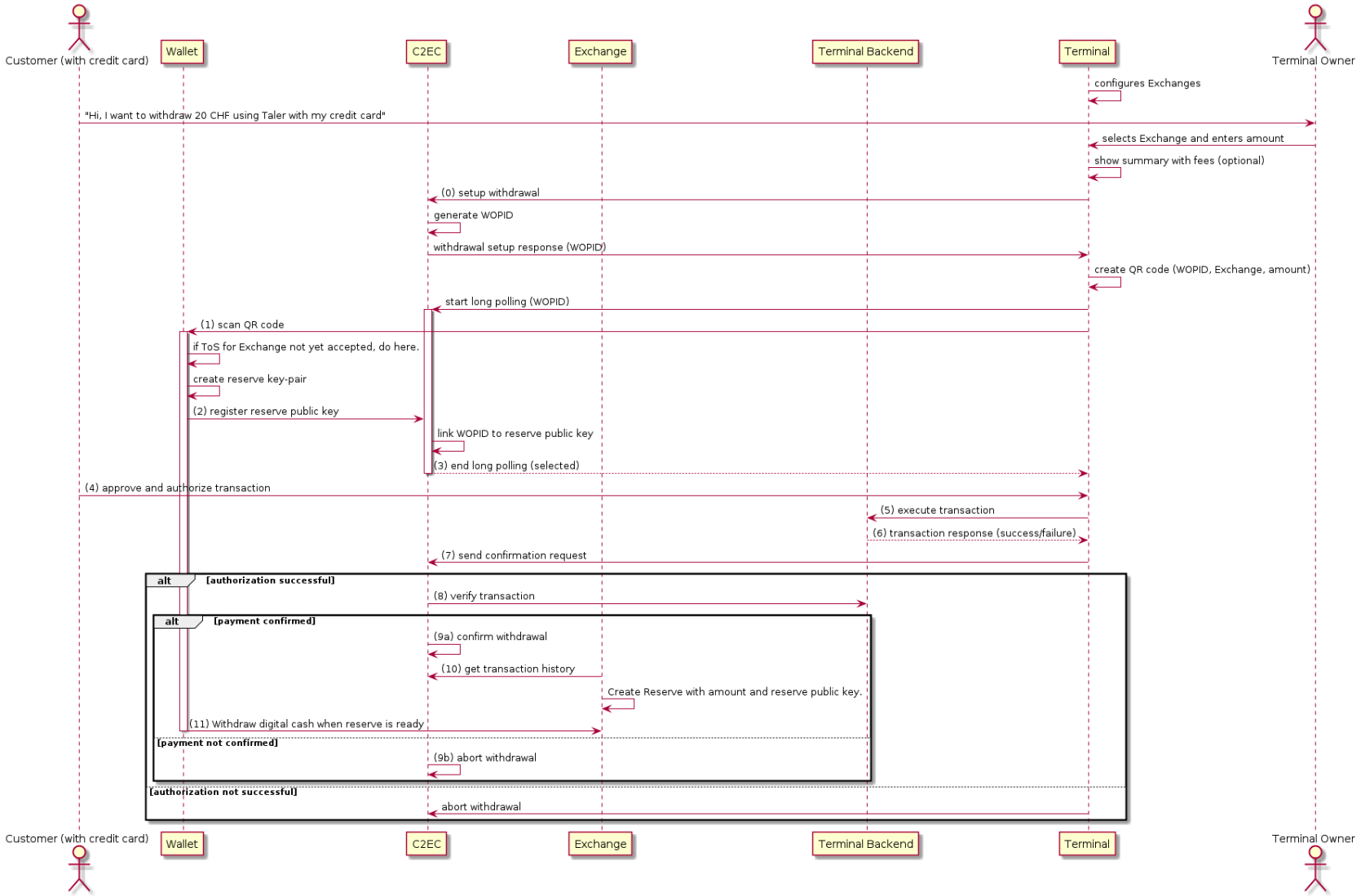

Figure 2.3: Process of a withdrawal using a credit card

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 who interact with each other. In this section the highlevel process is explained as showed in Figure 2.3.

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 entering the Amount to withdraw.
    a) The terminal sets up a withdrawal by asking C2EC to setup a *WOPID*.
    b) The terminal calculates fees and shows them to the customer.
    c) The *WOPID* is packed into a QR code (with Exchange and amount entered by the terminal owner).
    d) 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.
    e) The user accepts the offer and agrees with the ToS.
    f) The 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 possibly their pin. The terminal processes the payment over the an available connector configured on the Wallee backend. Possible connectors are for example Master Card, VISA, TWINT, Maestro, Post Finance, and others [5].

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. The C2EC component receives the setup request for withdrawal which will lead to generation of the *WOPID*.
2. The C2EC component receives a long polling request for a *WOPID* (from the terminal).
3. The C2EC component receives a request including a *WOPID* and a reserve public key.
4. The C2EC component validates the request and adds the key to the mapping. This establishes the *WOPID* to reserve public key mapping.
5. The C2EC component answers the long polling from the terminal.
6. The C2EC component receives the confirmation request of the terminal or is requested to abort the withdrawal.
7. The C2EC component verifies the notification by asking the provider backend for confirmation.
8. The C2EC component tells the Taler Wirewatch component of the Exchange about incoming transactions including 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 reserve public key with the respective *WOPID* which will hold the digital cash that can eventually be withdrawn by the wallet. The process of the wallet is already implemented through the Bank-Integration API [6] and only documented for completeness. The Bank-Integration API is implemented by C2EC.

1. The wallet scans the QR Code (*WOPID*, Exchange information and amount) on the terminal.
2. The wallet creates a reserve key pair.
3. The wallet sends the reserve public key to C2EC using the *WOPID* to map the public key to a withdrawal operation.

4. The wallet will be notified by the answer to a long-polling request when the digital cash is available at the Exchange's reserve belonging to the registered reserve public key.

3 Architecture

3.1 C2EC

The C2EC component is the central component in the cashless withdrawal of digital cash using Taler. It coordinates and initializes the parameters and mediates between the different stakeholders of a withdrawal. This finally allows the customer to withdraw digital cash from a reserve owned by the Exchange. To achieve this, C2EC provides an 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 Wire-Gateway API [7] and the Taler Bank Integration API [8]. The new Terminals API [9] is the interface of the payment service provider terminals to the GNU Taler ecosystem.

3.1.1 C2EC Perspective

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

- The C2EC component is requested by the Taler wallet to register a new *WOPID* to reserve public key mapping.
- Then, the C2EC component is notified by the terminal (e.g. a Wallee terminal) about a payment.
- The C2EC component confirms a payment by requesting the payment confirmation from the payment service provider backend (e.g. Wallee backend)
- The C2EC component runs the Taler Wire Gateway API that the respective Taler Exchange can retrieve fresh transactions and create reserves. This reserves are eventually withdrawn by the customer using their Taler wallet.

3.1.2 Withdrawal-Operation State Transitions

The C2EC component mediates between the stakeholders of a withdrawal in order to maintain the correct state of the withdrawal. It decides when a withdrawal's

status can be transitioned. The diagram in Figure 3.1 shows the transitions of states in which a withdrawal operation can be and which events will trigger a transition. The term confirmation 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 confirmation fails, indicating the provider could not process the transaction successfully, the failure case is *aborted*. *confirmed* and *aborted* are the final states.

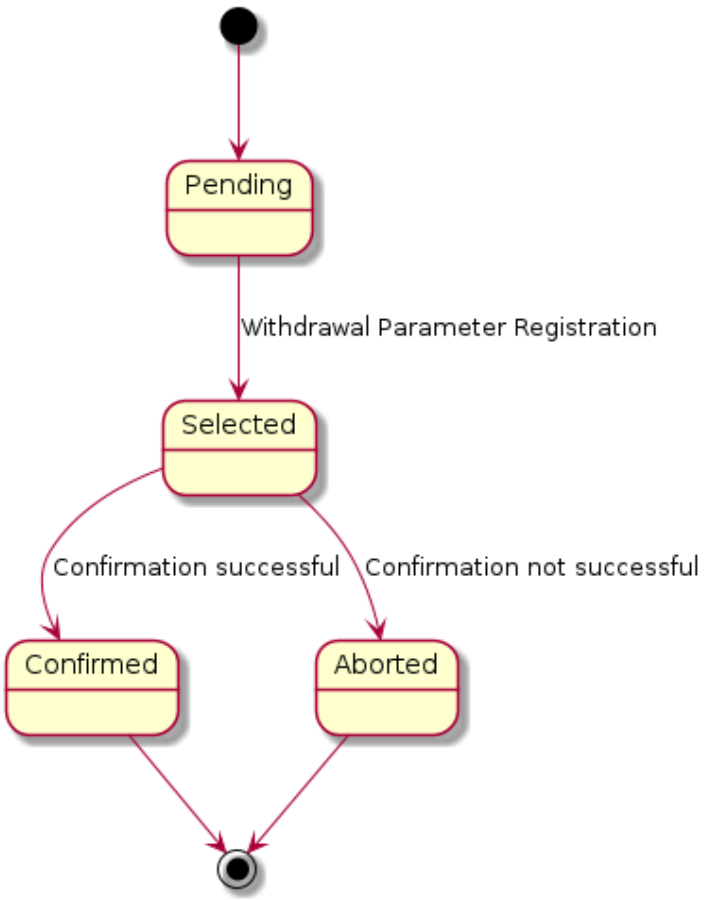

Figure 3.1: Withdrawal Operation state transition diagram

3.1.3 Authentication

Terminals and the Exchange wire watch process who authenticate against the C2EC API using Basic-Auth [10] must provide their respective access token. Fot his 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 [10]: *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 requests or data to C2EC.

Terminals API

That terminals can initiate and serve withdrawals in Taler, the Terminals API [9] is specified and implemented. The Terminals API mirrors all actions of a terminal at the C2EC component. This covers following endpoints:

1. Config (/config)
2. Withdrawal setup (/withdrawals)
3. Status of the withdrawal (/withdrawals/$WOPID)
4. Confirmation Request (/withdrawals/$WOPID/check)
5. Terminal side abort (/withdrawals/$WOPID/abort)

Fees

Fees are an important aspect of the withdrawal flow using established payment service providers. 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 and cover its costs. It is likely that these costs are rolled over to the customer in form of fees. This means that a terminal must have the capability to inform the Terminals API about fees. This can be achieved through the confirmation request in the Terminals API. Also the Exchange operator itself wants to charge fees to cover its costs. For example cashback is causing a lot of fees to the merchants supporting it:

> *de:*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. [11, Crefeld, ZEIT]

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 subsubsection 3.1.4) and eventually withdraw digital cash using the wallet. The withdrawal operation API is an implementation of the *Bank Integration API* [8].

Taler Wire-Gateway API

The Taler Wire-Gateway API must be implemented in order to capture incoming transactions and allow the withdrawal of digital cash. The specification of the Taler Wire-Gateway API can be found in the official Taler documentation [7].

The Wire-Gateway API enables the Exchange to communicate with the C2EC component using the API. The Exchange fetchs final withdrawals. The finality is guaranteed by C2EC by confirming the payment at the terminal backend. The Wire-Gateway API is implemented as part of C2EC. When the Exchange's wire watch process loads a confirmed withdrawal, the transaction was successfully processed. The Exchange will create a reserve with the corresponding reserve public key which can then be withdrawn by the wallet with the corresponding reserve private key.

The Wire-Gateway API of C2EC does not implement the testing endpoint */admin/add-incoming*. The endoint will respond with HTTP status code 501 (not implemented).

3.1.5 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*.

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

Terminal Provider

Figure 4.12 describes the provider entity 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 confirmation flow of *Wallee* once the payment notification for the withdrawal reaches C2EC.

Terminal

Entity displayed in Figure 4.13 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

The Entity displayed in Figure 4.14 represents the withdrawal processes initiated by terminals. This entity 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 form 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.

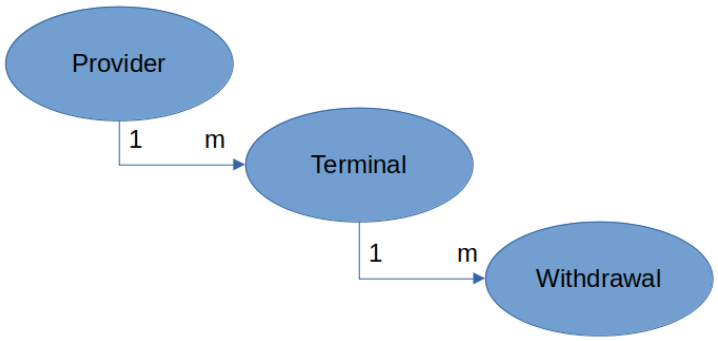

Figure 3.2: Relationships of the entities.

3.2 Wallee

Wallee offers level 1 PCI-DSS [12] compliant payment processes to its customers [13] and allows a simple integration of its process into various kinds of merchant systems (e.g. websites, terminals, etc).

3.2.1 Wallee Perspective

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

- The Wallee terminal uses the new Terminals API of C2EC to get notified about parameter selection and inform C2EC about the payment.
- The Wallee terminal needs the credit card (or other supported payment means) of the customer to authorize the payment.
- The Wallee terminal uses the Wallee Backend to authorize the payment using the supplied Android Till SDK subsubsection 3.2.2

3.2.2 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 4.14) of C2EC. The *WOPID* is therefore crucial and every participant of the withdrawal must eventually gain knowledge about the value of the *WOPID* 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 subsection 4.4.2.

Wallee Till API

Wallee supplies the Wallee Android Till SDK [14] which allows the implementation of custom application for their android based terminals. The API facilitates the integration with the Wallee backend and uses it to authorize payments.

