Peer-to-Peer Transactions for Privacy-Preserving Mobile Payments using GNU Taler

J. Florian Kimmes

November 10, 2020

Johannes Gutenberg University Mainz

Faculty 08 - Physics, Mathematics and Computer Science
Institute of Computer Science

Bachelor of Science Thesis

Peer-to-Peer Transactions for Privacy-Preserving Mobile Payments using GNU Taler

J. Florian Kimmes

1. Reviewer Prof. Dr. Nicolai Kuntze
School of Business
University of Applied Sciences Mainz

2. Reviewer Dr. Hans-Jürgen Schröder
Faculty 08 - Physics, Mathematics and Computer Science
Johannes Gutenberg University Mainz

Supervisor Prof. Dr. Nicolai Kuntze

November 10, 2020

J. Florian Kimmes

Peer-to-Peer Transactions for Privacy-Preserving Mobile Payments using GNU Taler

Bachelor of Science Thesis - November 10, 2020

Reviewers: Prof. Dr. Nicolai Kuntze and Dr. Hans-Jürgen Schröder

Supervisor: Prof. Dr. Nicolai Kuntze

Johannes Gutenberg University Mainz

Institute of Computer Science

Faculty 08 - Physics, Mathematics and Computer Science

Saarstr. 21

55112 Mainz

Abstract

GNU Taler offers a privacy-preserving payment system, enabling both online payments and mobile payments. This thesis's objective is to make the system more accessible for merchants by removing a complex server installation that is traditionally required to provide mobile payments through GNU Taler. The proposed new architecture suggests reimplementing the server in a software development kit and integrating it in the merchant's point-of-sales smartphone application. This thesis lays the groundwork for the necessary underlying peer-to-peer communication through two new components: firstly, a proximity-based message exchange component (ProxME) that discovers peers in its environment, automatically connects to the correct peer, and establishes an authenticated and encrypted session; and secondly, HTTP-over-ProxME (HoPME), which offers a framework for web service development on top of ProxME, which facilitates the reimplementation of the necessary endpoints that the merchant's server must provide. The present thesis evaluates the proposed system on the basis of requirements derived from user stories and relevant security standards and finally comes to the conclusion that the proposed system is a strong candidate solution for the thesis's research objective, based on its improved usability properties.

Contents

1 Introduction

Mobile payments are becoming increasingly more popular and slowly but steadily supersede cash payments in modern everyday life. Current research shows that the mobile payment transaction volume in Germany — in fact one of the slowest adopters of mobile payments in Europe [1] — has increased from just 133 million Euros in 2017 to a projected 8.3 billion Euros in 2020 [2]. With an increasing proportion of purchases made through mobile payments, so decreases the customers' privacy. Cash payments are by nature anonymous. With cash transactions, neither participant knows anything about the other party. This is in stark contrast to market-leading mobile payment providers, such as Apple Pay, and Google Pay [1]. All common mobile payment providers use systems in which an intermediary sees and processes all transaction details, such as the initiator, receiver, time, and paid amount of the transaction. Depending on the purchase, this can be highly sensitive data and is highly susceptible to violations of privacy. The rising popularity of mobile payments, however, shows that customers perceive an added value in using them. This is confirmed by a recent survey of consulting company pwc [1], in which survey participants state advantages such as convenience, simplicity, and ease of budgeting. Problems, according to participants, on the other hand, are a lack of security and privacy, a loss of control with stolen devices, and a perceived encouragement of spending more quickly. The discrepancy between the perceived benefits on the one hand, and the risks associated with the technology on the other hand, leads to the assumption that in the future mobile payments need a better technological foundation.

Recent research on mitigating privacy risks of online payments spawned the efforts around the GNU Taler project, hereinafter simply referred to as Taler. Taler offers a fully-fledged electronic online payment system for privacy-preserving payments. It offers anonymity to the customer, but enforces income-transparency for the merchant, by design. This results in the desirable situation, in which the customer benefits from complete privacy, while the merchant must disclose all their income, which makes tax evasion and other illegal business activities more difficult (see chapter 2 for an overview of the Taler system or [3] for an in-depth discussion of the technology). Additionally to online payments, Taler can be used for mobile payments as well, and therefore solves the discussed privacy issues that are inherent to the currently popular mobile payment solutions described above. Spreading the advantages of this new technology to a broader audience is desirable.

Naturally, one of the major adoption factors for new payment systems is the number of available merchants. It is therefore crucial to make the onboarding experience for merchants as seamless as possible. Unfortunately, the Taler setup is quite complex for mobile payment merchants, consisting of a point-of-sales terminal application (POS app) and a backend server that customers connect to and that handles the payment processes via HTTP messages. Setting up a server presents a major barrier to entry for a merchant that wishes to give the Taler system a trial. In contrast to webshops that must already maintain a server and large store chains that can potentially justify the added expenses, small brick and mortar shops likely do neither have the necessary know-how nor resources to deploy a production-grade payment server.

This leads to the research question of the present thesis: how to make the GNU Taler payment system more accessible for mobile payment merchants? The proposed hypothesis is that a peer-to-peer architecture that connects the customer's device directly to the POS app of the merchant preserves Taler's privacy guarantees and additionally solves the installation complexity and therefore improves Taler's adoption rate with merchants.

This thesis designs a prototype implementation of the proposed alternative peer-to-peer architecture, in which a Taler backend reimplementation is integrated with the POS app. It develops two necessary components that provide a new communication layer that easily integrates into existing Taler apps. The first component handles the low-level peer-to-peer networking details. It discovers peers in close proximity through the wireless capabilities provided by the smartphone, handles the connection process to the chosen peer, and provides an authenticated and encrypted session for Taler transactions. Building on the first component, the second component provides a framework to facilitate the reimplementation of the necessary Taler backend services inside the POS app. It features parsing and routing capabilities for HTTP requests over the established peer-to-peer connection. A reimplementation of the Taler backend is out of scope for this thesis but is assumed as given for the further discussion.

Chapter 2 gives an introduction to the previous academic work done on Taler, as well as the project's current development state. Chapter 3 discusses the requirements necessary for the proposed peer-to-peer system to work, including functional, non-functional, and security requirements. A high-level overview of the architecture proposed by this thesis is given in chapter 4, which is followed by the two main modules that together form the majority of the proposed system in chapter 5 and

6 respectively. Finally, in chapter 7 the proposed system is evaluated in regard to usability and security requirements.

Related Work 2

2.1 GNU Taler

The Taler project aims to “provide a payment system that makes privacy-friendly online transactions fast and easy” [4]. In 2016, Burdges et al. [5] designed the open Taler protocol for electronic online payments. Taler is a payment system that is based on the assumption that customers should have cash-like anonymity, but merchants should be held accountable for transactions in order to be taxable. In 2019, Florian Dold wrote his PhD thesis about the Taler system, contributing detailed insight into the system [3].

By comparing the currently available payment systems, the authors identified several design goals that their modern electronic payment system must accomplish: the customer’s privacy must be preserved, transactions must be taxable, a healthy payment provider competitions must be fostered and lastly the underlying currency must be non-volatile (see [3] for a detailed discussion).

2.1.1 Overview

This section gives a high-level overview of the Taler payment system. A more in-depth overview is provided by [5]. In order to explain the basic principles of the Taler architecture and the interaction of its components, the key actors in the Taler system are described in the following.

Taler payments are enabled by Taler “coins”. These coins are cryptographic tokens that are passed between the different actors of the system as monetary transactions. There are four key actors in the Taler system (illustrated in fig. 2.1).
The *Customer* anonymously withdraws coins from a Taler exchange and wire-transferrs the counter-value of those coins to an escrow account at this exchange. The customer stores the coins in her Taler wallet, a digital store on her device. She can later spend the withdrawn coins at a Taler merchant.
The *Exchange* is a financial service provider that provides Taler coins to customers (withdrawal) and receives Taler coins from merchants (deposition). It holds escrow funds for all withdrawn coins in circulation that are wire-transferred to merchants

upon depositing.
The *Merchant* receives coins in exchange for goods and services. She deposits received coins at the exchange in return for a wire transfer of the respective value.
The *Auditor* regulates Taler exchanges. Taler is designed to work in the legal framework of existing payment providers. Exchanges are trusted entities in this system, in the same way as banks and other payment providers are trusted entities in existing payment systems. Given the fact that Taler coins are not a new currency but only a representation for merchants that indicate that there is a counter-value for them in an escrow account at the exchange, it becomes clear that the exchange provider must be audited. Without audits, a rogue exchange provider could create coins with no counter-value to defraud merchants and customers.

The cash-like cycle of coins, in which each participant can check the authenticity of received tokens without authenticating the transaction partner, leads to the anonymity of the customer. This is paired with an "online-check" of received coins on the merchant-side that forces the merchant to deposit coins as soon as they are received, in order to ensure their authenticity (see section 2.1.3 for details). This "online-check" provides an obstacle for devious merchants that want to engage in illegal activities (e.g., tax evasion).

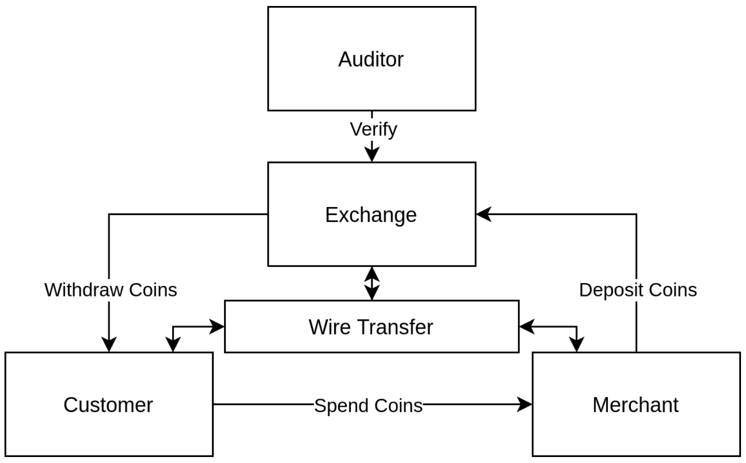


Fig. 2.1: A Taler system overview [5].

2.1.2 Blind Signatures

Blind signatures form the basis of Taler coins, the cryptographic tokens that are withdrawn from an exchange by customers and are then exchanged with merchants in the process outlined in the sections below. This section gives a basic introduction to the concept of blind signatures as introduced by [6].

The principle of blind signatures for anonymous payments has been suggested as early as 1983 by David Chaum [6].

The general idea is to be able to cryptographically sign a digital document without knowing the contents of it. In the case of anonymous cryptographic money, a bank must be able to sign its monetary tokens without knowing the underlying secret.

Chaum proposes using a private signing function s', its publicly known inverse s, a secret commuting function c and its inverse c'. s and s' behave exactly as one would expect from signing functions: $s(s'(x)) = x$, with s revealing no information about the nature of s'.

The important property of c is that it is commuting with s (i.e., $c(s(x)) = s(c(x))$), resulting in the desirable property:

$$c'(s'(c(x))) = s'(c'(c(x)))$$

$$\Leftrightarrow c'(s'(c(x))) = s'(x)$$

Additionally, $c(x)$ reveals no information about the nature of x. This makes it possible for the bank to sign $c(x)$, without knowing what x is. The customer can later strip the commuting function c, by applying c' to the signed value. Chaum proposes the following protocol to provide anonymous currency:

1. A customer generates a secret x, calculates $c(x)$ and sends it to the bank.

2. The bank signs the received value with s' and returns $s'(c(x))$ to the customer.

3. The customer can strip the obfuscating commuting function c by applying c', yielding $c'(s'(c(x))) = s'(x)$.

4. Anyone can use public s to verify the signed $s'(x)$.

Note that in order for the bank to know the value of the newly issued coin, every signed token has the same value in this system, as the bank has no insight into what it signs. Various options for introducing coin denominations are discussed in [3]. However, the Taler system settled for using multiple signing keys to represent different denominations. In this more elaborate approach, the signing party (i.e., the Taler exchange) has multiple signing keys that it uses to sign different coin denominations. Each public signing key has a value and some metadata associated with it, allowing all participants in the system to check the denomination of a coin by comparing the signature to the keys provided by the issuing exchange.

2.1.3 Taler Coins

Taler coins are implemented using a public key pair. Having access to a coin's private key makes a person the owner of said coin. Coins are only valuable if they are (blindly) signed by an exchange, signifying a certain counter-value in escrow, depending on the denomination associated with the signing key.