3.2.3 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 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* [15] and *Refund Service* [16] 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 4.14 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

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 [17]. 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.

3.3 Payto Wallee-Transaction Extension

RFC 8905 [18] specifies a URI scheme (complying with RFC 3986 [19]), 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. It introduces a registry which holds the specific official values of the standard. The registry is supervised by the GANA (GNUnet Assigned Numbers Authority) [20].

In case a refund becomes necessary, which might occur if a transaction does not succeed or a reserve is not withdrawn within the specified time, 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.

3.3.1 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.

3.3.2 Extensibility

The flow is extensible and other providers like Wallee might be added. New payment service providers might want to register their own refund payto-uri with the GNU Assigned Numbers Authority (GANA) if needed. This will allow a simple integration of the refund process into the system. C2EC establishes structures which abstracts the general flow and the integration of new provider is simply adding new requests within this structures.

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.

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 subsection 4.1.3. The component also implements a framework to extend the application to accept withdrawals through other providers than Wallee. In subsection 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 the numbers of the diagram in Figure 2.3 depicting the process 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, abort endpoints are implemented as described per client type.

The implementation of the terminals API can be found in subsubsection 4.1.2, the bank integration API is documented in subsubsection 4.1.2 and the wire gateway API implementation is documented in subsubsection 4.1.2.

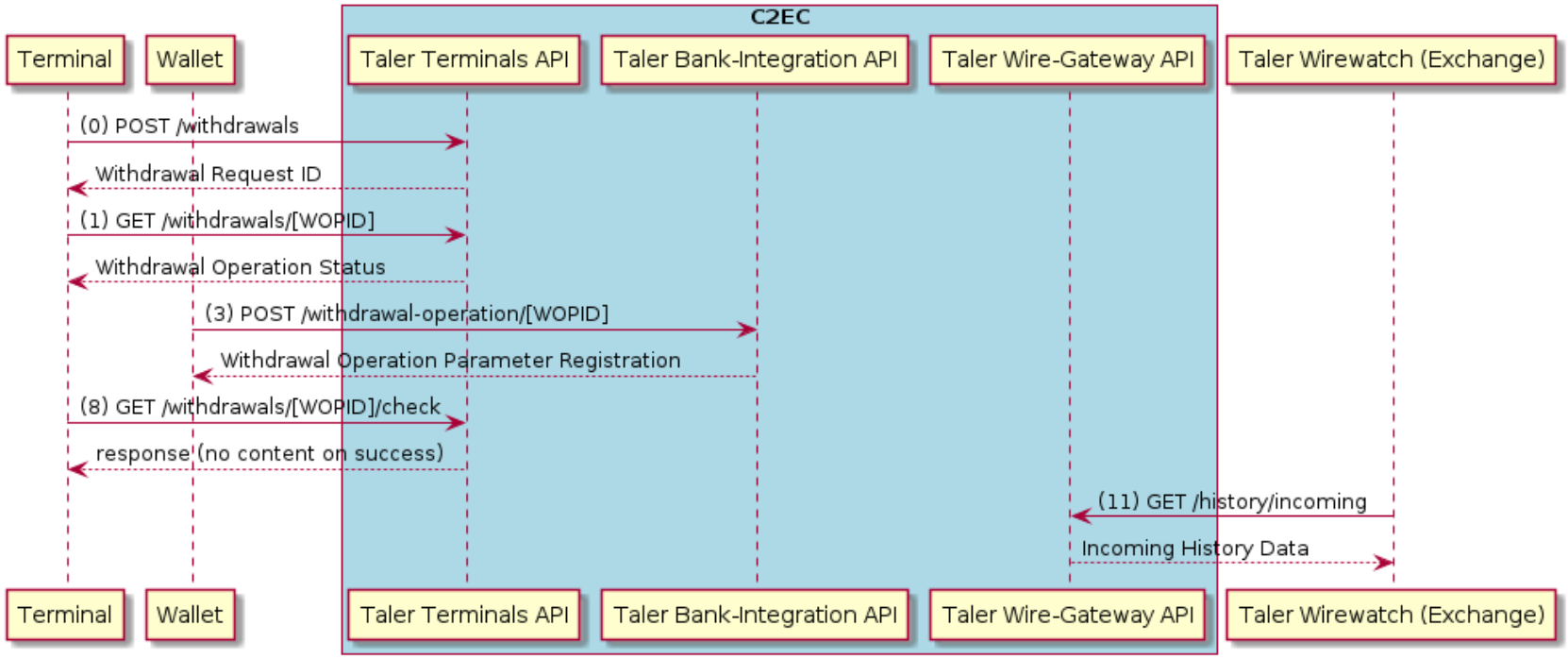

Figure 4.1: C2EC and its interactions with various components

Decoupling Withdrawal Steps

The concept of publishers and subscribers is used in the implementation. It allows decoupling different steps of the process and allows different steps to be handled and executed in their own processes. Subscriptions are implemented using long-polling and listeners for Postgres notifications. Long-polling imitates subscribers by rescheduling a request after some time and keeping the connection open until the specified timespan exceeds and the requested component answers the request.

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

Every action leading to a state transition of the withdrawal triggers an event. The trigger can be a real event trigger like a database trigger or a retry mechanism which periodically checks for state updates. The applications processes are listening to those events. Consuming clients such as the wallet or the terminal can wait to be notified by the API. The notification is achieved by registering the respective events via a long polling request. The long-polling request will then wait until the backend is ready to send the response. If the long-poll time exceeds and the result the consumer is looking for is not available, it must reschedule a long-poll request.

Following a short list of state transitions and from whom they are triggered and who awaits them:

- From pending to selected
  - Description: Registration of the withdrawal operation parameters.
  - Triggered by: Wallet
  - Awaited by: Terminal
- From selected to confirming
  - Description: Payment confirmation request sent to the Terminals API of C2EC.
  - Triggered by: Terminal
  - Awaited by: Confirmation process
- From selected to confirmed
  - Description: Payment confirmation success will send a withdrawal operation status update event.
  - Triggered by: Confirmation process

  - Awaited by: Consumer wallets (via Bank-Integration-API)
- From selected to aborted
  - Description: Payment confirmation failure will trigger a retry event.
  - Triggered by: Confirmation process
  - Awaited by: Retry process
- Refunds as transfer requests
  - Description: Transfers which represent refunds in C2EC.
  - Triggered by: Exchange (using the Wire Gateway API of C2EC)
  - Awaited by: Transfer process

4.1.2 Abort 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 subsubsection 4.1.2 and terminals API subsubsection 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 these 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 confirmation process (described in subsubsection 4.1.3). When the customer wants his money back they can wait for the reserve to be closed by the Exchange or get in touch with the operator who might trigger a manual refund. The manual refund will revert the payment and give the money back to the customer.

Terminal API

This section describes the Implementation of the Terminal API [9].

The C2EC Terminals 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. Quotas are checked by the payment service provider authorizing the payment.

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 subsection 4.4.5. 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. This configuration also supplies the base fees of the Exchange operator. These fees shall be communicated to the customer on the terminal and must be added to the withdraw amount. These fees are only the Exchange fees. The payment service provider might want to add their own (see subsection 4.1.5).

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 [21]. The cryptographical strength is crucial, because otherwise risks as described in subsection 4.4.2 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 or similar).

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 subsubsection 4.1.2

Trigger Confirmation (/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 confirmation 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. If for some reason it is not possible to know the fees here, potential fees can also be considered during the confirmation of the payment (this will lead to bad user-experience subsection 4.1.5).

Terminal side abort (/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 abort operation will not work and the refund process must be used to revert the authorized payment.

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 subsubsection 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 subsubsection 4.1.2

Bank-Integration API

The Bank Integration API was implemented according to the specification [8].

Namely this are the following endpoints:

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

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.

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.

Abort withdrawal (/withdrawal-operation/[WOPID]/abort)

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

Wire-Gateway API

The Wire-Gateway API [7] 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

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 [7] 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. The refund will always make a full refund. Partial refunds are not supported in the current implementation.

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 because of requests received by one of the components through the respective API.

Confirmation

The confirmation 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 confirmation 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 by providing a specific implementation of the interfaces.

Confirmation Retrier

If the confirmation 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. It allows retrying the confirmation step. This retry mechanism is run in a separate process started through the main process.

The retry will only be executed, when the transaction confirmation 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 confirmation 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 potential self synchronization issues

All processes doing retries use a randomized exponential backoff algorithm for scheduling the retries. The randomization prevents that the retry processes are all triggered at the exact same time and could crash the server due to heavy load. For the implementation a hard coded threshold of 20 percent of the targeted delay was chosen. The value can be adjusted by changing the constant.

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 confirmation 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 confirmation 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. FormatPayto: This function is responsible to create the correct payto URI given a withdrawal.

3. GetTransaction: This function is used by the confirmation 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.

4. 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. Before triggering the transaction, the refunded amount should be checked. The amount must not be bigger than the withdrawen amount. It can be smaller though, if the Exchange makes a partial refund.

Provider Transaction

Since the confirmation 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. This function is responsible to guarantee the **finality** (section 1.3).

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 a handle to check if the provider transaction is in an abort state. An abort state is a final state which will not change anymore. This indicates C2EC to stop retrying and abort the withdrawal.

3. Confirm: This function is called during the confirmation and contains business specific checks wether to confirm the payment or not. It must be separately implemented, because the transaction format varies between different payment system providers.

4. 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 confirmation 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 subsection 4.4.4.

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 subsubsection 4.1.4 was implemented. The transaction received by

Wallee's transaction service implement the provider transaction interface as described in subsubsection 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 Providers

Adding a new provider requires the implementation of the client- and transaction-interfaces as described in subsubsection 4.1.4 and subsubsection 4.1.4. The `SetupClient` function of the client interface must make sure to register itself to the global map of registered providers (accessible through `PROVIDER_CLIENTS`). Additionally, to the newly added provider implementation, the provider must also be registered in the database (section 4.5 describes how this could work). 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, adding a new entry to the GANA in order to prevent conflicts in the future might be a good idea.

4.1.5 Fees

During the implementation it appeared that there are several possible fee models, when thinking about the general case. Think of buying a bar of chocolate. From the perspective of getting what you want, you don't care if you pay 10 CHF or 10.10 CHF because in the end you get what you want - a bar of chocolate. Using a payment service provider to withdraw money, is special because if you want to withdraw 10 CHF then you want 10 CHF in your wallet and not 10 CHF minus the fees. When withdrawing money using the credit card you want the amount in your wallet you are asking for - not less. The fees must be transparently communicated to the customer, that they understand why the authorized amount will be higher than the amount they are asking to withdraw. The fees must be calculated in advance whenever possible. This leads to different models to add up fees. These four models were discovered:

Model 1 - Exchange Operator Fees

The payment system provider will charge its contractors (the merchants) in the background and expects the Exchange operator to calculate the fees to cover its costs. This means that the Exchange operator will specify its fees by approximating its costs to operate and maintain C2EC. Additionally a certain amount could be added on top of the calculated fees to make a profit.

Model 2 - Payment Service Provider Fees

The provider advertises the fees beforehand. The withdrawing terminal adds the fees before the transaction and tells them to C2EC. C2EC is not charging fees. In this case it must be checked during the confirmation that the amount confirmed is at least as big as the fees advertised to C2EC plus the amount to withdraw.

Model 3 - Payment Service Provider and Exchange Operator Fees

The combination of the first two models. The Exchange operator decides to add fees to withdrawals and the payment service provider adds fees as well. The implementation must combine the checks of model one and two.

Model 4 - Payment Service Provider "Late Fees"

A payment service provider might add fees but for some reason cannot tell them to C2EC before authorizing the transaction. In this case the confirmation process of the payment service provider needs to make sure that the fees are substracted from the withdrawal amount. Otherwise the Exchange operator must cover the costs, which will lead to loss of money. This must be prevented. This fee model

must be prevented when possible, because it leads to a bad user-experience. The amount which can be withdrawn will be lower than the amount authorized.

Mixing models

It could be a problem when mixing the different models in one instance of C2EC, because it could lead to conflicts. For example if a provider using the model one and a provider using the model two is operated within the same instance this could lead to more fees for the provider using model 2, since the fees of the Exchange operator should not be considered. In the future it might be possible to handle various fee models in one instance. This would require the implementation of all models.

4.2 Wallee Payment Terminal

4.2.1 Withdrawal Flow

The process (Figure 4.2) starts by first selecting the Exchange and loading the configuration from the Terminals-API. When this is successful the terminal app will navigate to the amount screen. Otherwise the withdrawal is 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. When the user scanned the QR Code and the terminal gets the withdrawal operation in state 'selected' from C2EC, the wallet has successfully registered its withdrawal parameters. In this case the terminal application changes to the authorization screen in which the withdrawing person must authorize the transaction using their credit card (or another supported payment mean). In any other case, the withdrawal operation is aborted and terminated. When the terminals backend sends the response of the authorization to the terminal, it sends the Terminals API of C2EC the confirmation request. This will start the confirmation process of the withdrawal immediately. If the confirmation request is successful, the terminal shows a summary of the transaction. Otherwise it shows that the transaction was not succesful and the withdrawal is aborted.

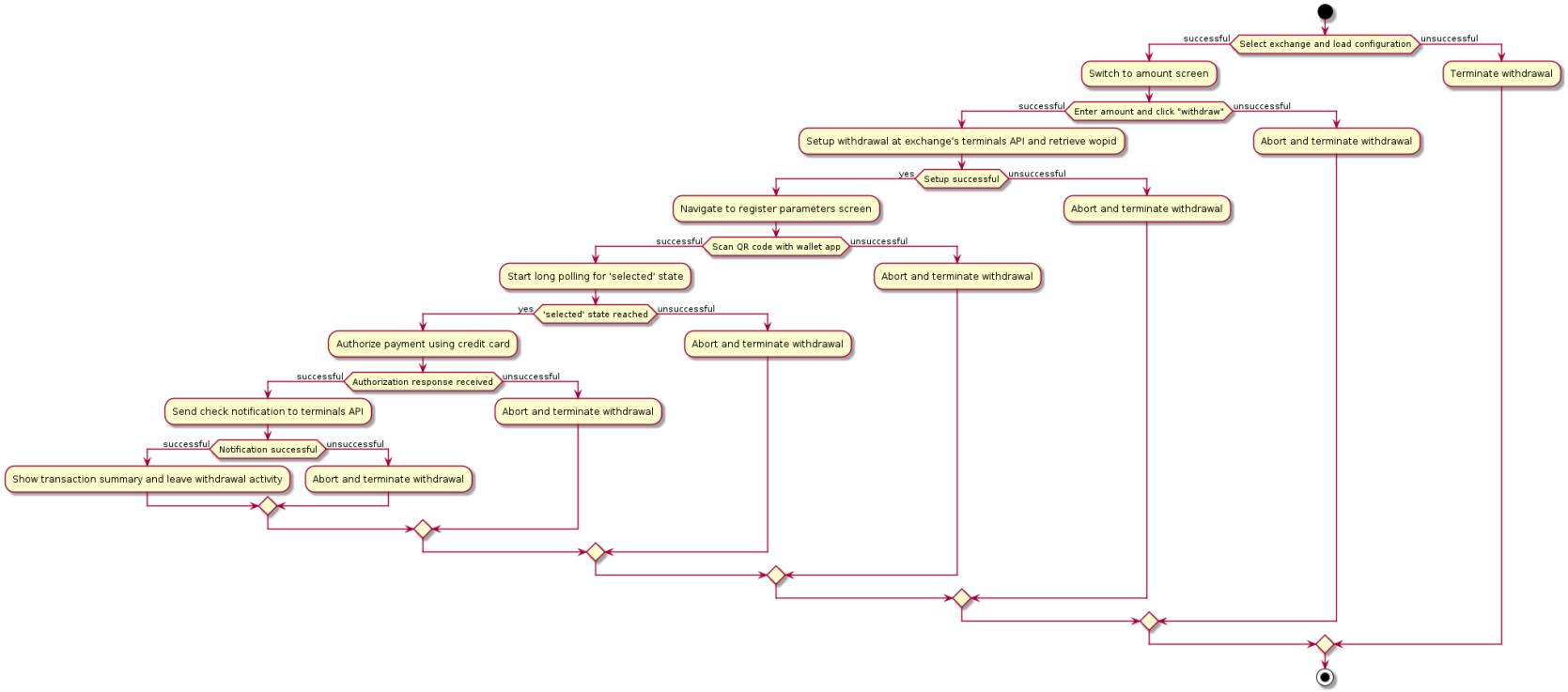

Figure 4.2: The flow of the terminal app

4.2.2 Screens

The application is implemented using Jetpack Compose [22] and each of the screens described in subsection 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. For the summary a standalone activity is used. The state is bound to the activity and compose will make sure to rebuild the UI if values change. It also prevents illegal states and that different withdrawals interfere each other. The state is maintained in a view model as described by Android's documentation [23]. The withdrawal activity handles the lifecycle of the view model instance and initializes the routing of the screens using Android's navigation controller as documented [24]. The navigation integration of Android allows the declarative definition of the in-app routing and is defined at the creation of the withdrawal activity.

Start Screen

When the app is started there will be two options for the user. The first option is to start a withdrawal and the second option is to go to the manage screen.

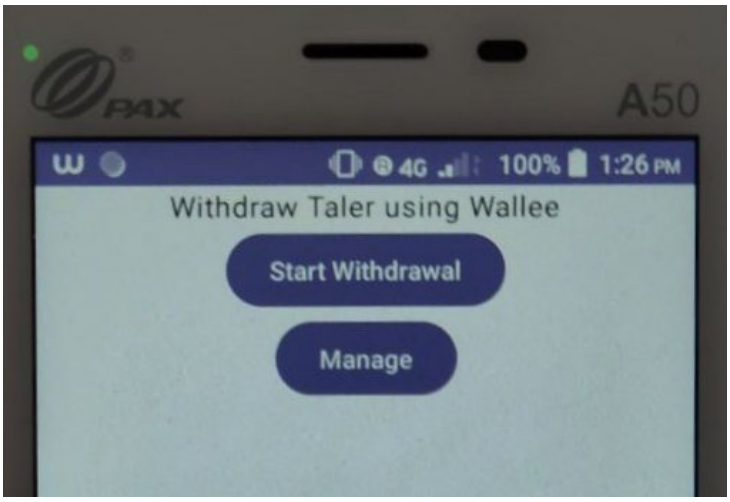

Figure 4.3: Terminal: Start withdrawal or go to manage section

Manage Screen

There are options and special functionalities which can be configured by the terminal operator. These operations are implemented in the manage screen of the app. The app allows to execute the final balance actions which will settle all authorized transactions of this specific terminal. The test transaction was used during the development to learn how the transaction is triggered. In the newest re-

lease it won't be part of the manage screen. The operator might want to test the implementation without having a wallet at hand. This can be done by enabling the wallet simulation. It will lead to the creation of a mocked reserve public key which will not be withdrawable by a wallet. Triggering a payment using the wallet simulation will lead to temporary loss of money (it will eventually be bounced and refunded through the Exchange when the reserve is closed). However, the fees charged will be gone. The wallet simulation should therefore only be used in a test environment. The last option allows the activation of instant settlement. This means that the final balance action will be triggered after each payment authorization.

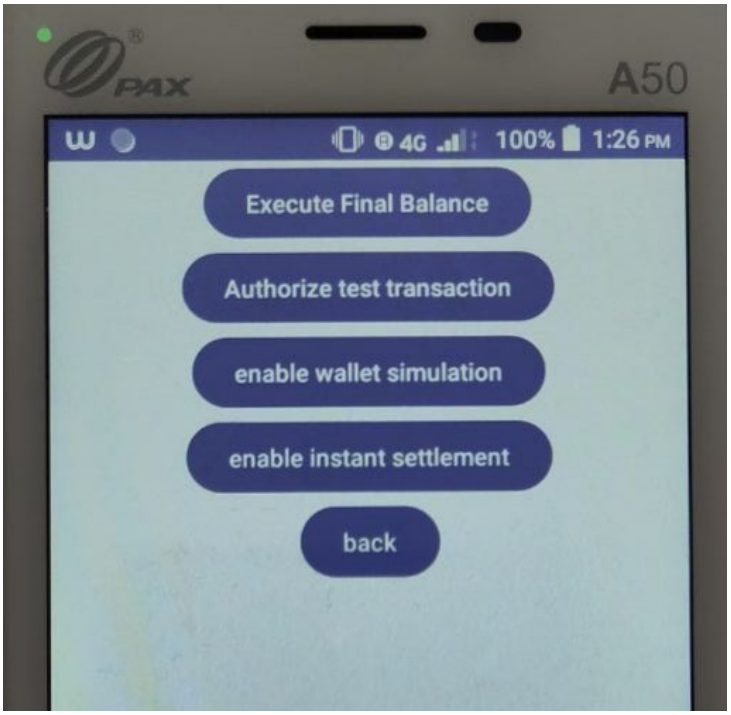

Figure 4.4: Terminal: Manage activities

Choose Exchange Screen

When a new withdrawal is started, the user chooses the exchange to withdraw from (Figure 4.5). 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.

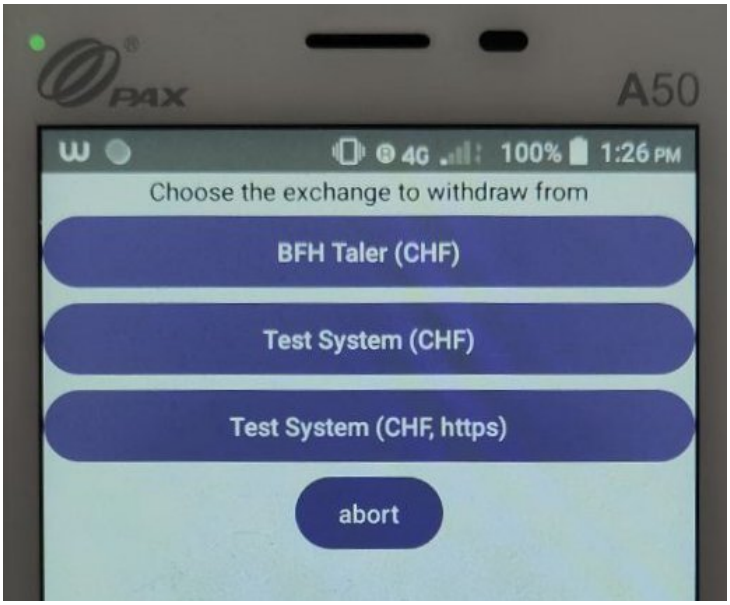

Figure 4.5: Terminal: Select the exchange to withdraw from

Amount Screen

The amount screen in Figure 4.6 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 [25]. Also the fees are shown to the customer on this screen and made transparent in advance.

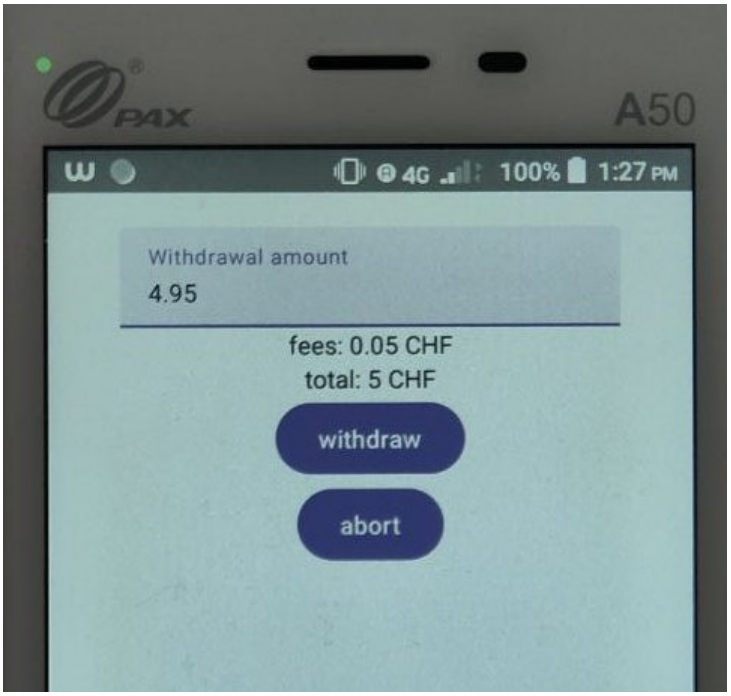

Figure 4.6: Terminal: Enter the desired amount to withdraw

In case the withdrawal amount is invalid, the withdrawal is not possible and an error shown to the customer as depicted in Figure 4.7.

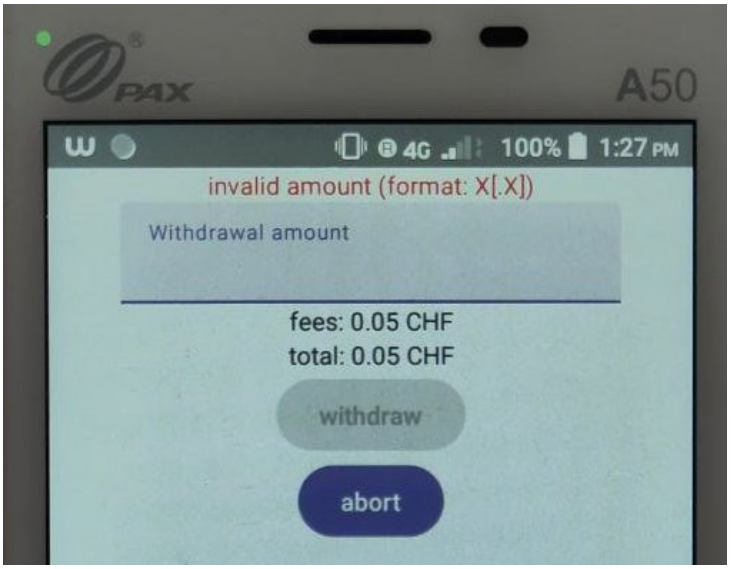

Figure 4.7: Terminal: Fix the amount

Parameter Registration Screen

After entering the amount, a QR code containing the taler withdraw URI is displayed (Figure 4.8). The customers 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. Once this is done the authorization can be started by clicking on the *authorize* button. When clicking on the *abort* button the withdrawal is aborted.