Withdrawing coins from an exchange works very similar to the protocol described by Chaum:

1. The customer starts the withdrawal process and makes a wire transfer from her bank account to the exchange.

2. The customer generates a new key pair and sends the blinded public key to the exchange.

3. The exchange signs the key with the negotiated denomination signing key and returns it to the customer.

4. The customer strips the obfuscating commuting function.

Customers hold their newly issued coin in a digital *wallet*. After receiving a signed coin from the exchange a customer can hold it there until she decides to spend it.

Spending coins does not require any merchant-side customer authentication, as the signed coins already guarantee a counter-value to the merchant from the exchange. This allows for an easier and quicker check-out experience than current authentication-based flows. In order to facilitate the integration with Taler for webshops, Taler offers a *merchant backend* component that exposes an HTTP interface to the webshop. This backend performs all cryptographic and operational processes, like signing contracts, verifying signatures and storing transaction data (see fig. 2.2). This makes it easier for merchants to integrate Taler into their webshop, as all necessary Taler code is provided and encapsulated by an HTTP API.

In addition to the added convenience for merchants, this architecture adds security benefits, by encapsulating critical functions in a separated component which is secured from direct access [5].

Once a customer wishes to pay a physical or digital commodity, the webshop creates a contract, containing the terms of purchase. The contract gets signed by the backend and then forwarded to the customer's wallet. The wallet signs the contract with

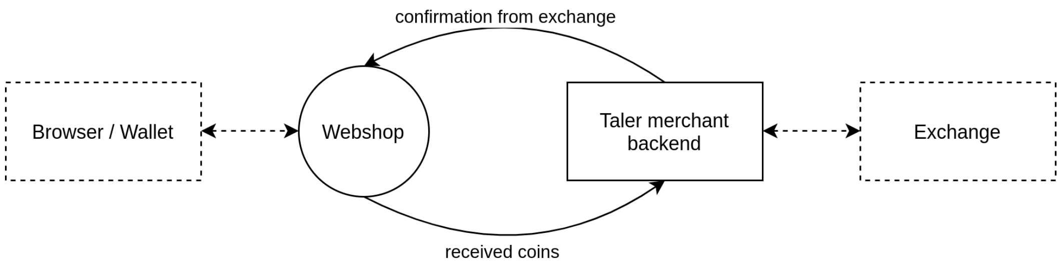


Fig. 2.2: An example of the Taler backend performing validation and depositing of coins [5]. The webshop frontend delegates all coin handling to the backend and waits for a response.

enough coins (i.e., private keys of coins) to settle the costs laid out in the contract, and sends it with the necessary coins to the webshop, which passes it on to its backend for verification (roughly outlined in fig. 2.2).

The backend deposits the received coins at the issuing exchange, which can verify its own (stripped) signature for each coin (as described in section 2.1.2). The exchange can now either directly make a wire transfer to the merchant or aggregate multiple deposits from the same merchant to settle the lump sum in one large wire transfer to reduce costs. Either way, the exchange signals the outcome of the verification to the merchant backend, which can then report it to the webshop to trigger the appropriate business logic. It is important to note that without directly depositing the coins, the merchant cannot receive any value, because a customer could attempt to double-spend the coins. Only the merchant that deposits a certain coin first receives its value from the exchange. This mechanism creates a safeguard against tax evasion, as every purchase is registered.

2.2 Conclusion

Taler provides an easy and quick checkout flow for customers of webshops and point-of-sale terminals, making it as convenient as other modern payment solution. Additionally, it solves all other problems with current mobile payment solutions that were stated by survey participants (see chapter 1). A lack of privacy was stated as a concern that is addressed by Taler with its primary design goal. Customers purchases are private from the exchange and from the merchant. Another concern was the percieved lack of security of current solutions. Since Taler is developed as free and open-source software, source code audits are possible for everyone, including governments and other institutions that have an interest in a stable and secure payment infrastructure. This hints at a potentially more secure system than that of alternatives. Losing a mobile-payment enabled smartphone was also stated as a concern. Since there is only ever a limited amount of Taler coins in a wallet (the same way only a limited amount of cash is in a physical wallet), the risks of stolen devices can be assessed much more clearly than with other mobile payment wallets that are connected to credit cards or bank accounts. The risk of losing a Taler wallet is the risk of losing the contained coins, however, the risk of losing a device connected to a credit card is much harder to assess.

All in all, it can be said that Taler solves the motivating problems from chapter 1 very elegantly. It is clear that Taler provides a good solution for private payments with the additional benefits outlined above making it a good fit for the underlying technology of this thesis.

3 Requirements Analysis

The following section develops the requirements that must be met by the proposed peer-to-peer system. They are divided into functional requirements, non-functional requirements, and security requirements. The proposed system is referred to simply as *the system* throughout the following requirements. Two methodologies are used to identify the requirements for the system. First, user stories are used to model user expectations of the resulting system. This method is a user-centric software engineering technique in which requirements are derived from short sentences written from the user's perspective. Only functional and non-functional requirements, excluding security requirements, are collected here. As a second methodology, two relevant security standards are examined for applicable security requirements.

3.1 User Stories

The user stories described in this section are inspired by the problems described in chapter 1. For each user story, a set of requirements is deduced. In order to avoid duplication, requirements are only listed for the first user story they apply to.

User Story 1:
"As a merchant I want to quickly set up a POS terminal app on my smartphone to receive Taler payments."

Functional Requirement 1: The system must accept incoming connections on the merchant's smartphone.
Functional Requirement 2: The system must display a QR code to convey the necessary connection data to the customer.
Functional Requirement 3: The system must be able to scan connection data from a QR code.
Functional Requirement 4: The system must establish a peer-to-peer connection to exchange information between the customer and the merchant.
Functional Requirement 5: The system must be able to send and receive HTTP requests.
Functional Requirement 6: The system must allow the exchange of Taler API objects.

Functional Requirement 7: The system must expose the necessary Taler APIs to set up a Taler POS terminal.
Functional Requirement 8: The system must expose the necessary Taler APIs to set up a Taler wallet.
Non-Functional Requirement 1: The system must be compatible with the latest Taler merchant backend specification.
Non-Functional Requirement 2: The system must be downloadable as an app.

User Story 2:
"As a customer I want to be able to make a payment regardless of the specifications of the merchant's smartphone."

Non-Functional Requirement 3: The system must work on smartphones running Android 5.0 or later.
Non-Functional Requirement 4: The system must support Bluetooth LE as a transport technology.
Non-Functional Requirement 5: The system can support other transport technologies such as Bluetooth, Wifi Direct, NFC, and ultrasonic sound.

User Story 3:
"As a customer I want payments to be quick and easy."

Non-Functional Requirement 6: To set up a connection, the system must exchange all necessary information with another system through scanning of exactly one QR code.
Non-Functional Requirement 7: The duration of a transmission of coins for a payment must be at most as long as a credit card payment.
Non-Functional Requirement 8: The system must allow payments independently of the exchange providers used by the participants to foster competition [7].

User Story 4:
"As a customer I want to rest assured that my payment information is secure."

Non-Functional Requirement 9: The system must preserve all privacy characteristics of the Taler system [7].
Non-Functional Requirement 10: The system must only disclose the minimal amount of information necessary [7].
Non-Functional Requirement 11: The system must be implemented as free software (see [8] for a definition) to provide transparency and user freedom [7].

Non-Functional Requirement 12: The system must be easily testable and therefore dependable.
Non-Functional Requirement 13: The system must facilitate code-reuse to avoid complexity.

3.2 Security Standards

Several standards exist in the space of (mobile) payment security. This section outlines key security requirements as recommended by two relevant authorities.

3.2.1 OWASP Mobile Application Security Verification Standard

The OWASP Mobile Application Security Verification Standard (MASVS) [9] offers a general security guide for mobile application developers. It is important that security critical applications like payment apps adhere to these essentials. OWASP MASVS was developed to provide developers and users with a metric to determine the security level of their applications. The following list contains all relevant requirements as defined by MASVS.

Security Requirement 1: All app components are identified and known to be needed.
Security Requirement 2: A high-level architecture for the mobile app and all connected remote services has been defined and security has been addressed in that architecture.
Security Requirement 3: All app components are defined in terms of the business functions and/or security functions they provide.
Security Requirement 4: The app only requests the minimum set of permissions necessary.
Security Requirement 5: All security controls have a centralized implementation.
Security Requirement 6: The app should comply with privacy laws and regulations.
Security Requirement 7: There is an explicit policy for how cryptographic keys are managed, and the lifecycle of cryptographic keys is enforced.
Security Requirement 8: The app uses proven implementations of cryptographic primitives.
Security Requirement 9: The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.

Security Requirement 10: The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.
Security Requirement 11: The app does not re-use the same cryptographic key for multiple purposes.

3.2.2 Payment Card Industry Data Security Standard

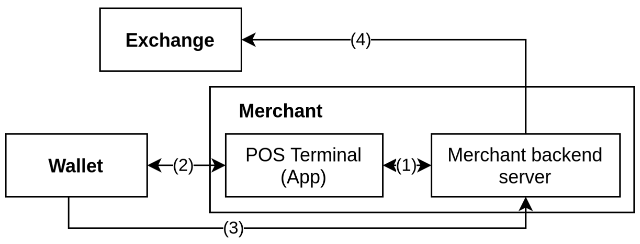
The Payment Card Industry Data Security Standard (PCI DSS) [10] is a standard for payment card processors. Even though Taler is not a payment card system, this standard is still applicable, since it has extensive sections on POS terminal security. The following list is an excerpt of the standard, containing only the most important and relevant requirements defined by PCI DSS. Requirements already laid out by the previous sections are not reiterated.

Security Requirement 12: Keep customer data storage to a minimum.
Security Requirement 13: Do not store sensitive authentication data after authorization (even if encrypted). If sensitive authentication data is received, render all data unrecoverable upon completion of the authorization process
Security Requirement 14: Use strong cryptography and security protocols to safeguard sensitive data during transmission over open, public networks.
Security Requirement 15: Maintain a documented description of the cryptographic architecture that includes details of all algorithms, protocols, and keys used for the protection of sensitive data, including key strength and expiry date.
Security Requirement 16: Store cryptographic keys in the fewest possible locations.
Security Requirement 17: Limit access to system components and sensitive data to only those individuals whose job requires such access.

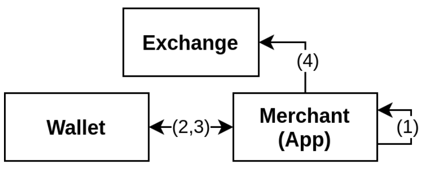
4 High-Level Architecture

The following chapter provides an architectural overview of the proposed peer-to-peer system. Firstly, the general idea of moving the Taler backend to an SDK is discussed. Secondly, the key components that make up the SDK are presented. Lastly, the internal relation of the components is described.

The original Taler design focuses heavily on separating the existing webshop business logic from Taler logic. That makes perfect sense when catering for existing webshops, as it reduces the amount of duplicated code and provides defence in depth by separating critical code from the webshop's attack surface. For mobile payments, however, it makes the setup of merchant infrastructure significantly more complicated (see fig. 4.1 for a comparison). The proposed peer-to-peer architecture takes a radically different approach, found most often in mobile applications. Instead of separating the merchant backend into an extra server, it is instead wrapped into a software development kit (SDK) and provided to mobile application developers as a library dependency. This paradigm shift has great implications for the usability of point-of-sale terminal apps: a merchant must only download a single app to her smartphone instead of installing and provisioning a server or hiring a system administrator to do said task. While an SDK does not provide the same strict separation as an extra HTTP connection, it still avoids code duplication in merchants' implementations and provides a well-defined API as a soft separation of concerns to help auditability.



(a) The traditional Taler payment process with a POS terminal.



(b) The payment process with a POS terminal in a peer-to-peer context.

Fig. 4.1: A comparison between the traditional Taler payment process and a peer-to-peer approach (see chapter 2 for details on the Taler payment process).
(1) The terminal creates a contract in the backend. (2) The terminal sends the contract terms to the wallet. (3) The wallet sends coins to the backend. (4) The backend deposits the coins at the exchange.

The proposed peer-to-peer Taler SDK (P2PTalerSDK) is designed as an Android library that can be included in existing projects, such as the Taler POS app [11]. It comprises several important components that together make up all features of a peer-to-peer enabled Taler backend server (fig. 4.2 illustrates a high-level overview of all components). Two components, the Proximity-based Message Exchange (ProxME) library and the HTTP-over-ProxME (HoPME) library are provided as prototypes by this thesis. ProxME provides the low-level communication capabilities and HoPME provides a framework for developing web services that run over ProxME. A core component that is out of scope and therefore not part of the present thesis prototype is the Taler backend server logic. In a traditional setup, it consists of all HTTP endpoints that are exposed by the backend server. Even though it is out of the scope of this thesis to provide a full reimplementation of the backend server, it is nevertheless assumed in the P2PTalerSDK architecture for the sake of further analysis (see section 8.2 for implementation recommendations).