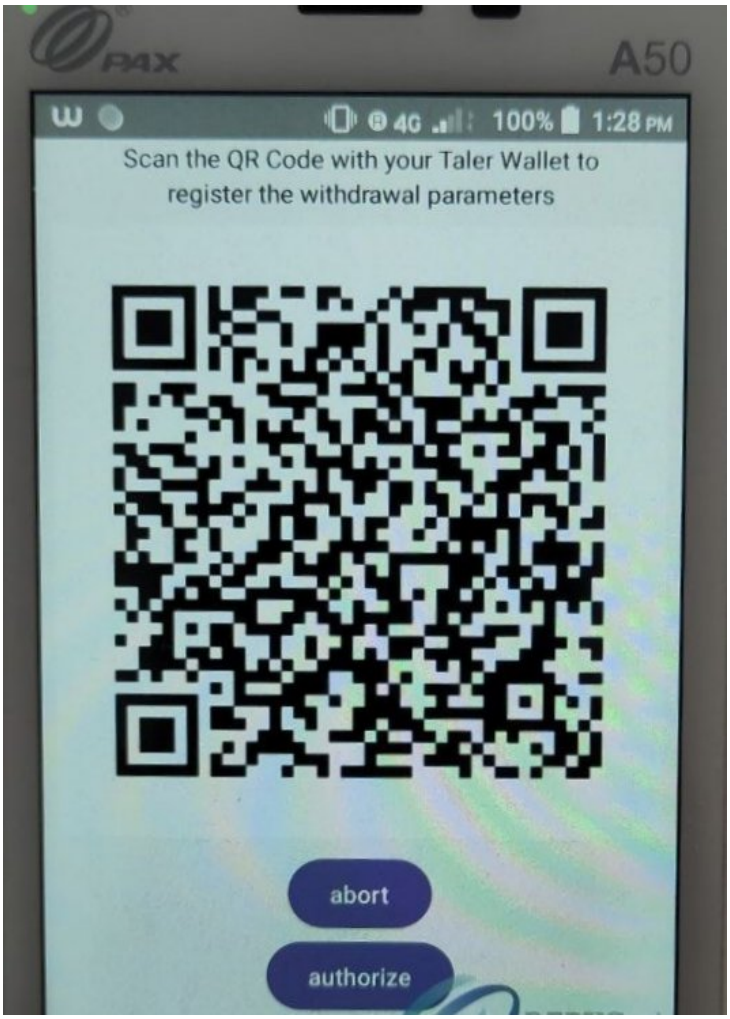

Figure 4.8: Terminal: Register withdrawal parameters

Authorization Screen

The authorization uses the *Android Till SDK* [14] to authorize the amount at the Wallee backend. It covers reading and verifiying the payment mean of the customer. The response handler of the SDK will delegate the response to the implementation of the terminal, which allows triggering the confirmation request using the Terminals API of C2EC. 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.

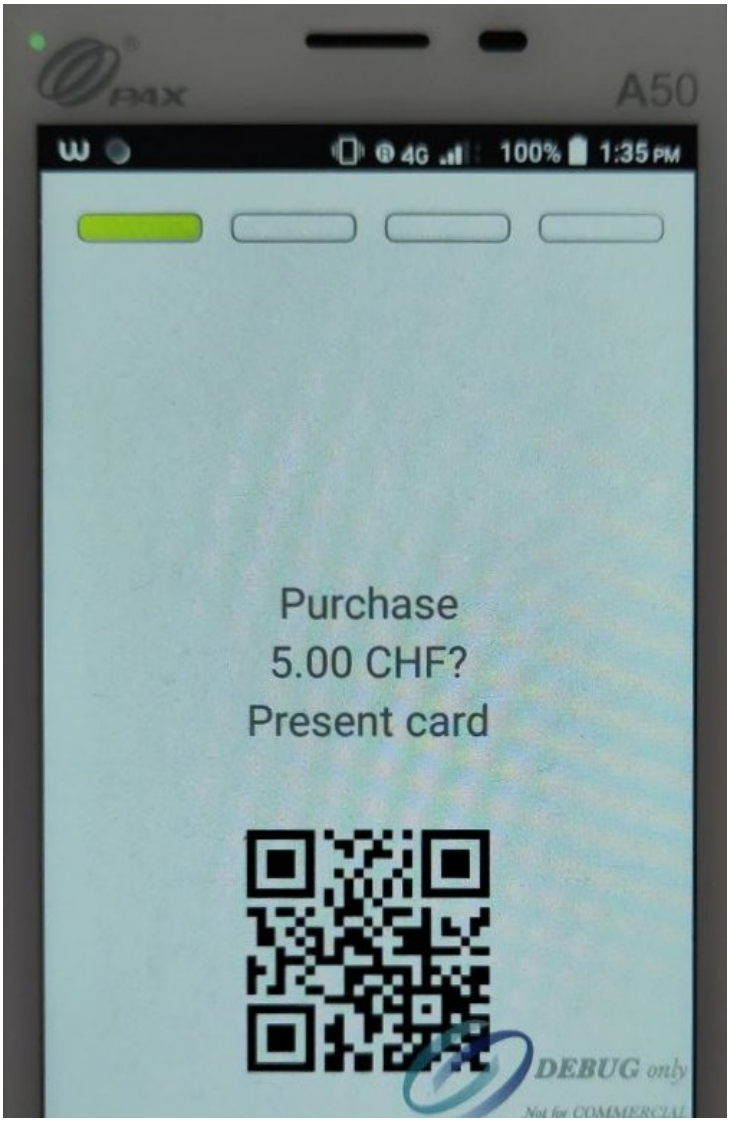

Figure 4.9: Terminal: Authorization using the Android Till SDK

Summary Screen

When the transaction was processed a summary of the transaction is displayed to the customer (Figure 4.10).

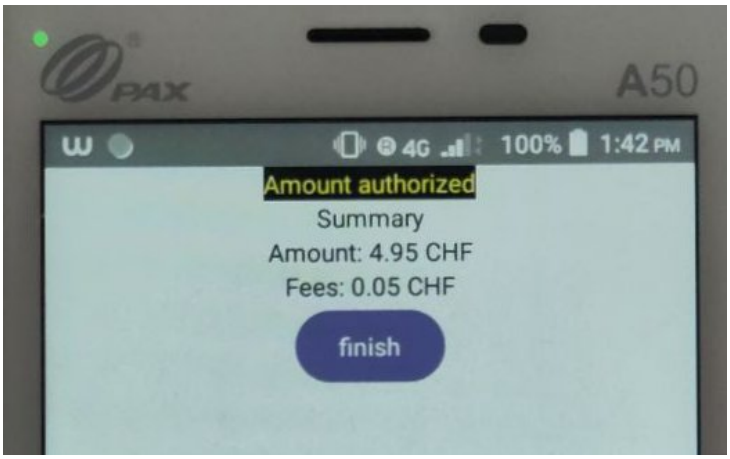

Figure 4.10: Terminal: Payment authorized

Failure Screen

To give the customer a feedback when anything goes wrong or when the withdrawal is aborted, the terminal will always show the rudimentary abort screen (Figure 4.11). This gives the users feedback what happened. What can go wrong and how the terminal acts upon these failures is described in subsection 4.2.3.

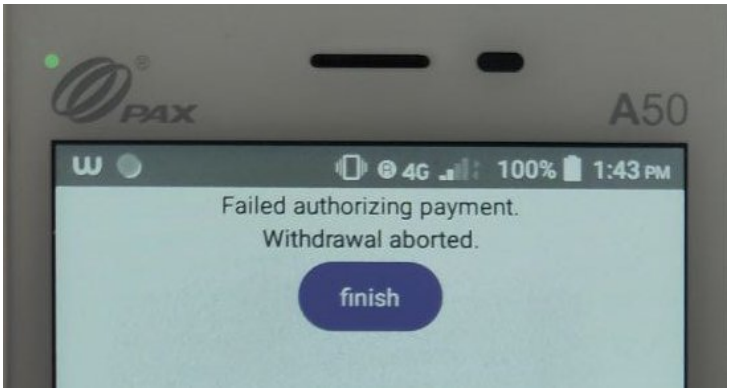

Figure 4.11: Terminal: Payment authorized

4.2.3 Abort Handling

During the flow various steps can fail or lead to the abort of the withdrawal. These edge cases must be considered and handled the right way. Generally we can split the abort handling on the terminal side into two different phases. The implementation of the Wallee POS Terminal follows a strict *abort on failure* strategy. This means that if anything goes wrong the withdrawal is aborted and a new withdrawal might be started. Generally the abort handling strategy is to abort the withdrawal when in doubt and values security (of the money) over the user-experience. The idea behind this strategy is that even when a nice user-experience is implemented, nobody will operate or use a withdrawal process if security of the money cannot be guaranteed.

Abort 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 subsubsection 4.1.2. Every problem which cannot be recovered or not further processed must lead to the abort of the withdrawal.

Abort 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 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 subsubsection 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.

4.2.4 Fulfilling Transactions

To achieve finality and real-time behaviour of the withdrawal flow, the transaction must be forcefully transitioned to the *fulfill* state in the Wallee backend. For

this reason, the payments are not only authorized but also automatically completed in the background. After the successful completion, the payment can be directly settled and with this guarantee the finality property. If the completion fails, the withdrawal will be aborted. Wallee was asked if the instant settlement using the final balance call has cost effects for the merchant. It appears that this is not the case. But Wallee wrote it is not the idea to trigger the final balance every minute. Because of this, the instant settlement is deactivated by default and the terminal operator might decide on their own if they want to activate it or not. Instant settlement can be activated in the manage screen as described in subsubsection 4.2.2.

4.3 Database

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

4.3.1 Schema

For the C2EC component the schema c2ec is created. It holds tables to store the entities described in subsection 3.1.5. 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 confirmation endpoint. Namely the base url and the credentials to authenticate the confirmation 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 hashed.

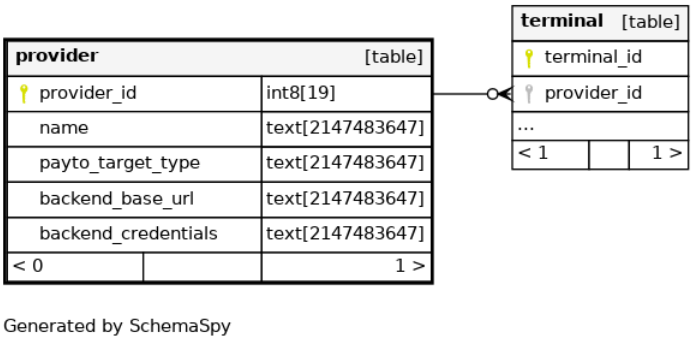

Figure 4.12: Terminal Provider Table

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 hashed 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.

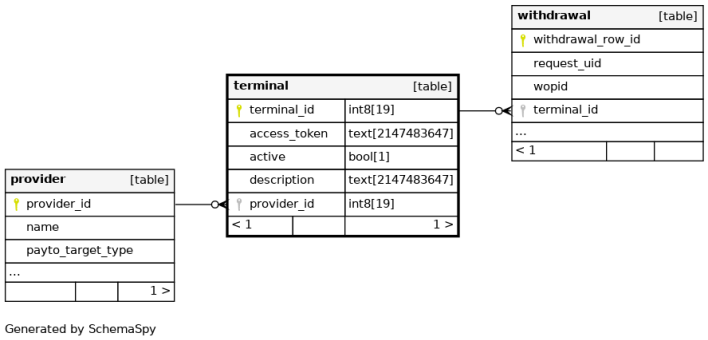

Figure 4.13: Terminal Table

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 subsubsection 4.1.2 or subsubsection 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 subsection 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. The field *confirmed_row_id* is used to separate the already confirmed withdrawals from the pending or aborted withdrawals. It is leveraged by the Wire-Gateway API to only handle already successfully confirmed rows. The existing *withdrawal_row_id* is not suitable for this case since it is not guaranteed that withdrawals are confirmed in the same order as they were added. In a future version of the application it is a good idea to put the confirmed transactions in a separate table which would reduce the complexity of the table and its usage. With this design the application takes care of writing the correct *confirmed_row_id* when a transasction is confirmed.

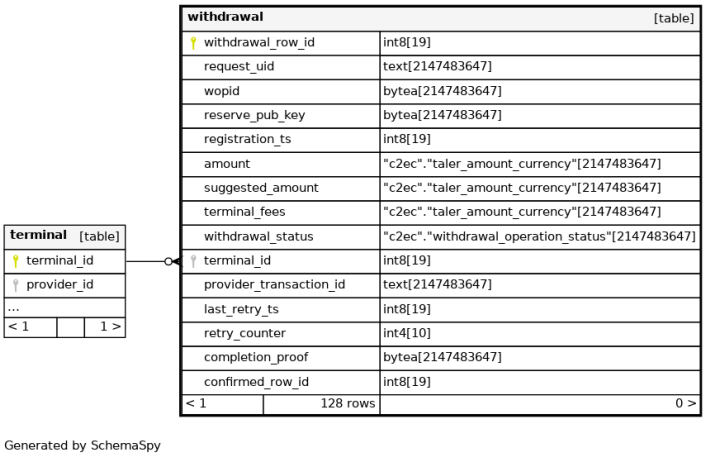

Figure 4.14: Withdrawal Table

Transfers

The transfer table is maintained through the transfer endpoint as described in subsubsection 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. The field *transferred_row_id* is used to separate the transferred transactions from the pending or failed transfers. It is leveraged by the Wire-Gateway API to only handle already transferred rows.

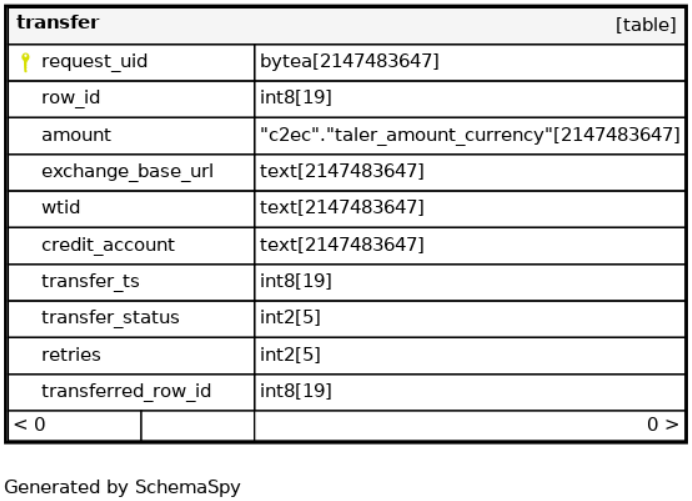

Figure 4.15: Transfer Table

Relationships

The relationships of the tables are created as described in subsubsection 3.1.5. 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.

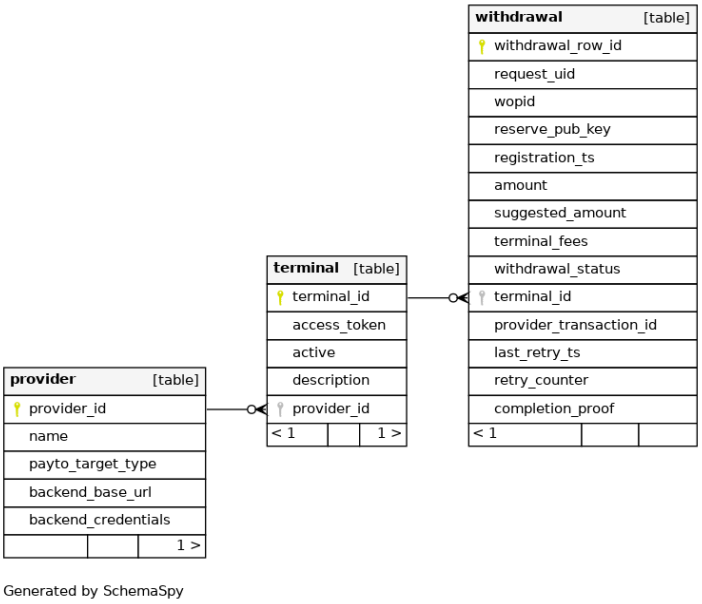

Figure 4.16: Relationships of the entities.

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.

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 reqeuests described in subsubsection 4.1.2 or subsubsection 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 subsubsection 4.1.2). It will start the confirmation of the transaction at the providers backend, through the provider specific confirmation process.

4.3.3 Migrating The Database

To add new SQL to the project a script can be added to the *db* directory of the *c2ec* directory. Make sure to add the path of the new SQL script to the migration script named *migration.sh*. This will execute the migration when using the migration command as documented in subsection 4.7.4. Additionally the new migration shall be registered to the versioning scheme which is installed alongside the c2ec scheme. For examples how this is done scripts within the *db* directory can help.

4.4 Security

4.4.1 General Security Considerations

To review and validate the security of the design two cases were reviewed. The first mirrors the easiest attack (EAV eavesdropping and trying to abuse *WOPID*).

The second case reviews where the most harm can possibly be done to the system.

EAV Abusing WOPID

The *WOPID* is used to link a reserve public key to a withdrawal operation. Since the registration is done through an API, an attacker could try to be first and register its own reserve public key before the customer. When the *WOPID* is somehow precomputable, an attacker could steal the money by registering their own reserve public key before the customer. This threat is mitigated by the request of the wallet resulting in a conflict response code when trying to add a reserve public key to an already registered withdrawal operation. The customer will see this error and not authorize the transaction and instead abort the withdrawal.

Further a *WOPID* can be abused triggering a confirmation or an abort request at the Terminals API or an abort request at the Bank-Integration API. The confirmation or abort from the side of the terminal are mitigated through the authentication of the terminals. When the eavesdropping adversary (EAV) [26] can somehow access the communication between a terminal and C2EC, the *WOPID* cannot be abused without also breaking the terminals credentials. What if the attacker decides to use the unauthenticated Bank-Integration API the wallet would normally use? The specification does not require some proof that the requester is the wallet owning the private key of the reserve. This could lead to tampering of the withdrawals in the time window of the confirmation of the payment. The problem could be mitigated by sending a signed token in the request (the request already is a POST request). The wallet could use its reserve private key to sign the token. The Bank-Integration API could then verify the token using the reserve public key assigned to the withdrawal operation. It is understandable that the risk is accepted, since a potential adversary would need to be sophisticated (needs to redirect requests of the wallet and read *WOPID* from the request). What about wallets run by people in countries which are politically not as stable as Switzerland and censorship is a problem? Maybe it's a good idea to add some mean of authentication to at least the abort endpoint of the Bank-Integration API. On the other hand the attacker needs access to the victims phone anyway and could possibly also use the keys.

Trying To Withdraw Money Without Paying

This case is possible, when an attacker can trick the C2EC to have confirmed withdrawals in its withdrawal table, without having a real confirmation of the payment service provider. This means the attacker can steal money from the exchange. For this an attacker would need to have the possibility to somehow trick the confirmation process of C2EC to issue confirmation requestes against a back-

end controlled by the attacker. This backend would then confirm the withdrawal. This will lead to the creation of the reserve on the side of the Exchange.

Implementation Issues

Another problem could be developers introducing confirmation bugs. The confirmation process of a transaction must be considered as the holy grail from the perspective of the developers. If they do not take biggest care implementing the confirmation process, this could lead to loss of money on the side of the Exchange operator. The program should strictly disallow withdrawals, if the transaction is not guaranteed to be final by the payment system provider. Otherwise the property of the guarantees concerning the finality is harmed and the system no longer secure (in terms of money). When adding new integrations, this section of the code needs great care and review before going to production.

4.4.2 Withdrawal Operation Identifier (WOPID)

The *WOPID* 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 or bully the operators by simply aborting any withdrawal. 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 [27]).

4.4.3 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* [28]. *Argon2* is the winner of the password hashing competition initiated by the cryptographer Jean-Philippe Aumasson [28]. It is a widely adopted best practice approach for hashing passwords. Storing the hash of the credentials makes abusing stolen credentials very hard and therefore prevents the abuse of credentials gathered through a database leak. The CLI

described in section 4.5 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 4.7 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

4.4.4 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 *application-user-key* (which is just an access-token the user receives when creating a new API user in the management backend of Wallee) as key and the above mentioned properties plus information about the requested HTTP method and the exactly requested path (including request parameters) as message [29]. 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 [30]. There are implementations of the mechanism in Java and other languages available. For Go it was implemented during the thesis.

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).

4.4.5 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 [10] an specified in the terminals API specification [9]. Therefore a generated access-token used as password and a username which is generated registering the terminal using the cli explained in subsection 4.4.6 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 [31] as described in RFC-7617 [10]. Therefore the C2EC component allows the configuration of a user-

name and password for the exchange. During the request of the exchange at the wire gateway API, the credentials are checked.

4.4.6 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 4.5). 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.

4.4.7 Hijacking And Stealing 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 described in section 4.5.

4.5 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 or by starting the cli with the *-c* option with the path to the *.ini* config file containing the connection string. 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 a simple 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.

4.5.1 Adding Wallee Provider

Adding the Wallee provider can be achieved calling *rp* (register-provider). It will then ask for properties like the base url and the credentials of the API user (generated by Wallee). Since the payto target type in case of Wallee will always be *wallee-transaction*, this is hard coded. The credentials supplied are hashed using argon2 [32]. If the database leaks for some reason, the passwords cannot be abused easily.

4.5.2 Adding Wallee 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 '-')

4.5.3 Deactivating Terminals

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. To detect suspicious activity in production appropriate montioring tools could be installed to automatically trigger alarms.

4.5.4 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.

4.6 Testing

Since the program leverages concurrency and parallizes work a simulation client and simulation program was implemented. The simulation 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 proof the functionality of the core. 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. The simulation can be configured with the specified configuration format. An example configuration can be found in the simulation source directory of the C2EC repository.

Besides the automated tests, using the above mentioned simulation, unit tests were implemented for parsing, formatting and encoding functions. Additionally manual test were fulfilled to ensure the system behaves correctly. To test the wire-gateway API, the *taler-exchange-wire-gateway-client* [33] facility was used supplied by GNU Taler to verify the correct functioning of the API.

In the end to approve the process, manual tests were executed. During this phase a few bugs were discovered which were not known before. After resolving them the system was ready to issue digital cash to the customer. During this phase tests were made with various means of payment: credit card, debit card, apple wallet (credit card). Also the withdrawal was tested using the IOS and Android version of the Taler wallet. Both platforms are working as expected.

4.6.1 Wallee Test System

The testsystem of Wallee has some behavioral specialities. The system will process payments based on the amount. After a short conversion with Wallee it was learned that following amounts will lead to approved payments:

3.00 - Approved

4.00 - Approved

5.00 - Approved

6.00 - Approved

7.00 - Approved

8.00 - Approved

9.00 - Approved

It appears that also other amounts will be approved but they were not listed by Wallee. The amounts in the list above are guaranteed to be approved.

4.7 Deployment

4.7.1 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, 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.
7. Set the postgres variables `PGHOST` and `PGPORT` to the correct values if needed

For the deployment of the Wallee payment 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.

4.7.2 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 subsubsection 4.4.3) 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.

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 4.5) and the *rp* command. There the space-id, the user-id of the application user and the application-user-key must be provided. The cli will register the provider using these values.

Registering Wallee Terminal

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 Terminal

To setup the Wallee terminal, the Android app must be configured and built with the credentials gained by the terminal registration process described in subsubsection 4.7.2.

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.

4.7.3 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 -c [PATH-TO-CONFIGFILE]`.

The terminal app must be deployed by the Wallee support. The Android package (APK) will be installed over the air by them once the APK was accepted and signed by them. To get a signed APK, it must be sent to *info@wallee.com*. They will first check and sign the APK. After this step another message must be sent to them with a link to the signed APK. With this request the information of the terminal to install the application on must be given. Wallee will then rollout the app on the specified device.

Making C2EC Accessible Via Internet

To make the C2EC instance available a web-server must be configured to receive requests and hand them to the C2EC instance. The exact configuration will not be covered within this thesis. The test installation uses a NGINX reverse proxy [34] to allow the access over the internet. A rudimentary configuration is enough to allow the access. It helps to set big timeouts since a lot of long-polling is done. To not undermine this, NGINX should not terminate the connection before the long-poll time exceeds. Setting big values for timeouts will be a good practice for C2EC. On the other hand clients should not run long running requests (more than a minute or two) against C2EC and instead leverage retries to extend the time they wait for a response. A value between 30 and 60 seconds might be a good choice for long-polling requests (These values are also used by the wallet and the wire watch process of the Exchange). Using too long values for long-pollings can result in less robust systems due to timeout problems.

4.7.4 Migration And Releases

When a new version of the system shall be installed, the new executable can be built by issueing `make build` from the sources root directory. After migrating the database using `make migrate` the newly built executable can be started. For new versions of the cli and the installation `make cli` and `make simulation` can be used.

5 Results

5.1 Discussion

This thesis shows that withdrawals in GNU Taler are possible using the payment service provider Wallee. The implementation displays how the objectives of finality, user-experience and security can be achieved. The C2EC implementation also achieves the integration into the rest of the Taler ecosystem and gives a reference on how this can be achieved.

The design of the Terminals API was a major field of work during the process. Only after several iterations, the specification was ready. The iterations were necessary to sharpen the understanding of how the terminal and C2EC must interact and integrate with the existing Taler components in order to make the withdrawals functional. At first the existing Bank-Integration API was copied and extended before merging the copy with the existing Bank-Integration API. After this step I extracted terminal specific endpoints to the new Terminals API. Like this the separation of terminal and wallet specific functionality could have been achieved. The current implementation keeps changes to the existing Bank-Integration API low and therefore allow the integration of the wallet without further changes.

The implementation of the existing Bank-Integration and Wire-Gateway API were a challenge because they must be implemented with great care to not violate the specification. Another challenging task was the design of C2EC. Making C2EC a useful, robust and extensible, required the understaning of details of Taler such as byte encodings or amount handling. This task was a little more time consuming than initially planned. At first, I assumed that C2EC would just be implemented and work. This was a bit optimistic. In reality the process was iterative. Only after a lot of iterations a suitable way for the implementation was found.

A challenge which was encountered during the implementation of the terminal application and the C2EC component, was the concurrency of processes. To make the withdrawal flow as easy and useful as possible, a lot of tasks need to be covered in the background and run besides each other. This added the technical requirement to decouple steps and leverage retries to increase the robustness of the process. It helped a lot to understand that the state of a withdrawal was the anchor these retry mechanisms must be built around.

Fees are a central aspect of the process and decide wether the implementation

can be used or not. The different fee models of subsection 4.1.5 describe how fees can add up during a withdrawal. The current implementation does not cover all fee models, because fee models two to four also depend on the payment system provider used for a specific withdrawal. The checks that C2EC can make to secure its own fees are implemented. Fee model one seems to be the most secure and easy to implement fee model. It can be covered by the core implementation of C2EC and does not rely on the payment system provider specific implementation. Using other fee models requires great care during the integration and adds complexity. For the thesis, model one was implemented because Wallee uses this model. These models also resolve the discussion of the midterm meeting with Dr. Alain Hiltgen.

Towards the end of the implementation it became obvious that a simple authorization of the payment was not enough to imitate the real time feeling of the withdrawal. Other requests were necessary to do so. To findout which requests needed to be filed against the Wallee backend some investigation had to be made. The documentation does explain which states exists in Wallee's transaction scheme but does not explain, which operation must be triggered to transition states. This made the investigation somewhat cumbersome. Also the integration of the backend needed more investigation than assumed. This also led to the

The new cashless approach to withdraw digital cash makes a faster uptake of GNU Taler possible. Potential customers will only need a supported payment mean to withdraw digital cash. They can now use C2EC and the terminal app for Wallee to withdraw digital cash using GNU Taler.

5.2 Limitations And Future Work

Due to the short time available during the thesis, features and integrations are missing which can make C2EC even more valuable. Because of this I provide a list of future work. Maybe there are other students or collaborators who want to join in and add their features to the existing code. The list might not be complete and any new ideas are appreciated. I splitted the list into the list of extensions and improvements. The improvements list also shows some limitations of the implementation done during the thesis.

5.2.1 Extensions

1. Integration of other providers: To make use of the implemented structures, it would be nice to add more payment service providers.
2. Management interface for terminals: To allow easier use of the system it might be nice to have a more sophisticated interface to manage terminals.