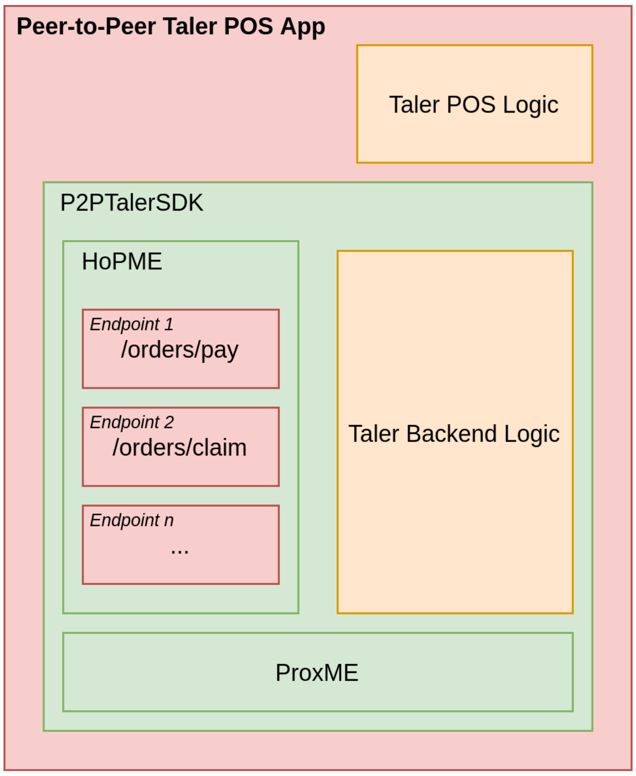


Fig. 4.2: A high-level overview of the proposed architecture, showing the interrelation of existing and new components. Green components are part of this thesis's prototype, orange parts are existing Taler components that must be adapted, and red parts do not yet exist.

Note that this thesis focuses mainly on the architectural challenges of the merchant-side in order to investigate the posed research question. However, to be able to successfully communicate, both participants must implement the proposed architecture. Peer-to-peer wallets can use either the ProxME implementation to send the necessary HTTP requests to a merchant's peer-to-peer app or use the traditional implementation to connect to a regular backend server. Ideally, P2PTalerSDK would provide the wallet's API as well, however, that it out of scope for the thesis at hand (see section 8.2 for suggested actions).

Internally P2PTalerSDK is designed as a layered architecture (see fig. 4.3). The layers are modelled after the network stack that the SDK implements.

ProxME, as the lowest layer, is responsible for discovering compatible devices nearby, connecting to the appropriate partner device, and setting up an authenticated and encrypted session. The established session allows exchanging arbitrary-length byte sequences that can be sent and initiated from both directions.

Built on top of this low-level session, the next layer provides an abstraction for the necessary HTTP communication. Traditionally, HTTP is implemented on the TCP/IP stack, but generally it is protocol-agnostic [12]. The HTTP-over-ProxME (HoPME) layer features a server-side development framework. It receives, parses, and routes HTTP requests. Controllers that are provided by the framework user determine the HTTP endpoint's features. HTTP Responses are automatically generated from response objects, also defined in said controllers.

Using these two building blocks, the P2PTalerSDK layer provides the actual backend functionality to both the wallet and the implementing app. The backend's HTTP endpoints are split into two categories. The first category contains all endpoints that are consumed by the wallet in the traditional Taler architecture. These HTTP endpoints are implemented using HoPME and are therefore exposed to the wallet over a peer-to-peer connection. The second category contains all HTTP endpoints that are traditionally consumed by the POS app (or webshop). These endpoints are implemented using regular Java methods and are exposed as part of the P2PTalerSDK library API. Apps that use the SDK, can therefore access these endpoints through method calls.

The components that make up the architecture are all encapsulated as libraries that work independently of the higher layers and independently of the project. This has the considerable advantage that the solutions generalize beyond the Taler use case. The omnidirectional real-time communication, offered by ProxME could potentially be used in many diverse applications, such as games, collaboration software, or other payment solutions. Similarly, the process of porting any set of HTTP-connected components to a peer-to-peer context is substantially facilitated using the HoPME library.

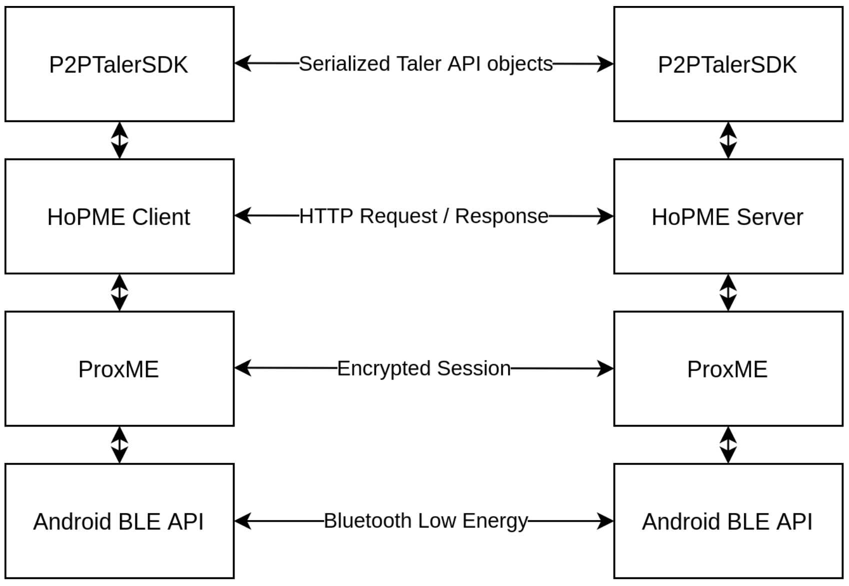


Fig. 4.3: An overview of all layers in the P2PTalerSDK architecture, using Bluetooth LE as an example for the underlying wireless transport channel.

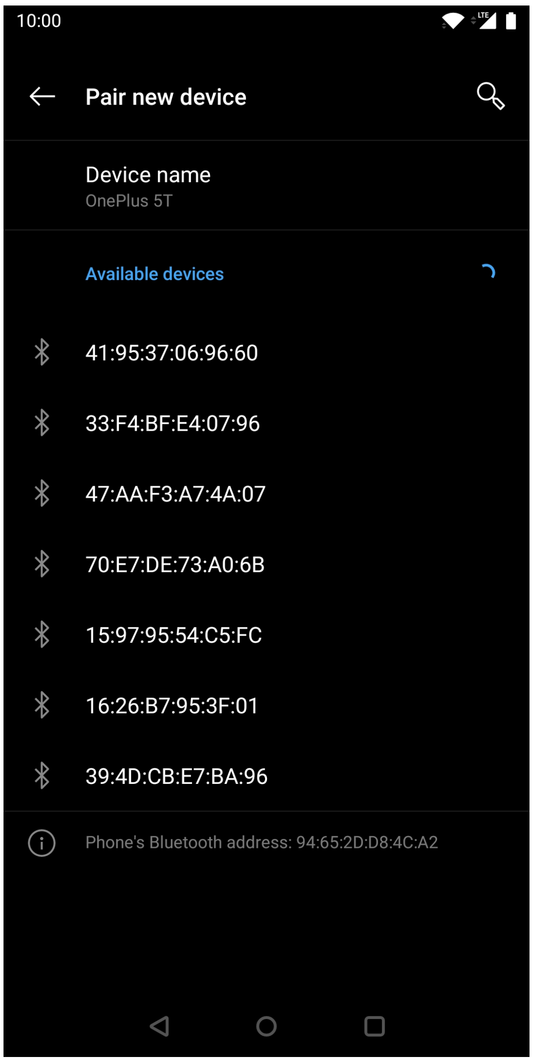
5 Proximity-Based Message Exchange (ProxME)

This section describes how ProxME solves the general problem of securely connecting two devices in physical proximity for communication, without relying on a network connection. Solving this problem requires several steps. Firstly, the correct communication participants have to be discovered from a pool of potential participants in any given environment. Secondly, both participants have to negotiate a communication technology that both devices support. Lastly, the established transport channel has to be secured (i.e., encrypted and authenticated).

5.1 Discovery and Negotiation

Two devices running ProxME must have a mechanism to signal each other that they are expected to connect. This phase in the connection is usually called pairing (e.g., within the context of Bluetooth [13]) and is performed by presenting the user a set of discovered device names and letting her select the one she wishes to connect to (see fig. 5.1). With ProxME, two users have to explicitly express their intent to connect their devices to each other as well, however, the user experience is more streamlined. In the context of mobile payments, in which this work is embedded, users already have a mental model of how the payment process should work: holding their smartphone near the POS terminal or scanning a QR code, followed by a confirmation of the amount in an app. The ProxME setup does not disrupt this model, as all it takes to connect is scanning a QR code. ProxME introduces two roles to manage connections: an *acceptor* role and a *connector* role. The acceptor creates and displays a QR code and asks the connector to scan it. After the connector scans the QR code, both participants have all the necessary information to connect to each other, negotiate the means of transmission and to start a key exchange.

In order to set up a connection, ProxME determines all supported means of transmission on each device. This may include any commonly supported radio transmission technologies, hereinafter called *transport channels*, such as Bluetooth, Bluetooth Low Energy, Wifi Direct, or even ultrasonic sound emitted from regular speakers [14]. All these channels have different terminology and techniques to form the initial connection, but all have similar semantics when it comes to connecting: one


10:00

Pair new device

Device name
OnePlus 5T

Available devices

41:95:37:06:96:60

33:F4:BF:E4:07:96

47:AA:F3:A7:4A:07

70:E7:DE:73:A0:6B

15:97:95:54:C5:FC

16:26:B7:95:3F:01

39:4D:CB:E7:BA:96

Phone's Bluetooth address: 94:65:2D:D8:4C:A2

Fig. 5.1: An example of a non-optimal user experience for payments: Android presents the user a list of devices to pair with when scanning for Bluetooth devices.

device publishes information about itself and one or many other devices scan their surroundings for published information, connecting to the transmitting device automatically or on command. In order to unify the terminology, publishing is referred to as *advertising* and scanning is simply referred to as *scanning*.

Figure 5.2 outlines the discovery process. The acceptor's device generates a random UUID (1) that is later used to identify whom to connect with and a new ephemeral key pair for this connection's encryption. This information — the UUID and the public key, together with the supported channels — is then encoded in the QR code (2) and presented to the connector (3). The connector uses this information and their own set of supported channels to advertise on the preferred channel (4). Channel priorities may be determined by transfer speed, reliability, and user preference. Usually, the connector should choose the fastest channel offered by the acceptor. Simultaneously, the acceptor scans on all offered channels for the newly generated

UUID (4). Once the acceptor has received an advertisement package with the correct UUID, both participants start a cryptographic handshake over that channel (5).

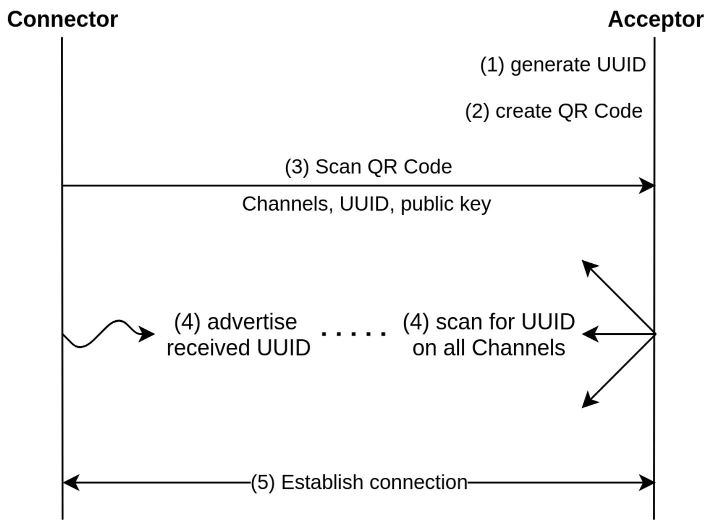


Fig. 5.2: The ProxME discovery process.

5.2 Secure Channel

In order to allow for secure communication, the participants set up application-layer encryption on top of the established transport channel link. This encryption layer is necessary because the underlying channel may not have any encryption at all, as is the case with ultrasonic signalling [14], or the encryption ends early, as with the Bluetooth Low Energy stack on Android, where all installed apps have access to received packets [15].

5.2.1 Sodium

In adherence to security best-practices, all cryptographic implementations described in this chapter rely on a well-established cryptographic library: Sodium [16], an open-source and third-party audited project. It contains implementations of hard to misuse primitives that can be used as building blocks for higher-level protocols. To facilitate the discussion of the key exchange protocol below, the following section explains the used concepts briefly.

Sealed Boxes allow an anonymous sender to encrypt a message to the public key of a receiver. While the receiver can check the integrity of the message, she cannot verify the identity of the sender [17]. Therefore, sealed boxes represent a one-way channel for a single confidential message. Sealed Boxes are in the following denoted as: $sealedBox_{pK}(m)$, with $pK := public\ key$, $m := message$.

Secret Streams offer the same guarantees as secret boxes per message but additionally establish a stream state for both participants to check for duplicate or removed messages in a stream of messages. By providing a simple means to 'ratchet' keys, they also make it easy to achieve perfect forward secrecy [18]. Secret Streams are in the following denoted as: $secretStream_{k,sss}(m)$, with k := $shared\ key$, $sss := secret\ stream\ state$, $m := message$.

Sodium offers, additionally to the explicitly mentioned primitives, miscellaneous other functions that were used, such as secure key generation, secure nonce generation, and cryptographic hashing among others.

5.2.2 Key Exchange

The presented cryptographic protocol is split into two phases: The *handshake phase* (see fig. 5.3, $M_0 - M_3$), in which two participants (connector and acceptor) agree on a common secret k and the *transport phase* (see fig. 5.3, $M_4 - M_n$), in which the common secret is used to set up a two-way secret message stream that has all properties of a secure channel. The handshake only authenticates the acceptor. If the connector must also be authenticated, this has to be done in the transport phase by a separate authentication scheme (e.g., comparing key fingerprints in person).

The connector initiates the handshake by scanning a QR code containing the acceptor's ephemeral public key p_A directly from her device's screen (fig. 5.3, M_0). The connector then generates a pre-session key e and encrypts it to p_A (denoted as $sealedBox_{p_A}(k)$ in fig. 5.3, M_1). Both the acceptor and the connector can then calculate the same 256-bit session key $k = h(p_A||e)$, by concatenating p_A and e, and hashing the result using the SHA256 hash function. The acceptor acknowledges the new key through a secret stream, the state $(A \rightarrow C)$ of which is initialized with the session key k and from then on used to encrypt messages from the acceptor to the connector (fig. 5.3, M_2). After receiving the acknowledgement, the connector sets up her own secret stream state $(C \rightarrow A)$ and in turn sends an encrypted acknowledgement to the acceptor (fig. 5.3, M_3). Both participants now hold two secret stream states. One for receiving messages and one for sending messages. This is a robust mechanism for asynchronous communication because each state is only incremented when receiving or sending a message respectively. At this stage, the transport phase begins and both participants can use their respective secret stream to encrypt messages to each other. If either participant receives an invalid message at any point in the handshake, they abort and reset their handshake state. The other participant times out and resets their handshake state as well. To keep the protocol's complexity to a minimum, and to prevent side-channel attacks, there is no mechanism to resend lost messages.

M_0 functions as the root of trust in this protocol. Because of the in-person, out-of-band transmission, the connector can be certain that p_A belongs to the acceptor and that, therefore, only the acceptor has the matching private key, essentially authenticating anyone with knowledge of the private key s_A as the acceptor. Due to the nature of the transmission, this message cannot be tampered with by an attacker. An attacker may be able to shoulder surf the QR code (i.e., eavesdrop), but not change it (i.e., tamper with p_A). Since the acceptor generates a new key pair for every protocol run, the connector also knows that all messages that rely on p_A are fresh (i.e., are not replayed from previous runs). With p_A the connector has already all the information she needs in order to calculate the session key k. The connector generates a new pre-session key e that she then hashes together with the received public key p_A to yield k. The result depends on fresh inputs from both participants, preventing replay attacks.

M_1 delivers the pre-session key e to the acceptor. This is the main common secret used for the encryption of the secret channel. Since it is encrypted with p_A, only the acceptor can decrypt it and later use it, as she holds the matching private key s_A. Additionally, as only the connector received the public key p_A for this session (out-of-band), and guessing the random 256-bit value is infeasible, an attacker must shoulder surf the QR code to be able to impersonate the connector. Note that this mechanism doesn't fully authenticate the connector, but introduces a substantial defence against mass-impersonation in an attacker's environment.

Messages M_2 and M_3 establish the secret stream on both sides of the connection and acknowledge the shared secret. The acknowledgement messages are encrypted with the session key k that contains 256 bits of entropy from both participants respectively. Therefore, both participants have a guarantee that these messages are fresh.

The scheme guarantees perfect forward secrecy for every message throughout an established session, as well as for every session itself. By creating a new public key p_A and new pre-session key e in every handshake, every session provides forward secrecy. An attacker only gets insight into exactly the current session (i.e., neither previous sessions nor future sessions), should either key be compromised. Additionally, Sodium's Secret Stream implementation offers perfect forward secrecy for messages in the stream's context. While the streams are initialized with the common secret k, each message is encrypted with a key that is newly derived from k. An attacker that can break the encryption of one message, can therefore not decrypt messages sent previously in the session.

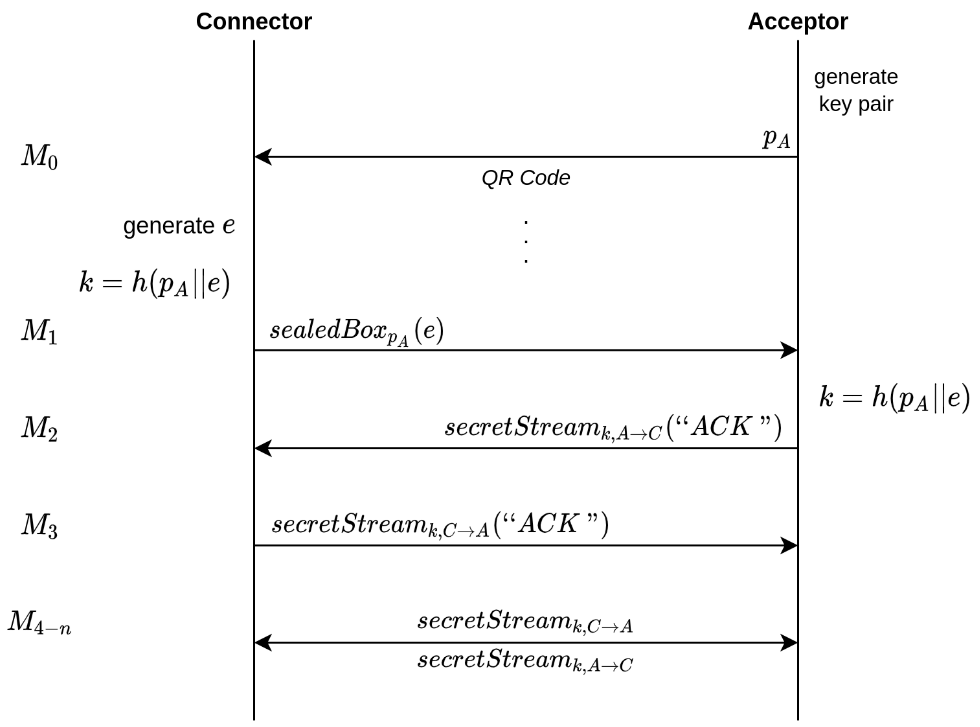


Fig. 5.3: The key exchange protocol scheme that establishes a secure channel between the connector and the acceptor.

5.3 Implementation

ProxME's implementation consists mainly of three modules. Each of these modules is divided into two corresponding implementations: a connector implementation and an acceptor implementation. While this implementation split is necessary to offer different functions and processes depending on the role, the implementations are often very similar. The first module is split into two classes, the `Acceptor` and `Connector`, which offer an interface to the library user. Internally they manage the connection state and house the other modules. The second module offers an interface for the client and server of the different transport channels and their respective implementations. Last but not least, the state machines that implement the key exchange protocol are also split into the two roles outlined above and work in sync to handle the key exchange and offer the user a secure channel.

5.3.1 Acceptor and Connector

The main interfaces that a (framework) user interacts with, are the `Acceptor` class and the `Connector` class (API classes). Depending on what role the device should assume, one of the two must be instantiated. Listing 5.1 gives an example of the process. On an exemplary button click in line 21 ff., a new `Acceptor` is created.

Lines 2–19 define the `ProxMeCallback` that handles various state changes in the connection. ProxME hides a lot of the underlying complexity from the user. It only messages the user to show the QR code string (on the `Connector`'s side, the scanned QR code string is passed to the class constructor), to notify the user of connection status changes, and to notify the user of received messages. The only other public methods are `start()` and `sendMessage(byte[] messageBytes)`. The former starts the advertising/scanning process, and the latter encrypts and sends `messageBytes` to the connected party.

```java
1  Acceptor acceptor;
2  ProxMeCallback acceptorCallback = new ProxMeCallback() {
3      @Override
4      public void qrStringReady(String qrString) {
5          Bitmap qr = QrCode.build(qrString);
6      }
7
8      @Override
9      public void connectionEstablished() {
10         acceptor.sendMessage("foo".getBytes());
11     }
12
13     @Override
14     public void messageReceivedSuccess(byte[] messageBytes) {
15         Log.d(TAG, "Received message: " + new String(messageBytes));
16     }
17
18     //...
19 }
20
21 void onConnectClick(View view) {
22     acceptor = new Acceptor();
23     acceptor.onStateChanged(acceptorCallback);
24     acceptor.start();
25 }
```

Listing 5.1: An example showing the usage of the Acceptor class's simple public interface.

Internally, multiple steps are executed on initialization. First, all available channels are enumerated. A modern flagship Android smartphone may support many technologies, such as Bluetooth LE, Bluetooth, Wifi Direct and NFC, whereas older smartphones usually come with a more restricted feature set. For the prototype of the present thesis, only the Bluetooth Low Energy channel is implemented to provide a proof-of-concept.

Then, the QR code string is generated or parsed, depending on the executed role. Using a string (i.e., bytes) instead of binary data is a practical decision, based on the available QR encoders and decoders. The additional overhead that this representation causes does not exceed the size constraints of a quickly readable QR code and therefore does not warrant implementing a new QR encoder and decoder on Android. The QR code structure is presented in figure 5.4. The first four bytes represent the protocol version that the acceptor supports. A parser must read these first four bytes and then parse the next bytes as outlined in the specification of that version. For the proposed version `0000`, the next eight bytes encode the supported channels of the acceptor. Each of the eight positions represents one channel. Making it possible to encode all possible combinations of eight supported channels. The implemented prototype assumes that the first position represents Bluetooth Low Energy and therefore only sets the first byte to one, with all others set to zero (i.e., `10000000`). The next 32 byte long field contains the *acceptor UUID*, a unique ID that identifies every connection (see section 5.1). Lastly, the acceptor's public key for this connection is attached as a hex-encoded 64-byte string.

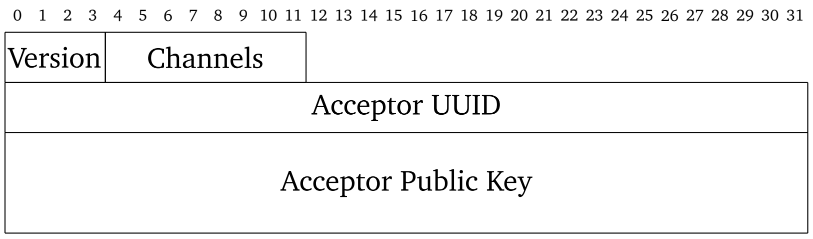
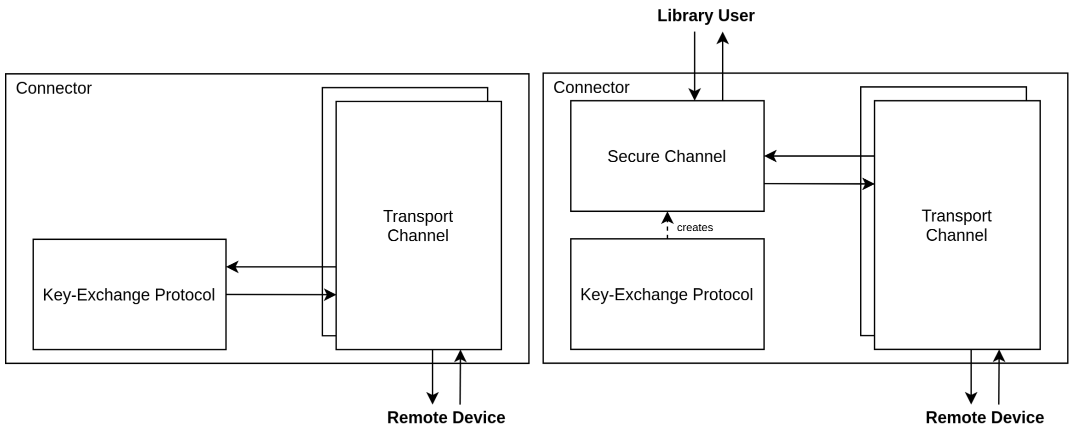


Fig. 5.4: The QR code string definition for the ProxME discovery, as defined by version `0000` of the protocol.