The implemented cli helps to get an understanding, how such a management interface can add terminals to the system.

3. Automated registration: With increasing use of C2EC, it might be nice for the operator to allow an automatic registration of new terminals.
4. Support quotas: To fully support the Terminals API, the quotas endpoints can be implemented.
5. Proper packing: To fully integrate into the Taler landscape, C2EC should be packed and installed like other Taler components such as the Exchange.

5.2.2 Improvements

1. Paydroid app: Run a Wallee terminal on behalf of the BFH.
2. Paydroid app: The app must be released including the credentials. This is a security risk since these credentials are shipped through (secure?) channels. A way to register to an exchange in the app is a nice extension.
3. C2EC: Remove doubled provider structures. Currently providers are saved to the database and must be configured in the configuration. To make the setup and management easier, the providers can only be configured inside the configuration.
4. C2EC: Proper separation of confirmed and unconfirmed withdrawals and transferred and untransferred refunds to reduce complexity of the implementation.
5. C2EC: Only one provider per instance is allowed to use the same payto target-type. Currently an additional instance must be configured, if two or more payment service providers are using the same payto target type.
6. Implement more fee models: To allow easier integration of other providers, the described fee models can be centralized in one location. This would help to improve the quality and robustness of the system.
7. Database locking: Currently no database locking is used. This can lead to race conditions. To completely prevent this locks can be applied.
8. IPv6 support: The process must also listen on IPv6 addresses.
9. Support cli without interaction. The cli currently is purely interactive. It would be a nice improvement if the cli would be usable without interactio. This would allow to use the cli for automated tasks.
10. Partial refunds: The current implementation only allows refunds of the entire withdrawal amount. In the future the implementation can support partial refunds.

5.3 Conclusion

5.3.1 Technically

Generally, I think the implementation does its job very well. I was able to implement the required processes and the targeted user-experience. The implementation (in C2EC as well as in the Paydroid app) suffers of some technical debts (listed in subsection 5.2.2). I finally accepted them, because I had to prioritize the formal parts of the thesis, such as the poster, video, book entry or this documentation. I could have prevented some of these issues, if I would have read the documentation and specification more carefully. But overall I am satisfied with the work I did concerning the short time span that was available for the implementation.

C2EC

The implementation of C2EC was the biggest part of my work during the thesis. It is the core of the framework and I had to think about API specification conformance, extensibility, correctness and database integration. I wanted to achieve an architecture, which is similar to the one of the GNU Taler Exchange. This inlcuded to decouple steps in the withdrawal process using triggers and the notify feature of postgres. I learned a lot about how postgres works and how I can write working postgres functions and triggers. First, I failed to properly document the database fields, tables and functions. After a review with Prof. Dr. Christian Grothoff, I learned about the comment functionality of the postgres SQL standard.

In Go code of the C2EC component I had to implement a robust way to communicate between parallel running processes. The Go concurrency model made this possible in an straight forward and what I think, comprehensible way. I correctly anticipated that it would pay out to first implement the concepts for my requirements in a dummy project and then adapt them to C2EC. Additionally, the implementation of the database access layer behind an interface allows to change the database without changing the entire application.

Concerning the extensibility of C2EC I was able to implement code level abstractions which allow an seemless integration of additional payment system providers. After the feedback of Prof. Dr. Christian Grothoff I was able to eliminate an unnecessary layer of abstraction, which made it even easier.

I think I have been able to apply a lot of the knowledge I have gained over the last three years. From time to time I thought: "ah, this is why the professor told us this". This helped me to deepen my understanding of topics like encoding, REST API, concurrency and many more.

Paydroid Terminal Application

The paydroid application was challenging to me since I never wrote a real Android application on my own. That is why I think I did a good job by implementing a best practice structure with the view models, composable and navigation controller. Thanks to the feedback of Prof. Dr. Benjamin Fehrensen, I was able to improve the design and verify the correctness of these best practices.

I first had problems to understand how exactly the versioning in Android works. The backward compatibility is given even when big time gaps between the feature needed and the version in use occur. In the beginning I suffered to understand the difference of the none compose and compose era of Android programming and mixed the patterns. After better understanding how features work in jetpack compose I think I implemented a modern Android app.

Since the app needs to do requests in the background I had to understand how this could be achieved. Therefore, I needed to understand how I can access other threads. I think in this area is the biggest shortcoming of my implementation. I implemented an asynchronous state-handling. Threads running detached from the UI-Thread will update the model, which will lead to the regeneration of the composables. I think it would be a better way to implement a the asynchronous tasks using Androids state flow features. Due to the lack of time I decided to not do this anymore.

It was interesting to learn about the difference of Go goroutines and Kotlin coroutines. While running background tasks using goroutines works perfectly fine, in Kotlin on Android I learnt it is required to start a new thread and launch coroutines on the new thread. Otherwise Android will not allow network requests, because it disallows I/O operations on its UI thread. From my point of view this shows a limitation of coroutines on top of JVM threads. They are not real parallel but just suspend work on the thread and check periodically if they can process further. If there are thread level restrictions (like the Android restrictions), they will affect the execution of the coroutines on top and like this undermine the concept of coroutines.

5.3.2 Methodically

To organize the work I did a rough planning of the work and the artefacts in the beginning. On top of this plan I did a weekly iterative planning of the work I wanted to do. This plan was presented through the weekly meeting with Prof. Dr. Christian Grothoff and Prof. Dr. Benjamin Fehrensen. Sometimes the plan needed to be sligthly adjusted due to their feedback. This led to the organization of doing my planning at Thursday night so I could plan my work and adjust the plan after our weekly meeting at Wednesday morning. I think I could have made the process a bit more transparent but in the end I was able to deliver the

artefacts and deliverables on time. Sometimes I lost focus because there were so much loose ends to keep up with. I then did something different and ordered my thoughts. This helped sometimes but not always. When the stress level was rising this was even more difficult. In such situations taking a step back and prioritzing the work helped me.

5.3.3 Personally

The world of payment systems seems a bit chaotic to me. I think this is the result of a lot of different approaches for the same problem developed at the same time. Standards exist but they mainly suggest things and do not enforce them. The technical documentation is obfuscated in big documents with a lot of boiler plate text. This makes it very hard to act appropriately without finding out by hand how a process works exactly. For example to bring a Wallee transaction into the fulfill state (which allows the shipping of goods) you must settle the transaction and exeeute the final balance. The documentation does not care about this. I had to write e-mails with Wallee to finally understand this. Even the people of Wallee messed up their own transaction states.

The thesis was constrained with a lot of insecurities for me. How does the process look? How can I implement the process? How does GNU Taler even work? How does Wallee work? In the end I am proud of what I accomplished during the thesis. I was able to understand the different API and write a program which fulfills the properties needed for the withdrawal. Additionally I could learn a lot about designing an API and especially parallelization in Go and Android.

I am thankful that the Bern University of Applied Sciences supports free software projects like GNU Taler. It was a great opportunity for me as student and as human to gain direct insights and work on a GNU project during my thesis. I remember Prof. Dr. Christian Grothoff telling me during an onsite session: "Nicht so kurzfristig denken!" (do not think short-term). This also showed the horizon of the project to me. It tries to sustainably change the payment landscape for good. That is what I like the most about free software. It is built to last. The world will not get better when we keep pushing towards short-term profit benefiting individuals, global warming and war. GNU Taler and other GNU projects are making a difference and take a humanitarian perspective on technology. Providing technology supporting *humans*. This was the reason I started my journey in computer science with my apprenticeship in 2015 and eventually decided to do my thesis on GNU Taler. It has not changed since. Even when my contribution is small I believe it is important. When everyone adds their ideas and work to the plate, we can achieve a better world. The title picture, generously provided by cartoonist Bruno Fauser [35], visualizes this attitude. Sometimes it is hard to not loose faith for the good, but; the good wins. Always.

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.

June 11, 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. Accessed: 2024-06-10.

[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. Accessed: 2024-06-10.

[3] GLS Bank. Taler - die zukunft des digitalen, sicheren und nachhaltigen bezahlens. https://www.gls.de/privatkunden/taler. Accessed: 2024-06-10.

[4] NGI TALER. Ngi taler. https://taler.net/en/ngi-taler.html. Accessed: 2024-06-10.

[5] Wallee. Payment connectors. https://app-wallee.com/connectors. Accessed: 2024-06-10.

[6] Taler. Withdrawal. https://docs.taler.net/taler-wallet.html#withdrawal. Accessed: 2024-06-10.

[7] Taler. Taler wire gateway http api. https://docs.taler.net/core/api-bank-wire.html. Accessed: 2024-06-10.

[8] Taler. Taler bank integration api. https://docs.taler.net/core/api-bank-integration.html. Accessed: 2024-06-10.

[9] Taler. Terminal api. https://docs.taler.net/core/api-terminal.html. Accessed: 2024-06-10.

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

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

[12] PCI Security Standards Council. Pci data security standard. https://docs-prv.pcisecuritystandards.org/PCI%20DSS/Standard/PCI-DSS-v4_0.pdf. Accessed: 2024-06-10.

[13] Wallee. Transaction states. https://app-wallee.com/de-de/doc/payment. Accessed: 2024-06-10.

[14] Wallee. Android till sdk. https://github.com/wallee-payment/android-till-sdk. Accessed: 2024-06-10.

[15] Wallee. Transaction service. https://app-wallee.com/de-de/doc/api/web-service#transaction-service. Accessed: 2024-06-10.

[16] Wallee. Refund service. https://app-wallee.com/de-de/doc/api/web-service#refund-service. Accessed: 2024-06-10.

[17] Wallee. Transaction states. https://app-wallee.com/de-de/doc/payment/transaction-process. Accessed: 2024-06-10.

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

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

[20] GNUnet Project. The gnunet assigned numbers authority (gana). https://gana.gnunet.org/. Accessed: 2024-06-10.

[21] Douglas Crockford. Base 32. https://www.crockford.com/base32.html. Accessed: 2024-06-10.

[22] Developer-Android. Build better apps faster with jetpack compose. https://developer.android.com/develop/ui/compose. Accessed: 2024-06-10.

[23] Developer-Android. Viewmodel overview. https://developer.android.com/topic/libraries/architecture/viewmodel. Accessed: 2024-06-10.

[24] Developer-Android. Navigation. https://developer.android.com/guide/navigation. Accessed: 2024-06-10.

[25] 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.

[26] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2020.

[27] Golang Doc. rand. https://pkg.go.dev/crypto/rand. Accessed: 2024-06-10.

[28] Jean-Philippe Aumasson. Password hashing competition. https://www.password-hashing.net. Accessed: 2024-06-10.

[29] Wallee. Authentication. https://app-wallee.com/en-us/doc/api/web-service#_authentication. Accessed: 2024-06-10.

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

[31] Taler. Taler wire gateway http api. https://docs.taler.net/core/api-bank-wire.html#authentication. Accessed: 2024-06-10.

[32] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications. RFC 9106, September 2021.

[33] Taler. taler-exchange-wire-gateway-client. https://docs.taler.net/manpages/taler-exchange-wire-gateway-client.1.html. Accessed: 2024-06-10.

[34] NGINX. Nginx reverse proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/. Accessed: 2024-06-10.

[35] Bruno Fauser. *Träumen von einer besseren Welt*. Cartoonist Bruno Fauser, Hinterkappelen, fauser.ch, 2023.

List of Figures

List of Tables

Appendix A

1.11.7. Terminal API

Table of Contents

1.11.7.1. Introduction

Terminals are devices where users can withdraw digital cash.

This API is offered by a payment service backend and is used by such terminals. It enables imposing limits on withdrawals per unique user ID (and communicating such limits to the terminals) as well as setting up and triggering withdrawal operations.

Implementations of this API typically interact with a terminal-specific payment service (or a bank) to realize the service.

1.11.7.2. Authentication

Terminals must authenticate against all terminal API using basic auth according to HTTP basic auth.

1.11.7.3. Config

`GET /config`

Return the protocol version and configuration information about the bank. This specification corresponds to `current` protocol being version **0**.

Response:

200 OK:

Response is a TerminalConfig.

Details:

```
interface TerminalConfig {
  // Name of the API.
  name: "taler-terminal";

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

  // Terminal provider display name to be used in user interfaces.
  provider_name: string;

  // The currency supported by this Terminal-API
  // must be the same as the currency specified
  // in the currency field of the wire gateway config
  currency: string;

  // The withdrawal fees which of this Terminals API endpoint.
  // If the Exchange chooses to charge no fees, then just configure
  // the zero amount.
  withdrawal_fees: Amount;

  // Wire transfer type supported by the terminal.
  // FIXME: needed?
  wire_type: string;
}
```

GET /quotas/$UUID

Obtain the current transaction limit for the given $UUID. The UUID should be an encoding of a unique identifier of the user.

Response:

200 OK:

Response is a WithdrawLimit.

Details:

```
interface WithdrawLimit {
  // Maximum amount that can be withdrawn now.
  limit: Amount;

  // Time when the limit may increase.
  expiration: Timestamp;
}
```

POST /quotas/$UUID/lock

This endpoint allows a terminal to reserve a given amount from the user's quota, ensuring that a subsequent operation will not fail due to a quota violation.

Request:

The request should be a WithdrawLimitLock.

Response:

204 No content:

The change was accepted.

409 Conflict:

The proposed lock would push the user above the limit.

Details:

```
interface WithdrawLimitLock {

  // Amount that should be reserved from the quota.
  limit: Amount;

  // ID for the lock.  FIXME: could also be 32-byte nonce?
  lock: string;

  // How long should the lock be held?
  expiration: Timestamp;
}
```

DELETE /quotas/$UUID/lock/$LOCK

This endpoint allows the terminal to clear a lock it may have previously created.