The connection setup procedure is also coordinated by the API classes. Figure 5.5 shows how incoming and outgoing packets are routed in the different communication phases, to ensure that only ever encrypted messages are exchanged. Every channel implements the `ProxMeChannel` interface (see lst. 5.2). This allows the `Connector` and `Acceptor` to pipe messages to the channel without knowing the exact implementation of the channel (e.g., Bluetooth LE or Wifi Direct). In order to guarantee that the user cannot accidentally send or receive unencrypted messages, the instantiated API class implements two separate callbacks for the channel (see lst. 5.2, line 3): one for the key exchange and one for the user. Only when the key exchange protocol module reports a successful handshake, is the user callback registered. This ensures that the key exchange, as outlined in section 5.2.2, is completed, before any message is sent over the secure channel.

The `Acceptor` and `Connector` classes can be therefore seen as the public API for ProxME that implement the plumbing mechanisms for the underlying transport channels and the key exchange module.



(a) During the handshake phase, the `Connector` directs all communication to to the key exchange protocol module.

(b) After the key exchange module reports success, it creates a secure channel module to which the `Connector` directs all future communication.

Fig. 5.5: An overview of the interaction of the main components in the ProxME system, using the example of the `Connector` class.

5.3.2 Transport Channels: Bluetooth LE

The prototype transport channel implementation is the Bluetooth LE network stack on Android. Bluetooth LE was chosen because it offers a good common denominator for cross-platform availability over several smartphone generations [15], [19]. A high-level abstraction was introduced for transport channels, making the integration of other transport channel implementations trivial. The `ProxMeChannel` interface (see lst. 5.2) must be implemented by both sides of the connection, even if the underlying paradigm assigns separate roles to the communication participants. Bluetooth LE is one such technology. The Bluetooth Core Specification [13] outlines

```java
public interface ProxMeChannel {
    void sendMessage(byte[] message);
    void onStateChanged(ProxMeChannelStateChangedCallback callback);
    void start();
    ArrayList<byte[]> split(byte[] messageBytes);
}
```

Listing 5.2: The channel interface that every transport channel must implement.

how a Bluetooth LE connection must be established. The specification separates the connection into a peripheral and a central role. The peripheral acts as a server that offers a set of services and the central acts as a client that consumes services offered by surrounding peripherals. In the context of ProxME, the acceptor assumes the role of the client and the connector assumes the role of the server.

The Android operating system handles the low-level details of the Bluetooth LE stack and only exposes interactions with the Generic Attribute Profile (GATT Profile). The GATT Profile is the highest abstraction of the Bluetooth LE stack. It establishes a framework of operations and definitions for how data must be transferred and stored. The specification states that data is stored in GATT Characteristics. A GATT Characteristic is simply a value accompanied by optional information on how the data should be represented (GATT descriptors). GATT Characteristics are stored in GATT services, which are "a collection of data and associated behaviours to accomplish a particular function or feature of a device or portions of a device" [20]. Multiple services together form a profile that fulfills a use case (see fig. 5.6). The GATT Profile integrates with the central–peripheral duality by enabling

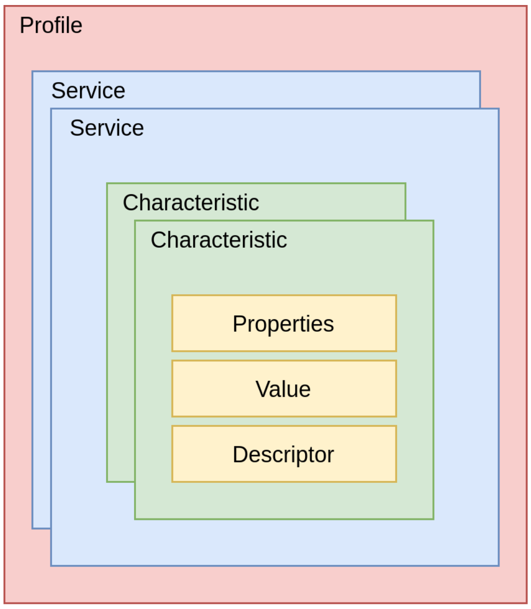


Fig. 5.6: The GATT Profile structure as outlined in the Bluetooth Core Specification [13].

a remote key-value store. A characteristic is offered on the peripheral (i.e., server) that can be remotely accessed by the central (i.e., client). The client can read values from and write values to a characteristic, depending on the permissions that are set by the peripheral. Additionally, a central can choose to subscribe to a characteristic for notifications. The peripheral can then notify all subscribed centrals, whenever something noteworthy happens on the peripheral (e.g., a characteristic value changes).

Before a central can use a peripheral, it has to discover it. In Bluetooth LE the ATT protocol that powers GATT offers *advertising packets* [21]. These packets hold up to 37 bytes and can be used to simply broadcast information or to signal that a peripheral has a service that is *connectable*. Scanning applications can filter discovered advertising packets marked as connectable for certain criteria and then connect via the GATT Profile method described above. In ProxME, advertising is done on the connector's side because the Android operating system's permission system equates "access to Bluetooth Low Energy scanning" with "access to the devices physical position [15]". To use an app that scans, it is, therefore, necessary to grant location permissions and have the Android location provider enabled at all time. This is inconvenient for a wallet app since customers would essentially be forced to have their location provider enabled at all times for quick and easy payments. A merchant on the other hand has a well-defined time and location where she would need to have the location provider enabled, namely on her job, in the store. It is therefore an acceptable trade-off to scan on the merchant's app instead of the customer's wallet.

As ProxME has only one task — sending arbitrary data — it has only one GATT service: the *ProxME GATT service*. This service is advertised with the acceptor UUID that the connector received via the QR code. The connector therefore filters for that UUID and can be certain that no device other than the intended partner advertises it. In line 2–7 of listing 5.3, a `ScanFilter` is set up using the `acceptorId` from the QR code. The `ScanSettings` can be tuned to a high-burst, low-latency mode without impacting the battery performance too much (see lst. 5.3, line 9–11), because the scan stops as soon as the other device has been discovered, with a maximum duration set to 30 seconds. This performance mode allows for quicker discovery.

```
1  ArrayList<ScanFilter> filters = new ArrayList<>();
2  filters.add(new ScanFilter.Builder()
3          .setServiceUuid(
4              ParcelUuid
5                  .fromString(ProxMeProfile
6                      .createUuidFromString(acceptorId).toString()))
7          .build());
8  ScanSettings settings = new ScanSettings.Builder()
9      .setCallbackType(ScanSettings.CALLBACK_TYPE_FIRST_MATCH)
10     .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
11     .build();
12 bluetoothLeScanner.startScan(filters, settings, bleScanCallback);
```

Listing 5.3: Setting up an Android Bluetooth LE scan filter for the received `acceptroId`.

The ProxME GATT service holds the ProxME *Message Characteristic*, which is used as a postbox. After a Bluetooth LE connection has been established, both participants can use this characteristic to message the other party. On the one hand, the central can write to the characteristic and therefore effectively send a message to the peripheral. The peripheral, on the other hand, can notify the central of own changes and by this means send messages to it.

Android's maximum transmission unit (MTU) on the ATT layer is 23 bytes by default but ProxME negotiates the highest possible MTU that is supported by both devices, up to 517 bytes. All `ProxMeChannel` implementations must offer a `split(byte[] messageBytes)` method to split messages longer than the negotiated MTU, to ensure that every message is properly encrypted and authenticated. If a user wants to send a message longer than the negotiated MTU, the message is split into MTU-sized chunks and each chunk must be encrypted and authenticated using the secure channel laid out in section 5.3.3.

Listing 5.4 outlines the algorithm used to split long messages into MTU-sized chunks. The algorithm assumes an incoming message as bytes in `mBytes` and the negotiated

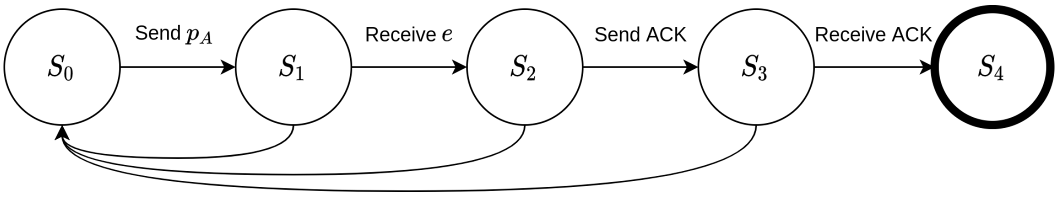
```
1  ArrayList<byte[]> messages = new ArrayList<>();
2
3  // iteration message
4  byte[] m_i;
5
6  // step through message in mtu-sized chunks
7  for (int i=0; i < mBytes.length; i=i+mtu) {
8      // last iteration
9      // (current position + mtu exceeds message length)
10     if (i+mtu >= mBytes.length) {
11         m_i = new byte[mBytes.length - i];
12         System.arraycopy(mBytes, i, m_i, 0, mBytes.length - i);
13     }
14     // iteration i
15     else {
16         m_i = new byte[mtu];
17         System.arraycopy(mBytes, i, m_i,0, mtu);
18     }
19     messages.add(m_i);
20 }
```

Listing 5.4: The split method of the ProxME Bluetooth LE channel implementation. `mBytes` is a `byte[]` that holds the original message and `mtu` is the negotiated maximum transmission unit. After terminating `messages` holds a list of `byte[]` with a length of at most `mtu`.

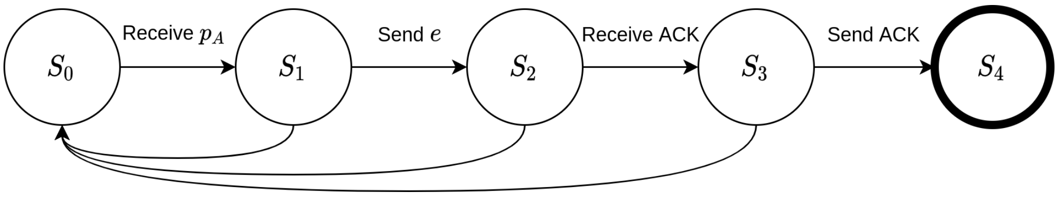
MTU as an int in `mtu`. Line 1 instantiates the `ArrayList` that holds a list of `byte[]` that together make up `mBytes` and of which none has a length that exceeds `mtu`. `m_i` holds the current iteration's message chunk. The loop from line 7 to line 20 iterates over the original message in MTU-sized steps, checking if it is the last iteration (line 10), and if not, simply copying the chunk to the `m_i` and adding it to the `messages`. If it is the last iteration, it bounds-checks the remaining bytes and copies them to `m_i`.

5.3.3 Protocol State Machine

ProxME uses deterministic finite-state machines to ensure that the key exchange either finishes successfully or fails early. There are mechanisms in place to prevent side-channel attacks from timing information, as described further below. Each party initiates a state machine on connection. Instead of reading symbols like conventional state machines, they advance their states by sending and receiving valid protocol messages. By doing so, they stay in sync and finish the handshake. As soon as one party receives an invalid message, fails a cryptographic check , or otherwise gets an invalid input, it discards any received input and resets to the initial state. Figure 5.7 shows how the two state machine implementations complement each other. Each state represents a message in the protocol outlined in section 5.2.2, one noteworthy exception being state S_4, which represents the successful handshake and is implemented by a hand-over to the `SecureChannel` implementation outlined in section 5.3.4. Once the state machine reaches S_4 it uses the exchanged secret to create said `SecureChannel` and signals via a registered callback that it successfully finished the handshake, triggering that all future traffic, incoming and outgoing, will be routed to this new object that handles encyption and decryption.



(a) The state machine modelling the acceptor's handshake behaviour.



(b) The state machine modelling the connector's handshake behaviour.

Fig. 5.7: The transition functions of the deterministic finite-state machines that model both sides of the connector–acceptor handshake.

Figure 5.8 shows the chosen architecture for the state machine implementation. The abstract `KeyExchangeProtocol` class holds the current protocol state (`HandshakeState`) and provides methods to advance this protocol state. The server (`KeyExchangeProtocolServer`) and client (`KeyExchangeProtocolClient`) extend this abstract class and provide their implementations of the four `HandshakeStates` of the respective protocol description as private classes. Additionally, a `reset()` method is provided by the concrete child class to reset the state machine to the correct initial state and to remove all cryptographic material from the protocol.