Response:

204 No content:

The lock was cleared.

404 Not found:

The lock is unknown.

409 Conflict:

The lock was already used in a withdrawal operation.

POST /withdrawals

This endpoint allows the terminal to set up a new withdrawal operation.

Request:

The request should be a TerminalWithdrawalSetup.

Response:

200 Ok:

The operation was created. The response will be a TerminalWithdrawalSetupResponse.

404 Not found:

A lock was specified but the lock is not known for the given user.

409 Conflict:

A conflicting withdrawal operation already exists or the amount would violate the quota for the specified user.

Details:

```
interface TerminalWithdrawalSetup {

  // Amount to withdraw.  If given, the wallet
  // cannot change the amount!
  amount?: Amount;

  // Suggested amount to withdraw. If given, the wallet can
  // still change the suggestion.
  suggested_amount?: Amount;

  // A provider-specific transaction identifier.
  // This identifier may be used to attest the
  // payment at the provider's backend. Optional,
  // as we may not know it at this time.
  provider_transaction_id?: string;

  // The non-Taler fees the customer will have
  // to pay to the service provider
  // they are using to make this withdrawal.
  // If the fees cannot be precalculated,
  // they can be specified in the /withdrawals/$WITHDRAWAL_ID/check
  // request after the transaction was executed.
  terminal_fees?: Amount;

  // Unique request ID to make retried requests idempotent.
  request_uid: string;

  // Unique user ID of the user.  Optional
  // in case a user Id is not (yet) known.
  user_uuid?: string;

  // ID identifying a lock on the quota that the client
  // may wish to use in this operation.  May only be
  // present if user_uuid is also given.
  lock?: string;
}
```

```
interface TerminalWithdrawalSetupResponse {

  // ID identifying the withdrawal operation being created.
  withdrawal_id: string;
}
```

POST /withdrawals/$WITHDRAWAL_ID/check

Endpoint for providers to notify the terminal backend about a payment having happened. This will cause the bank to validate the transaction and allow the withdrawal to proceed. The API is idempotent, meaning sending a payment notification for the same `WITHDRAWAL_ID` return successfuly but not change anything. This endpoint is always *optional*: the bank, exchange and wallet should all eventually notice the wire transfer with or without this endpoint being called. However, by calling this endpoint checks that might otherwise only happen periodically can be triggered immediately.

The endpoint may also be used to associate a user ID at a very late stage with the withdrawal — and still get an immediate failure if we are above the quota.

> **Note**
>
> The backend shall **never** just accept this claim that the payment was confirmed, but instead needs to internally attest that the payment was successful using provider-specific transaction validation logic! The point of this endpoint is merely to trigger this validation **now**.

Request:

The body of the request must be a TerminalWithdrawalConfirmationRequest.

Response:

204 No content:

The payment notification was processed successfully.

404 Not found:

The withdrawal operation was not found.

409 Conflict:

The withdrawal operation has been previously aborted and cannot be confirmed anymore.

451 Unavailable for Legal Reasons:

The withdrawal operation cannot be confirmed because it would put the user above the legal limit. The payment service will eventually bounce the transfer (if it were to become effective), but the user should already be shown an error.

Details:

```
interface TerminalWithdrawalConfirmationRequest {

  // A provider-specific transaction identifier.
  // This identifier may be used to facilitate the
  // backend to check that the credit was confirmed.
  provider_transaction_id?: string;

  // The fees which the customer had to pay to the
  // provider
  terminal_fees?: Amount;

  // A user-specific identifier for quota checks.
  user_uuid?: string;

  // ID identifying a lock on the quota that the client
  // may wish to use in this operation.  May only be
  // present if user_uuid is also given.
  lock?: string;
}
```

`GET /withdrawals/$WITHDRAWAL_ID`

Query information about a withdrawal, identified by the `WITHDRAWAL_ID`.

Request:

Query Parameters:

- **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.
- **old_state** – *Optional.* Default to “pending”.

Response:

200 OK:

The withdrawal operation is known to the bank, and details are given in the BankWithdrawalOperationStatus response body.

404 Not found:

The operation was not found.

`DELETE /withdrawals/$WITHDRAWAL_ID/abort`

Aborts `WITHDRAWAL_ID` operation. Has no effect on an already aborted operation. This endpoint can be used by the terminal if the terminal aborts the transaction, ensuring that the operation is also aborted at the bank.

Request:

The request body is empty.

Response:

204 No content:

The withdrawal operation has been aborted.

404 Not found:

The withdrawal operation was not found.

409 Conflict:

The withdrawal operation has been confirmed previously and can’t be aborted.

1.11.7.4. Endpoints for Integrated Sub-APIs

`ANY /taler-integration/*`

All endpoints under this prefix are specified by the. GNU Taler bank integration API.
This API handles the communication with Taler wallets.

`ANY /taler-wire-gateway/*`

All endpoints under this prefix are specified by the GNU Taler wire gateway API.

The endpoints are only available for accounts configured with
`is_taler_exchange=true`.

Previous
‹ 1.11.6. Taler Conversion Info API

Next
1.12. The Donau RESTful API ›

© Copyright 2014-2024 Taler Systems SA (GPLv3+ or GFDL 1.3+).

1.11.5. Taler Bank Integration API

This chapter describe the APIs that banks need to offer towards Taler wallets to tightly integrate with GNU Taler.

Table of Contents

`GET /config`

Return the protocol version and configuration information about the bank. This specification corresponds to `current` protocol being **v2**.

Response:

200 OK:

The exchange responds with a IntegrationConfig object. This request should virtually always be successful.

Details:

```
interface IntegrationConfig {
  // Name of the API.
  name: "taler-bank-integration";

  // libtool-style representation of the Bank protocol version, see
  //
https://www.gnu.org/software/libtool/manual/html_node/Versioning.html#Versionin
g

  // The format is "current:revision:age".
  version: string;

  // URN of the implementation (needed to interpret 'revision' in version).
  // @since v2, may become mandatory in the future.
  implementation?: string;

  // Currency used by this bank.
  currency: string;

  // How the bank SPA should render this currency.
  currency_specification: CurrencySpecification;
}
```

1.11.5.1. Withdrawing

Withdrawals with a Taler-integrated bank are based on withdrawal operations. Some user interaction (on the bank's websitei) creates a withdrawal operation record in the bank's database. The wallet can use a unique identifier for the withdrawal operation (the `WITHDRAWAL_ID`) to interact with the withdrawal operation.

`GET /withdrawal-operation/$WITHDRAWAL_ID`

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

Request:

Query Parameters:

- **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.
- **old_state** – *Optional.* Default to “pending”.

Response:

200 OK:

The withdrawal operation is known to the bank, and details are given in the BankWithdrawalOperationStatus response body.

404 Not found:

The operation was not found.

Details:

```
interface BankWithdrawalOperationStatus {
  // 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 **v1**
  status: "pending" | "selected" | "aborted" | "confirmed";

  // Currency used for the withdrawal.
  // MUST be present when amount is absent.
  // @since v2, may become mandatory in the future.
  currency?: string;

  // Amount that will be withdrawn with this operation
  // (raw amount without fee considerations).  Only
  // given once the amount is fixed and cannot be changed.
  // Optional since **vC2EC**.
  amount?: Amount;

  // Suggestion for the amount to be withdrawn with this
  // operation.  Given if a suggestion was made but the
  // user may still change the amount.
  // Optional since **vC2EC**.
  suggested_amount?: Amount;

  // Maximum amount that the wallet can choose to withdraw.
  // Only applicable when the amount is not fixed.
  // @since **vC2EC**.
  max_amount?: Amount;

  // The non-Taler card fees the customer will have
  // to pay to the bank / payment service provider
  // they are using to make the withdrawal.
  // @since **vC2EC**
  card_fees?: Amount;

  // Bank account of the customer that is debiting, as an
  // RFC 8905 payto URI.
  sender_wire?: string;

  // Base URL of the suggested exchange.  The bank may have
  // neither a suggestion nor a requirement for the exchange.
  // This value is typically set in the bank's configuration.
  suggested_exchange?: string;

  // Base URL of an exchange that must be used.  Optional,
  // not given *unless* a particular exchange is mandatory.
  // This value is typically set in the bank's configuration.
  // @since **vC2EC**
  required_exchange?: string;

  // URL that the user needs to navigate to in order to
  // complete some final confirmation (e.g. 2FA).
  // Only applicable when status is selected or pending.
  // It may contain the withdrawal operation id.
  confirm_transfer_url?: string;

  // Wire transfer types supported by the bank.
```

```
  wire_types: string[];

  // Reserve public key selected by the exchange,
  // only non-null if status is selected or confirmed.
  // @since **v1**
  selected_reserve_pub?: EddsaPublicKey;

  // Exchange account selected by the wallet;
  // only non-null if status is selected or confirmed.
  // @since **v1**
  selected_exchange_account?: string;

  // @deprecated since **v1**, use status instead
  // Indicates whether the withdrawal was aborted.
  aborted: boolean;

  // @deprecated since **v1**, use status instead
  // Has the wallet selected parameters for the withdrawal operation
  // (exchange and reserve public key) and successfully sent it
  // to the bank?
  selection_done: boolean;

  // @deprecated since **v1**, use status instead
  // The transfer has been confirmed and registered by the bank.
  // Does not guarantee that the funds have arrived at the exchange already.
  transfer_done: boolean;
}
```

POST /withdrawal-operation/$WITHDRAWAL_ID

This endpoint is used by the GNU Taler wallet to supply additional details needed to complete a withdraw operation.

Request:

```
interface BankWithdrawalOperationPostRequest {

  // Reserve public key that should become the wire transfer
  // subject to fund the withdrawal.
  reserve_pub: EddsaPublicKey;

  // RFC 8905 (payto) address of the exchange account to be
  // credited for the withdrawal.
  selected_exchange: string;

  // Selected amount to be transferred. Optional if the
  // backend already knows the amount.
  // @since **vC2EC**
  amount?: Amount;
}
```

Response:

200 OK:

The bank has accepted the withdrawal operation parameters chosen by the wallet. The response is a BankWithdrawalOperationPostResponse.

404 Not found:

The bank does not know about a withdrawal operation with the specified `WITHDRAWAL_ID`.

409 Conflict:

- `TALER_EC_BANK_WITHDRAWAL_OPERATION_RESERVE_SELECTION_CONFLICT`: The wallet selected a different exchange or reserve public key under the same withdrawal ID.
- `TALER_EC_BANK_DUPLICATE_RESERVE_PUB_SUBJECT`: the reserve public key is already used.
- `TALER_EC_BANK_UNKNOWN_ACCOUNT`: the selected exchange account was not found.
- `TALER_EC_BANK_ACCOUNT_IS_NOT_EXCHANGE`: the selected account is not an exchange.

- `TALER_EC_BANK_AMOUNT_DIFFERS` : the specified amount will not work for this withdrawal (since **vC2EC**).
- `TALER_EC_BANK_AMOUNT_REQUIRED` : the backend requires an amount to be specified (since **vC2EC**).

Details:

```
interface BankWithdrawalOperationPostResponse {
  // 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
  status: "selected" | "aborted" | "confirmed";

  // URL that the user needs to navigate to in order to
  // complete some final confirmation (e.g. 2FA).
  //
  // Only applicable when status is selected or pending.
  // It may contain withdrawal operation id
  confirm_transfer_url?: string;

  // @deprecated since **v1**, use status instead
  // The transfer has been confirmed and registered by the bank.
  // Does not guarantee that the funds have arrived at the exchange already.
  transfer_done: boolean;
}
```

`POST /withdrawal-operation/$WITHDRAWAL_ID/abort`

Aborts `WITHDRAWAL_ID` operation. Has no effect on an already aborted operation. This endpoint can be used by the wallet if the user aborts the transaction, ensuring that the operation is also aborted at the bank.

Since protocol **v2**.

Request:

The request body is empty.

Response:

204 No content:

The withdrawal operation has been aborted.

404 Not found:

The withdrawal operation was not found.

409 Conflict:

The withdrawal operation has been confirmed previously and can’t be aborted.

Previous
‹ 1.11.4. Taler Bank Revenue HTTP API

Next
1.11.6. Taler Conversion Info API ›

© Copyright 2014-2024 Taler Systems SA (GPLv3+ or GFDL 1.3+).

1.11.3. Taler Wire Gateway HTTP API

This section describes the API offered by the Taler wire gateway. The API is used by the exchange to trigger transactions and query incoming transactions, as well as by the auditor to query incoming and outgoing transactions.

This API is currently implemented by the Taler Demo Bank, as well as by LibEuFin (work in progress).

`GET /config`

Return the protocol version and configuration information about the bank. This specification corresponds to `current` protocol being version **0**.

Response:

200 OK:

The exchange responds with a WireConfig object. This request should virtually always be successful.

Details:

```
interface WireConfig {
  // Name of the API.
  name: "taler-wire-gateway";

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

  // Currency used by this gateway.
  currency: string;

  // URN of the implementation (needed to interpret 'revision' in version).
  // @since v0, may become mandatory in the future.
  implementation?: string;
}
```

1.11.3.1. Authentication

The bank library authenticates requests to the wire gateway via HTTP basic auth.

1.11.3.2. Making Transactions

`POST /transfer`

This API allows the exchange to make a transaction, typically to a merchant. The bank account of the exchange is not included in the request, but instead derived from the user name in the authentication header and/or the request base URL.

To make the API idempotent, the client must include a nonce. Requests with the same nonce are rejected unless the request is the same.