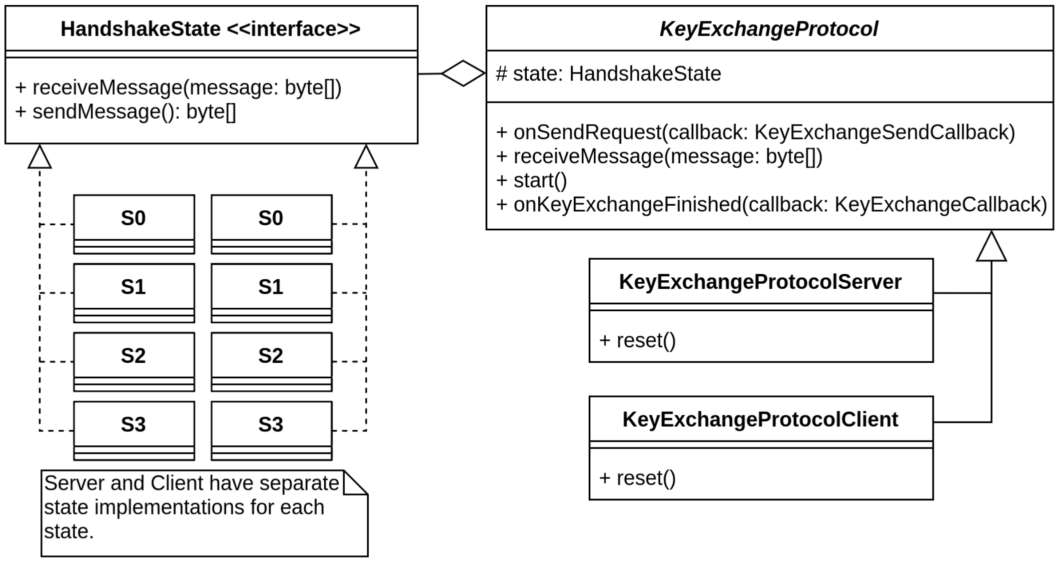

HandshakeState <<interface>>
+ receiveMessage(message: byte[])
+ sendMessage(): byte[]
KeyExchangeProtocol
# state: HandshakeState
+ onSendRequest(callback: KeyExchangeSendCallback)
+ receiveMessage(message: byte[])
+ start()
+ onKeyExchangeFinished(callback: KeyExchangeCallback)
S0
S0
S1
S1
S2
S2
S3
S3
KeyExchangeProtocolServer
+ reset()
KeyExchangeProtocolClient
+ reset()
Server and Client have separate state implementations for each state.

Fig. 5.8: A simplified architectural diagram, outlining the main key exchange protocol implementation.

Once a protocol run is started using the `start()` method, the initial state is loaded and executed. Using the acceptor's initial state as an example, this section describes the state progression (see lst. 5.5). The `HandshakeState` interface (line 1) dictates two methods. First, the `receiveMessage` method, which is called by the protocol context (`KeyExchangeProtocol`) as soon as it receives a message from the transport channel (see fig. 5.5a). The protocol context passes itself and the received message to this method (line 3), in order to process the message and then store state-independent information in the protocol context. Second, the `sendMessage` method. It is called internally by the protocol context anytime a valid message was successfully received. This method also takes the `KeyExchangeProtocol` context as a parameter, to store state-independent information. Additionally, it returns the calculated response as a byte array, which will be forwarded to the transport channel.

In the presented example, any incoming message is invalid, because the acceptor does not expect a message in the initial state. The typical approach to handle this case is, therefore, to reset the protocol, notify via a callback, and abort the handshake (line 5–8). If a send is requested in this state, it generates a new key pair (line 16), advances the state variable of the context to the next state (line 25) and returns the public key to be sent (line 26).

This architecture is designed to facilitate auditing the soundness of the cryptographic implementations in ProxME. All cryptographic components of the handshake are

```
private class S0 implements HandshakeState {
    @Override
    public void receiveMessage(KeyExchangeProtocol context,
                               byte[]              message) {
        context.reset();
        HandshakeException ex = new HandshakeException("...");
        context.finishedCallback.keyExchangeAborted(ex);
        throw ex;
    }

    @Override
    public byte[] sendMessage(KeyExchangeProtocol context) {
        Box.Lazy lazyBox = (Box.Lazy) lazySodium;

        try {
            context.kp_A = lazyBox.cryptoBoxKeypair();
            context.p_A = context.kp_A.getPublicKey();
        } catch (SodiumException e) {
            context.reset();
            HandshakeException ex = new HandshakeException("...", e)
            context.finishedCallback.keyExchangeAborted(ex);
            throw ex;
        }

        context.state = new S1();
        return context.p_A.getAsBytes();
    }
}
```

Listing 5.5: The initial state (S0) of the acceptor uses the `sendMessage` method to create a new key pair. The `receiveMessage` implements the reset and signals a failed handshake through a callback.

encapsulated in the described classes, outlined in this chapter, and are only exposed to the rest of the system through a well-defined API. All steps of the protocol are encapsulated in a separate state class and are therefore easy to comprehend. Since the transition function (see fig. 5.7) is mostly linear, it is easy to audit as well.

5.3.4 Secure Channel

Once the handshake is completed, instead of advancing to the next state, the last state notifies the API class of the success by using the `keyExchangeSuccessful` method in the `KeyExchangeCallback` (see fig. 5.8). This method takes a `SecureChannel` object (5.9) as a parameter. This contains all necessary key material for the two-way encrypted stream.

The `SecretStream.State` [18] is Sodium's implementation of a secure channel (see section 5.2.1). It is initialized on both sides with a 192-bit random nonce N that is exchanged with the ACK messages in the handshake, as outlined above. Once initialized with the session key and N, the state uses an incremented counter nonce to ensure that messages cannot be removed, reordered, or duplicated. Message authentication additionally ensures that messages cannot be truncated or modified. [18].

The API classes interact with the secure channel through the use of the `encryptMessage` method and the `decryptMessage` method. To send a message the outgoing message must first be split according to the transport channel's MTU size, as outlined in section 5.3.2. The resulting `ArrayList<byte[]>` is then passed to the `encryptMessage`, which encrypts each message part (`byte[]`) separately. Every message part contains a tag indicating if it is the last in the series of parts comprising a longer message, or if it is an intermediate link. Each received message part is passed to the `decryptMessage` method upon receipt. There, the message is decrypted and the attached tag is checked. If it is an intermediate message, it is stored in the `receivedMessageQueue` and the method returns null. Else, if it is the last message part the queue is concatenated and the resulting complete message is returned.

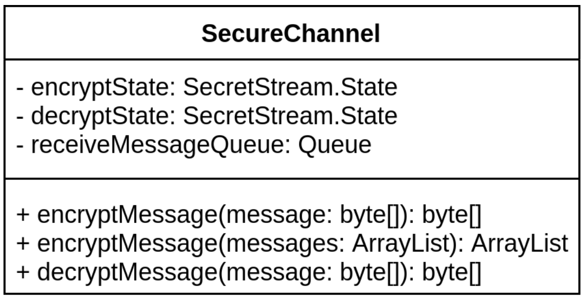

SecureChannel
- encryptState: SecretStream.State
- decryptState: SecretStream.State
- receiveMessageQueue: Queue
+ encryptMessage(message: byte[]): byte[]
+ encryptMessage(messages: ArrayList): ArrayList
+ decryptMessage(message: byte[]): byte[]

Fig. 5.9: A diagram that outlines the secure channel implementation.

HTTP-over-ProxME (HoPME) 6

HTTP-over-ProxME (HoPME) provides a framework to process HTTP requests in a peer-to-peer manner. One Android device must assume the role of a client and one must assume the role of a server. While usually, HTTP communication takes place over TCP/IP connections [12], the proposed design uses the ProxME protocol for the underlying transport instead. Two devices connect via the mechanisms outlined in section 5.1 and then use the resulting persistent connection for multiple HTTP exchanges. The framework can be thought of as a peer-to-peer web service provider framework (WS provider). Note that the term web service only refers to the web technology HTTP, but does not imply the need for a connection to the world wide web. This thesis provides only the WS provider framework, as that is required to build the Taler merchant backend web service. However, for meaningful communication, the client must implement the equivalent endpoints as a WS consumer. Section 8.2 discusses potential future work on the client-side of HoPME.

6.1 Web Service Provider Framework

The WS provider framework's main purpose is to provide boilerplate code, to facilitate writing ProxME-integrated web service endpoints. The framework user writes only one WS method per endpoint. Related endpoints are then pooled in a controller, which is used by HoPME for routing. The framework handles routing of incoming requests, parsing of HTTP requests, and automatic generation of HTTP responses. This allows for rapid development of peer-to-peer web service APIs such as the Taler merchant backend API.

The general structure of the framework are illustrated in figure 6.1. The server component initializes a ProxME acceptor and registers for incoming connections using the provided callbacks, as outlined in chapter 5. After that, the server functions mainly as a mediator between the transport layer and the router. Incoming requests are passed to the parser. The parser deconstructs the request into its component parts, namely the HTTP request method, the URI, the headers, and the payload, and stores this information in a machine-readable `Request` object. The router then uses this information to route the request to the correct controller. The router uses Java annotations in the user-defined WS method implementation to determine where to route the request to. Routing is determined by the HTTP request method and the URI. The WS method in the controller must specify the parameter it expects from the request and return a `Response` object, containing a status code, and optionally a payload. The server then delegates the `Response` to the HTTP generator and sends the resulting HTTP response via the ProxME API to the client.

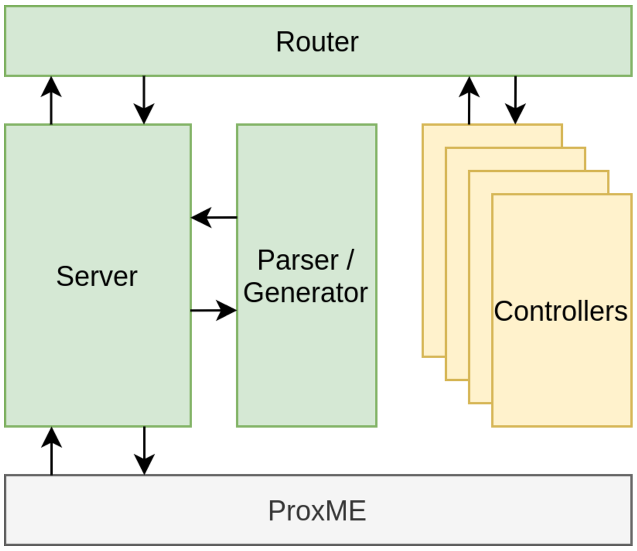


Fig. 6.1: The HoPME framework architecture, with all framework-provided components colored green, and the user-provided components colored yellow.

Using the framework is simple. To implement a new WS endpoint, the user simply creates a WS method in which she implements the feature that should be offered by the new web service. HoPME organizes WS methods in *controllers*. Controllers are a collection of WS methods that represent endpoints that are logically associated. These controllers are then registered with the router and are thereby activated. A WS method is a standard Java method, but with exactly one Java annotation added, indicating the WS endpoint and the HTTP request method that it represents (see lst. 6.1, line 3). The annotation name indicates the HTTP method type (e.g., `@GetMapping` for GET, `@PostMapping` for POST, etc.), and the annotation value specifies the URI at which the endpoint is located (e.g., "/orders/4"). Any path segment in the URI can be replaced by a variable. Such *path variables* must be surrounded by braces (e.g., `/orders/{ORDER_ID}`), and must be specified as a parameter in the Java method signature (see lst. 6.1, line 6). Every method parameter must correspond to one of three parameter types defined in the following and must be annotated accordingly. *Path variables*, denoted by the braces described above must be annotated as `@PathVariable("foo")` where the annotation value must be equal to the variable in the path segment. *Request parameters* are key-value pairs, appended to the URI. Each pair is separated by an ampersand: e.g., `/orders?foo=bar&foo1=bar1`. These are annotated with `@RequestParameter("foo")`, where the annotation value represents the parameter key. HTTP requests have an optional *request body* that can be specified as a parameter annotated as `@RequestBody`. The method arguments

```java
public class TalerPaymentsController extends Controller {

    @GetMapping("/orders/{ORDER_ID}")
    public Response getOrder(@RequestParameter("foo") boolean foo,
                             @RequestParameter("bar") String bar,
                             @PathVariable("ORDER_ID") int orderID,
                             @RequestBody PaymentRequest request) {

        //...

        PaymentResponse response = new PaymentResponse();
        return new Response(StatusCodes.TEAPOT, response);
    }
}
```

Listing 6.1: An example HoPME controller class, demonstrating how a WS method looks like.

are extracted from the routed HTTP request and are passed to the appropriate WS method parameter, based on the aforementioned Java annotations. Since HTTP requests are parsed from ASCII text and encoded data [12], the extracted values must be cast to the parameter types defined in the method signature. This casting ensures that the WS method invocation is type-safe. Malformed HTTP requests, with invalid parameter values, are therefore discarded before reaching the controller logic, making any type checking logic redundant.

Custom user-defined objects are "cast" using serialisation. Usually web services use JSON or XML to transport payloads [22]. Taler specifies API objects that are exchanged between the client and the merchant backend web service, serialized as JSON. However, because of the harsh network conditions of ProxME, HoPME automatically serializes API objects as CBOR. CBOR uses the JSON data model of nested key-value mappings with various pre-defined data types, but instead of a human-readable format, it uses a concise binary representation [23]. This results in a much smaller message size than JSON, especially when transferring binary data, because JSON uses a base64 encoding for binary data, adding more overhead to the message size. The smaller message size in turn leads to shorter transmission times [23]. The CBOR serialization and deserialization happens transparently at the framework level, enabling the user to simply work with API objects, defined on both sides, without having to worry about transport encodings.

Every WS method defined in a HoPME controller, must return a `Response` object. This response object holds a response code and optionally an arbitrary object (see lst.

6.1, line 12). When processing the response, the HTTP generator will serialize the object to CBOR, as described above, and add it to the generated HTTP response.

6.2 Implementation

The HoPME implementation encapsulates much of the complexity that the user is spared. In order to offer the possibility of a simple controller implementation to the framework user, HoPME parses the Java annotations of each method and provides algorithms to route requests accordingly. Java annotations are evaluated by using Java's reflection API. The process of reflection allows a Java program to introspect and change behaviour at runtime. It can therefore parse annotations and dynamically build a map of routes. The routing implementation and an HTTP parser and HTTP generator are presented in the following sections.

6.2.1 HTTP Parser and Generator

The first important feature of HoPME is parsing and generating HTTP messages. HTTP messages are defined in [12]. A client sends *HTTP requests* and the server, thereupon, sends *HTTP responses*. An example request to the Taler project's demo backend is displayed in listing 6.2 and the server's response can be seen in listing 6.3.

Requests consist of three segments. Firstly, the Request-Line, containing the HTTP request method, the URI with optional request parameters and the protocol version (fig. 6.2, line 1) must be specified. The request method specifies the semantic meaning of the request and the URI points to the requested resource. Request parameters are key-value pairs concatenated by an equal sign character. They are separated from the URI by a question mark character and from each other by an ampersand character. Secondly, a series of header lines follow, each containing a header-key and a header value, separated by a colon character (line 2–5). Lastly, an optional payload following a blank line may be added. This payload consists of binary data, the encoding of which must be specified in the Content-Type header. As outlined above, HoPME always uses CBOR as a payload encoding.

Responses have a similar structure to requests (see lst. 6.3 for an example). The first line, the Status-Line is constructed from the protocol version, a status code and textual representation of the status code. The status code represents the result of the server's "attempt to understand and satisfy the request" [12]. The structure of the rest of the responses is the same as the structure of requests outlined above.

There are several open-source Java implementations of HTTP parsers. However, there are no production-grade implementations that are independent of the TCP/IP stack. Parsing and generating HTTP is simple since all characters, including delimiters, are ASCII characters. Constructing and deconstructing messages is well-defined in the protocol specification and this thesis's implementation does not involve any innovation, other than that it is available without any coupling to the TCP/IP stack. The implementation details will therefore not be reiterated here. The parser creates a `HttpRequest` object from the extracted information, which can be used and understood by other HoPME components. Analogously, the generator assembles valid HTTP responses from `HttpResponse` objects.

```
GET /orders/202...Q7E?token=N35...GNW HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: backend.demo.taler.net
```

Listing 6.2: An exemplary GET HTTP request, issued to the demo backend server operated by the Taler project.

```
HTTP/1.1 402 Payment Required
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Encoding: deflate
Content-Length: 272
Content-Type: application/json
Date: Thu, 05 Nov 2020 11:40:15 GMT
Server: nginx

{
    "already_paid_order_id": null,
    "fulfillment_url": "https://shop.demo.taler.net/en/essay/...",
    "taler_pay_uri": "taler://pay/backend.demo.taler.net/202..."
}
```

Listing 6.3: The Taler demo backend server's HTTP response to a GET request to the `/orders/<orderId> endpoint.`

6.2.2 Router

The router, as the second and arguably more interesting feature of HoPME, has three important tasks. First, it holds a map of all routes. A route is a link between an HTTP request method plus URI and a WS method. The user's controllers are registered with the router and the router extracts a route for each WS method. Second, it routes incoming requests to the correct controller and WS method. Since this is the most common task of the router, this must be a fast operation. Lastly, it maps and casts arguments from the HTTP request to the required parameter type in the WS method. The following section describes the necessary processes that are executed to accomplish the outlined tasks. Figure 6.2 presents a simplified diagram illustrating the involved Java classes and their relation to each other.

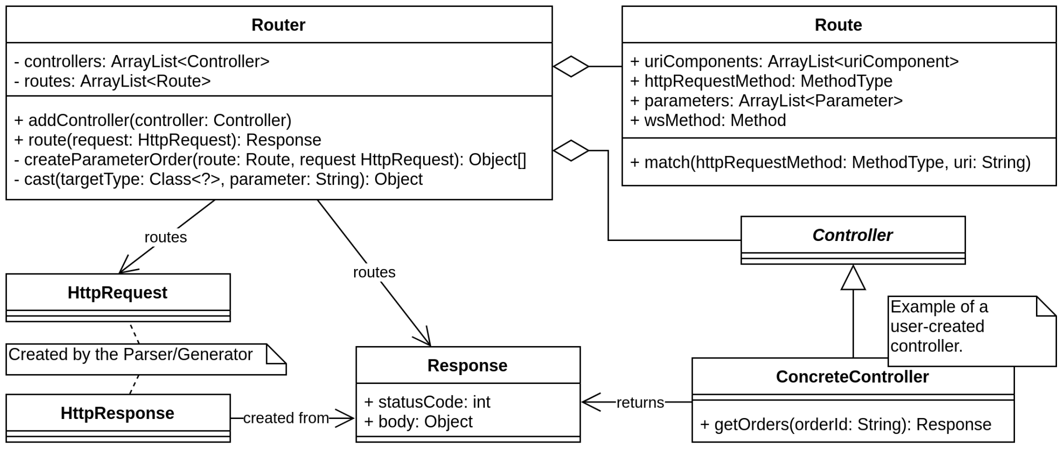

Router
- controllers: ArrayList<Controller>
- routes: ArrayList<Route>
+ addController(controller: Controller)
+ route(request: HttpRequest): Response
- createParameterOrder(route: Route, request HttpRequest): Object[]
- cast(targetType: Class<?>, parameter: String): Object
Route
+ uriComponents: ArrayList<uriComponent>
+ httpRequestMethod: MethodType
+ parameters: ArrayList<Parameter>
+ wsMethod: Method
+ match(httpRequestMethod: MethodType, uri: String)
routes
routes
Controller
HttpRequest
Created by the Parser/Generator
HttpResponse
created from
Response
+ statusCode: int
+ body: Object
returns
Example of a user-created controller.
ConcreteController
+ getOrders(orderId: String): Response

Fig. 6.2: A simplified diagram, outlining the interaction of various classes that provide the routing functionality. Some details were left out for clarity.

A framework user registers her controller (denoted as `ConcreteController` in fig. 6.2) using the `addController` method in the `Router` class. Upon registration, the `Router` uses the Java reflection APIs to validate each method in the controller. Each method must have exactly one annotation, specifying the HTTP request method and the URI, as outlined in the previous section. Additionally, the `Router` checks all parameters in the same manner. Each parameter must be annotated with the correct parameter type. For each method a new `Route` is created and stored in the `Router`. A `Route` contains all necessary information to later route incoming HTTP requests to the correct WS method. The URI and HTTP request method are extracted from the Java method annotation. The parameter annotations provide a parameter identifier and the type of the parameter (i.e., PathVariable, RequestParameter, or RequestBody). This information is later used to map incoming request data to the correct parameter. The provided URI is decomposed into its segments and stored as an array of `UriComponents`. Each variable segment is marked as such. Once the `Route` is fully assembled, it is added to the `Router`'s cache for routing.

Listing 6.4 shows the implementation of the routing procedure. The server passes incoming `HttpRequests` to the shown `route` method. This method searches all registered `Routes` for a match, maps the request data onto the expected WS method parameters, executes the WS method and returns the response to the server. Since all annotations from WS methods are already parsed and validated upon registration of the associated controller, finding the correct `Route` is simple and fast. Each route compares the provided request method type and the requested URI to its stored data (line 3,4). When comparing the URIs all variable segments are simply skipped (i.e., segments surrounded by { and }). A route matches a request if the request method type and all non-variable segments match.

Each route stores its respective WS method as a `Method` reference, which is a class that is part of the Java reflection API. The `Method` class offers the `invoke(Object obj, Object... ar` method. This executes the method on the object that is provides as the first first argument. It uses all following provided objects as arguments for that method execution. Line 7–12 show how this principle is applied in HoPME. First, the `createParameterOrder` method is used to extract the necessary information from the `httpRequest`, as an array of objects, sorted by their occurrence in the argument list for the executed method. This is done by iterating over the stored parameter list of the route object, and for each extracting necessary information from the request (e.g., the request body). Each argument generated this way, is then cast to the expected type (i.e., deserialized from a string representation) and stored as an `Object` in the correct position in an array. In order to serialize and deserialize objects to CBOR, the open-source library Jackson is used, with the official CBOR-backend implementation. The library is well integrated with Android and production-ready [24]. After extracting this argument list, the target WS method is invoked, in line 10–12, by passing the controller as a first argument and the obtained argument list as a second argument. The Java Virtual Machine then unwraps the array of objects and passes each as a new argument to the target method, in the order they appear. The resulting `Response` object, defined by the user is returned to the server for further processing.

```java
public Response route(HttpRequest httpRequest) {
    for (Route route : routes) {
        if(route.match(httpRequest.getMethodType(),
                       httpRequest.getUri())) {
            try {
                // call matching controller route
                Object[] parameters
                    = createParameterOrder(route, httpRequest);
                Object responseObject
                    = route.getMethod().invoke(
                            route.getController(),
                            parameters);

                // process response
                if (!(responseObject instanceof Response)) {
                    throw new ServerException("Invalid response.");
                }
                return (Response) responseObject;
            } catch (Exceptions...) {
                throw new ServerException("Invalid request.");
            }
        }
    }
    throw new RoutingException("Could not find route.");
}
```

Listing 6.4: The `route` method tries to match an incoming HTTP request to a registered route. It executes the registered method and returns the HTTP response.

7 Evaluation

This thesis is built on the belief that a peer-to-peer architecture improves the Taler system in mobile payment scenarios. The following section evaluates the proposed solution, and thereby the underlying assumption, based on the requirements outlined in chapter 3. The evaluation is split into two domains: the user experience and the security of the implementation. Improvements to the user experience must be evaluated to determine if the proposed system is an effective solution to the research question. However, because of the significant architectural changes in the underlying communication platform, the security of the new system must be evaluated as well, to identify potential negative trade-offs.

7.1 User Experience

The main goal of the peer-to-peer Taler payment system is to facilitate the merchant's installation process, and therefore to smoothen the onboarding experience. The following two scenarios showcase the change in user experience.

In this first scenario, a merchant sets up a full POS terminal installation using the traditional Taler architecture. First, the merchant downloads the Taler POS app [11] to her smartphone. At the first start of the app, it demands a URL to download the POS terminal configuration file which contains a list of the details of all offered products, and the location of the Taler merchant backend. To continue, the merchant must now install the Taler merchant backend and an accompanying SQL database on a server she hosts [25]. After installing and configuring the server, she can statically host the POS terminal configuration file and provide the URL to the POS app. The POS app is now fully functional. To complete this whole procedure, the merchant had to own and set up a server, install and configure a Taler backend server, and configure the network securely. Surely, this is a task for a system administrator, especially with regard to the criticality of the data this system handles in production.

The second scenario describes the same procedure with a modified version of the Taler POS app, containing the P2PTalerSDK. First, the merchant must download the Taler POS app. At the first start of the app, it prompts the merchant to enter the details of all offered products. After entering the details, the app is fully functional.