Request:

```
interface TransferRequest {
  // Nonce to make the request idempotent.  Requests with the same
  // request_uid that differ in any of the other fields
  // are rejected.
  request_uid: HashCode;

  // Amount to transfer.
  amount: Amount;

  // Base URL of the exchange.  Shall be included by the bank gateway
  // in the appropriate section of the wire transfer details.
  exchange_base_url: string;

  // Wire transfer identifier chosen by the exchange,
  // used by the merchant to identify the Taler order(s)
  // associated with this wire transfer.
  wtid: ShortHashCode;

  // The recipient's account identifier as a payto URI.
  credit_account: string;
}
```

Response:

200 OK:

The request has been correctly handled, so the funds have been transferred to the recipient's account. The body is a TransferResponse.

400 Bad request:

Request malformed. The bank replies with an ErrorDetail object.

401 Unauthorized:

Authentication failed, likely the credentials are wrong.

404 Not found:

The endpoint is wrong or the user name is unknown. The bank replies with an ErrorDetail object.

409 Conflict:

A transaction with the same `request_uid` but different transaction details has been submitted before.

Details:

```
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 wire transfer initiation performed by the bank.
  // It is different from the /history endpoints row_id.
  row_id: SafeUint64;
}
```

1.11.3.3. Querying the transaction history

GET /history/incoming

Return a list of transactions made from or to the exchange.

Incoming transactions must contain a valid reserve public key. If a bank transaction does not conform to the right syntax, the wire gateway must not report it to the exchange, and send funds back to the sender if possible.

The bank account of the exchange is determined via the base URL and/or the user name in the `Authorization` header. In fact the transaction history might come from a "virtual" account, where multiple real bank accounts are merged into one history.

Transactions are identified by an opaque numeric identifier, referred to here as *row ID*. The semantics of the row ID (including its sorting order) are determined by the bank server and completely opaque to the client.

The list of returned transactions is determined by a row ID *starting point* and a signed non-zero integer *delta*:

- If *delta* is positive, return a list of up to *delta* transactions (all matching the filter criteria) strictly **after** the starting point. The transactions are sorted in **ascending** order of the row ID.
- If *delta* is negative, return a list of up to *-delta* transactions (all matching the filter criteria) strictly **before** the starting point. The transactions are sorted in **descending** order of the row ID.

If *starting point* is not explicitly given, it defaults to:

- A value that is **smaller** than all other row IDs if *delta* is **positive**.
- A value that is **larger** than all other row IDs if *delta* is **negative**.

Request:

Query Parameters:

- **start** – *Optional.* Row identifier to explicitly set the *starting point* of the query.
- **delta** – The *delta* value that determines the range of the 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` milliseconds for new transactions that match the query to arrive and only then send the HTTP response. A client must never rely on this behavior, as the bank may return a response immediately or after waiting only a fraction of `long_poll_ms`.

Response:

200 OK:

JSON object of type IncomingHistory.

204 No content:

There are not transactions to report (under the given filter).

400 Bad request:

Request malformed. The bank replies with an ErrorDetail object.

401 Unauthorized:

Authentication failed, likely the credentials are wrong.

404 Not found:

The endpoint is wrong or the user name is unknown. The bank replies with an ErrorDetail object.

Details:

```
interface IncomingHistory {
  // Array of incoming transactions.
  incoming_transactions : IncomingBankTransaction[];

  // Payto URI to identify the receiver of funds.
  // This must be one of the exchange's bank accounts.
  // Credit account is shared by all incoming transactions
  // as per the nature of the request.
  credit_account: string;

}
```

```
// Union discriminated by the "type" field.
type IncomingBankTransaction =
| IncomingReserveTransaction
| IncomingWadTransaction;
```

```
interface IncomingReserveTransaction {
  type: "RESERVE";

  // Opaque identifier of the returned record.
  row_id: SafeUint64;

  // Date of the transaction.
  date: Timestamp;

  // Amount transferred.
  amount: Amount;

  // Payto URI to identify the sender of funds.
  debit_account: string;

  // The reserve public key extracted from the transaction details.
  reserve_pub: EddsaPublicKey;

}
```

```
interface IncomingWadTransaction {
  type: "WAD";

  // Opaque identifier of the returned record.
  row_id: SafeUint64;

  // Date of the transaction.
  date: Timestamp;

  // Amount transferred.
  amount: Amount;

  // Payto URI to identify the receiver of funds.
  // This must be one of the exchange's bank accounts.
  credit_account: string;

  // Payto URI to identify the sender of funds.
  debit_account: string;

  // Base URL of the exchange that originated the wad.
  origin_exchange_url: string;

  // The reserve public key extracted from the transaction details.
  wad_id: WadId;
}
```

GET /history/outgoing

Return a list of transactions made by the exchange, typically to a merchant.

The bank account of the exchange is determined via the base URL and/or the user name in the `Authorization` header. In fact the transaction history might come from a “virtual” account, where multiple real bank accounts are merged into one history.

Transactions are identified by an opaque integer, referred to here as *row ID*. The semantics of the row ID (including its sorting order) are determined by the bank server and completely opaque to the client.

The list of returned transactions is determined by a row ID *starting point* and a signed non-zero integer *delta*:

- If *delta* is positive, return a list of up to *delta* transactions (all matching the filter criteria) strictly **after** the starting point. The transactions are sorted in **ascending** order of the row ID.
- If *delta* is negative, return a list of up to *-delta* transactions (all matching the filter criteria) strictly **before** the starting point. The transactions are sorted in **descending** order of the row ID.

If *starting point* is not explicitly given, it defaults to:

- A value that is **smaller** than all other row IDs if *delta* is **positive**.
- A value that is **larger** than all other row IDs if *delta* is **negative**.

Request:

Query Parameters:

- **start** – *Optional.* Row identifier to explicitly set the *starting point* of the query.
- **delta** – The *delta* value that determines the range of the 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` milliseconds for new transactions that match the query to arrive and only then send the HTTP response. A client must never rely on this behavior, as the bank may return a response immediately or after waiting only a fraction of `long_poll_ms`.

Response:

200 OK:

JSON object of type OutgoingHistory.

204 No content:

There are not transactions to report (under the given filter).

400 Bad request:

Request malformed. The bank replies with an ErrorDetail object.

401 Unauthorized:

Authentication failed, likely the credentials are wrong.

404 Not found:

The endpoint is wrong or the user name is unknown. The bank replies with an ErrorDetail object.

Details:

```
interface OutgoingHistory {
  // Array of outgoing transactions.
  outgoing_transactions : OutgoingBankTransaction[];

  // Payto URI to identify the sender of funds.
  // This must be one of the exchange's bank accounts.
  // Credit account is shared by all incoming transactions
  // as per the nature of the request.
  debit_account: string;

}
```

```
interface OutgoingBankTransaction {
  // Opaque identifier of the returned record.
  row_id: SafeUint64;

  // Date of the transaction.
  date: Timestamp;

  // Amount transferred.
  amount: Amount;

  // Payto URI to identify the receiver of funds.
  credit_account: string;

  // The wire transfer ID in the outgoing transaction.
  wtid: ShortHashCode;

  // Base URL of the exchange.
  exchange_base_url: string;
}
```

1.11.3.4. Wire Transfer Test APIs

Endpoints in this section are only used for integration tests and never exposed by bank gateways in production.

POST /admin/add-incoming

Simulate a transfer from a customer to the exchange. This API is *not* idempotent since it's only used in testing.

Request:

```
interface AddIncomingRequest {
  // Amount to transfer.
  amount: Amount;

  // Reserve public key that is included in the wire transfer details
  // to identify the reserve that is being topped up.
  reserve_pub: EddsaPublicKey;

  // Account (as payto URI) that makes the wire transfer to the exchange.
  // Usually this account must be created by the test harness before this API
is
  // used.  An exception is the "exchange-fakebank", where any debit account
can be
  // specified, as it is automatically created.
  debit_account: string;
}
```

Response:

200 OK:

The request has been correctly handled, so the funds have been transferred to the recipient's account. The body is a AddIncomingResponse.

400 Bad request:

The request is malformed. The bank replies with an ErrorDetail object.

401 Unauthorized:

Authentication failed, likely the credentials are wrong.

404 Not found:

The endpoint is wrong or the user name is unknown. The bank replies with an ErrorDetail object.

409 Conflict:

The 'reserve_pub' argument was used previously in another transfer, and the specification mandates that reserve public keys must not be reused.

Details:

```
interface AddIncomingResponse {
  // 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 wire transfer initiation performed by the bank.
  // It is different from the /history endpoints row_id.
  row_id: SafeUint64;
}
```

1.11.3.4.1. Security Considerations

For implementors:

- The withdrawal operation ID must contain enough entropy to be unguessable.

Design:

- The user must complete the 2FA step of the withdrawal in the context of their banking app or online banking Website. We explicitly reject any design where the user would have to enter a confirmation code they get from their bank in the context of the wallet, as this would teach and normalize bad security habits.

Previous
‹ 1.11.2. Taler Core Bank API

Next
1.11.4. Taler Bank Revenue HTTP API ›

© Copyright 2014-2024 Taler Systems SA (GPLv3+ or GFDL 1.3+).

Appendix B

Project Management

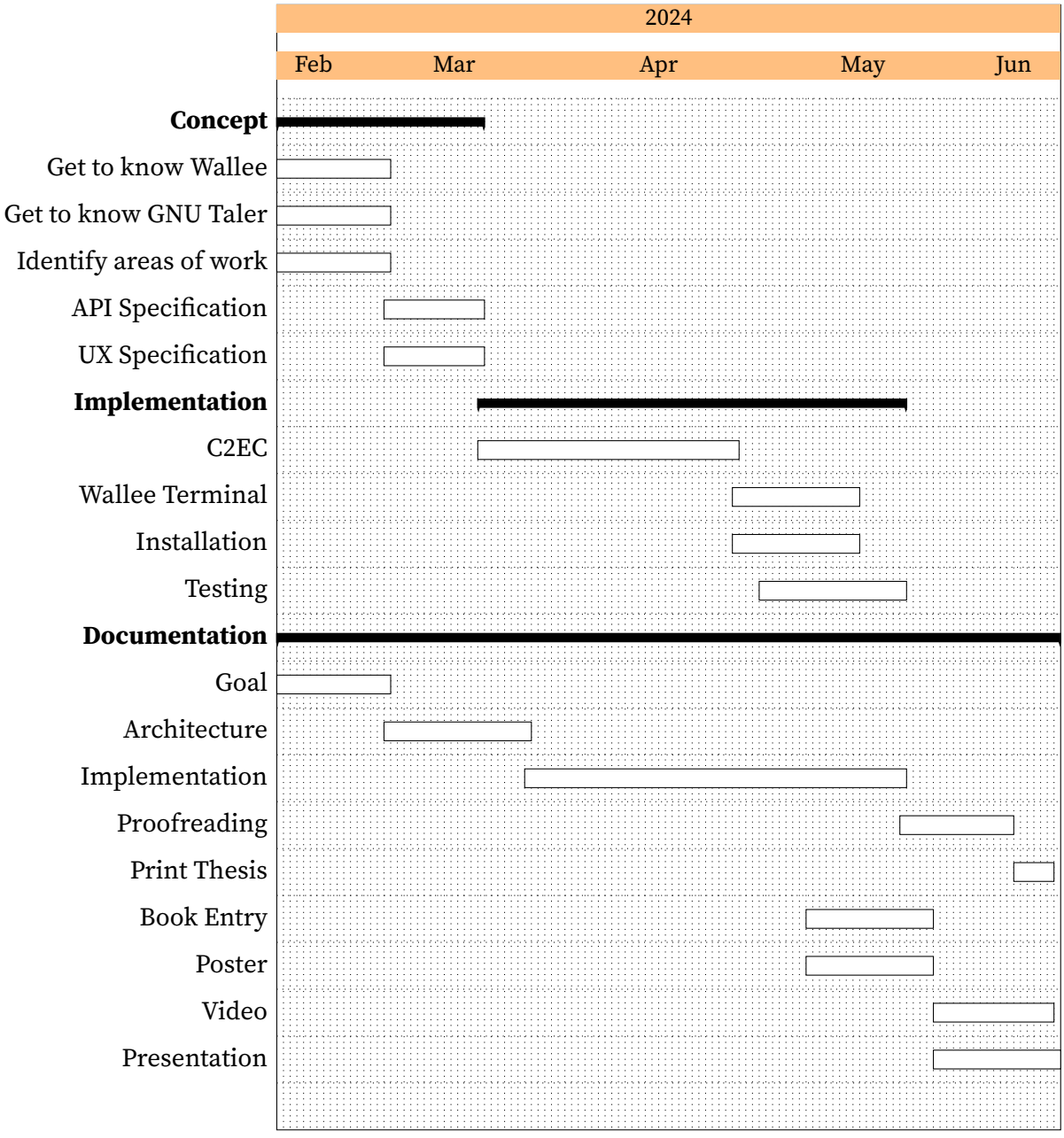

Figure 1: The project plan

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 5.3.3 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 a big project organisation. Requirements are captured as specifications within the Taler documentation repository or in the architecture section (chapter 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 confirmation there is the additional case that neither confirm nor abort is an option and instead retries are required.
- Remove doubled abstractions (Abstracting confirmation 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

15.05.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Poster and Book review
- Wallee was informed about the APK.

Action points

- Fix Poster

22.05.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Poster and Book review
- Setup of C2EC

Action points

- Logging must be enhanced to log every request
- Document Future Work (setup process, etc.)
- Finalize Poster and Book
- Ask Wallee for the APK review.

29.05.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Video: dicsussion about what the video shall contain.

Action points

- Video
- Testing with wallet

05.06.2024

Participants

- Fehrensen Benjamin
- Grothoff Christian
- Häberli Joel

Topics

- Finalizing code
- Finalizing documentation
- Wire-Watch API bugs

Action points

- Send thesis to Dr. Alain Hiltgen
- Ben installs APK when ready