It is abundantly clear that the difference in installation complexity from the first to the second scenario is a significant reduction, all other things being equal. However, for the new merchant implementation to work, a paying customer must also run a modified Taler wallet app that enables peer-to-peer communication through ProxME. The user experience of the customer must therefore also be taken into consideration. Fortunately, the customer's usage pattern is not impacted by a peer-to-peer wallet: she scans a single QR code from the merchant, in both cases. As already outlined in chapter 4, the user's wallet is able to simultaneously support both the peer-to-peer architecture and the traditional Taler architecture. Therefore, changes to the underlying transport are hidden from the customer, when paying either at a peer-to-peer merchant, or a server-bound merchant. On a positive note for the customer, the peer-to-peer architecture partly solves the problem of customer–customer transactions as a side effect. Since the peer-to-peer system must be included in the wallet implementation anyways to support *sending* peer-to-peer payments, it can, in principle, additionally be used to *receive* peer-to-peer payments. Wallet apps could implement the needed user interfaces for receiving private Taler transactions from other customers. This practice was originally suggested as a topic of further research in [5]. It is beyond the scope of this thesis to discuss the details of the scheme and additional arrangements needed.

All in all, it can be said that the peer-to-peer system accomplishes its main goal without major trade-offs. P2PTalerSDK provides a good solution to the usability problem it aims to solve by heavily reducing the installation complexity of the merchant. Additionally, it offers an indication for further usability improvements on the customer-side.

7.2 Security

In order to assess the security properties of the proposed system, the security requirements described in section 3.2 are grouped into categories by topic. The proposed system is analyzed with respect to each category.

The first group of security requirements is concerned with the architectural integrity of the proposed software system. Security Requirement 1–5, and 15 deal with the design and functioning of software components and their relation to each other. The architecture (chapter 4, as well as the individual components (chapter 5 and 6) are described in detail — especially with regard to security-critical components. The thesis at hand provides a design document for the proposed system and security features are a primary component of its design principles. All requirements that are part of this group are met by the design and documentation process in this thesis.

The second group of security requirements describes the correct use of cryptographic material and algorithms. The following requirements are members of this group: Security Requirements 7–11, 14, and 16. Even though cryptography plays a major role in this thesis, its application is deliberately contained in a limited, well-defined module: the ProxME protocol (see section 5.3.3 and 5.3.4). All other components rely on this module to exchange critical information securely, but must not be assessed in terms of compliance with cryptographic requirements, because of the strict separation of concerns. This principle facilitates the following evaluation of security requirements greatly. The ProxME protocol implementation uses Sodium, an open-source, third-party audited cryptographic library that defaults to modern best practice cryptographic primitives and parameters. This ensures that the protocol is based on a solid foundation and is not susceptible to known attacks against the used algorithms and primitives. The protocol specification outlines clearly when and how to use, and when do discard key material. The protocol implementation — a state machine — guarantees that keys are never accidentally reused by resetting all cryptographic state on an error and on restart of the protocol. All requirements in this group are therefore met due to the careful separation of concerns, adherence to cryptographic best practices and detailed specification of cryptographic protocols. Future implementations of the Taler merchant backend logic and Taler wallet logic that may be integrated into P2PTalerSDK, very likely contain cryptography and must then be reevaluated by the latest security standards.

The last group of security requirements is concerned with access control and safeguarding private data. Requirements in this group are Security Requirement 6, 12, 13 and 17. Taler's unique approach to payment processing, in which payments do not produce customer data, greatly facilitates the assessment of requirements in this group. The P2PTalerSDK does not process personally identifiable information of customers, and therefore no customer data is stored in the process. This ensures that the proposed system is in accordance with the fundamental ideas of most privacy regulations. The proposed shift from a server-centric architecture to an app-centric architecture simplifies the concept of access control. In the traditional Taler installation, a merchant must monitor and manage access for employees, whereas with an app the concept of access control could be reduced to physically having access to the smartphone. The requirements of this final category are also entirely met, through the inherited privacy guarantees from Taler and the user-empowering peer-to-peer design.

The proposed system was designed and implemented with the latest security best practices and guidelines in mind. All developed security requirements are met by the proposed system. This suggests that P2PTalerSDK is based on a solid security foundation and is a good candidate for further research.

7.3 Results

The P2PTalerSDK attempts to solve the problem of improving merchant adoption rates of GNU Taler. The preceding evaluation unambiguously illustrates that it is a good candidate solution for this problem, as it removes an enormous barrier to entry for potential merchants. The simplified installation and maintenance leads to a reduction of costs for both, merchants that contemplate trying out Taler, and for merchants that run Taler productively in their shop. This reduction of costs mainly materializes through reduced costs in human resources, but also from the difference in initial acquisition costs (i.e., server vs. smartphone), and maintenance costs (i.e., electricity and depreciation). Additionally, the architectural changes do not degrade Taler's privacy guarantees and provide a security level that is in accordance with current digital payment security standards. All in all, it is very likely that the proposed system would lead to higher adoption rates by merchants.

8 Conclusion

8.1 Contributions

The thesis at hand provides an alternative to Taler's server-based mobile payments architecture, in order to offer a solution to the research problem of facilitating the adoption of the Taler technology.

The proposed architecture removes the need for merchants to operate a backend server by setting up a peer-to-peer connection between the customer's and the merchant's mobile device. This new architecture is implemented by providing P2PTalerSDK, a mobile SDK that wraps the merchant's backend server and runs it on her smartphone. While not reimplementing the Taler backend itself, this thesis contributes the underlying network technology stack that enables a peer-to-peer exchange of Taler coins.

First, this thesis develops a new library, ProxME, that handles discovery, connection, and secure communication of mobile devices in close proximity. ProxME, discovers the correct communication partner through scanning of a QR code. To find peers in close proximity, the prototype implements an automatic wireless advertising and scanning technique, using Bluetooth Low Energy. Once two ProxME devices have discovered each other, they use state-of-the-art cryptography to set up an encrypted and authenticated wireless connection. Using this connection, both participants can initiate communication and exchange messages securely.

Second, to make the integration of existing Taler backend code as easy as possible, this thesis provides another library, HoPME, that uses ProxME to exchange HTTP messages with close-proximity peers. HoPME acts as a regular web service framework for the implementation of the SDK's backend server, but tunnels all communication securely via the underlying ProxME peer-to-peer connection. HoPME features an HTTP parser and generator that handles incoming and outgoing HTTP messages transparently. Internally, web service endpoints must not care about the underlying protocol. HTTP requests are automatically routed by the HoPME framework, allowing for rapid development of HTTP endpoints.

Finally, the evaluation of the proposed system reveals that it is a very strong candidate solution for the research problem of this thesis. The introduced architecture removes a great barrier to entry for merchants and the evaluation results point, therefore, towards higher adoption rates.

8.2 Future Work

This thesis provides only a partial implementation of the proposed architecture. In order to develop a fully functioning prototype, the following components must be investigated in further research. First and foremost, the Taler backend must be ported to the P2PTalerSDK. Since the backend is written in C [26], the Java native interface API [27] can be used to run the original implementation on the smartphone. A Java native interface wrapper must be written for each of the backend server's public endpoints. These wrappers can then be called by a thin HoPME implementation for each web service. Alternatively, the backend could be reimplemented in Java. In both cases, HoPME facilitates development by providing a simple framework to implement the required web service endpoints. Second, HoPME should ideally provide a client-side framework to easily create HTTP requests over ProxME. Currently, in order to communicate with a merchant, the customer's peer-to-peer wallet implementation must craft HTTP messages and manually send them over ProxME. This makes a port of the Taler wallet cumbersome. Third, ProxME is designed to be easily extensible with new transport channels. However, the current prototype only supports Bluetooth LE. To cover a wide range of supported devices more transport channel implementations are needed.

In addition to developing new components, there are interesting research problems in the field of improving the existing architecture. Firstly, a formal verification-proof of correctness of ProxME's key exchange protocol, described in section 5.2, has yet to be provided. This is an important contribution that would strengthen the costumer's trust and confidence in the system. And secondly, in order to make the prototype implementation of ProxME and HoPME ready for operation in a production environment, a testing methodology must be established and implemented for the existing codebase. Especially unit tests for the cryptographic implementations are recommended.

Bibliography

[1] Prof Dr Nikolas Beutin and Maximilian Harmsen. (2019). “Mobile payment report 2019,” [Online]. Available: `https://www.pwc.de/de/digitale-transformation/pwc-studie-mobile-payment-2019.pdf` (visited on Oct. 8, 2020) (cit. on p. 1).

[2] Statista Inc. (2020). “Mobile POS payments - germany,” [Online]. Available: `https://www.statista.com/outlook/331/137/mobile-pos-payments/germany` (visited on Nov. 8, 2020) (cit. on p. 1).

[3] F. Dold, “The GNU taler system: Practical and provably secure electronic payments,” Ph.D. dissertation, University of Rennes 1, 2019 (cit. on pp. 1, 5, 7).

[4] Taler Systems SA. (2020). “GNU taler project,” [Online]. Available: `https://taler.net/en/index.html` (visited on Oct. 21, 2020) (cit. on p. 5).

[5] J. Burdges, F. Dold, C. Grothoff, and M. Stanisci, “Enabling secure web payments with GNU Taler,” in *Security, Privacy, and Applied Cryptography Engineering*, 6th International Conference, SPACE2016 (Dec. 14, 2016), C. Carlet, M. A. Hasan, and V. Saraswat, Eds., ser. Lecture Notes in Computer Science, vol. 10076, Hyderabad, India: Springer, Cham, 2016 (cit. on pp. 5, 6, 8, 9, 44).

[6] D. Chaum, “Blind signatures for untraceable payments,” in *Advances in Cryptology Proceedings of Crypto 82*, R. Rivest and A. Sherman, Eds. Plenum, 1983, pp. 199–203 (cit. on p. 6).

[7] Taler Systems SA. (2020). “Gnu taler: Design principles,” [Online]. Available: `https://taler.net/en/principles.html` (visited on Aug. 26, 2020) (cit. on p. 12).

[8] Free Software Foundation. (2020). “What is free software?” [Online]. Available: `https://www.gnu.org/philosophy/free-sw.html` (visited on Nov. 8, 2020) (cit. on p. 12).

[9] OWASP Foundation, *The mobile application security verification standard*, 2020 (cit. on p. 13).

[10] PCI Security Standards Council, LLC, “Payment card industry (pci) data security standard,” Standard 3.2.1, 2018 (cit. on p. 14).

[11] Taler Systems SA. (2020). “Gnu taler merchant pos manual,” [Online]. Available: `https://docs.taler.net/taler-merchant-pos-terminal.html` (visited on Nov. 8, 2020) (cit. on pp. 16, 43).

[12] R. Fielding, J. Gettys, J. Mogul, *et al.*, “Hypertext transfer protocol – HTTP/1.1,” RFC Editor, RFC 2616, Jun. 1999 (cit. on pp. 17, 35, 37, 38).

[13] Bluetooth SIG Inc., “Bluetooth core specification,” Specification 5.2, 2019 (cit. on pp. 19, 27, 28).

[14] M. Zeppelzauer and A. Ringot. (2019). “Sonitalk: An open protocol for data-over-sound communication,” [Online]. Available: `https://sonitalk.fhstp.ac.at/wp-content/uploads/documentation/draft-zeppelzauer-data-over-sound-00.pdf` (visited on Nov. 8, 2020) (cit. on pp. 19, 21).

[15] Android Open Source Project. (2020). “Bluetooth low energy overview,” [Online]. Available: `https://developer.android.com/guide/topics/connectivity/bluetooth-le` (visited on Oct. 14, 2020) (cit. on pp. 21, 27, 29).

[16] F. Denis. (2020). “Libsodium - introduction,” [Online]. Available: `https://doc.libsodium.org/` (visited on Aug. 26, 2020) (cit. on p. 21).

[17] ——, (2020). “Libsodium - sealed boxes,” [Online]. Available: `https://doc.libsodium.org/public-key_cryptography/sealed_boxes` (visited on Aug. 26, 2020) (cit. on p. 21).

[18] ——, (2020). “Libsodium - encrypted streams and file encryption,” [Online]. Available: `https://doc.libsodium.org/secret-key_cryptography/secretstream` (visited on Aug. 26, 2020) (cit. on pp. 22, 34).

[19] Apple Inc. (2020). “Core bluetooth,” [Online]. Available: `https://developer.apple.com/documentation/corebluetooth` (visited on Nov. 8, 2020) (cit. on p. 27).

[20] Bluetooth SIG Inc., “Bluetooth core specification,” Specification 5.2, 2019, p. 285 (cit. on p. 28).

[21] ——, “Bluetooth core specification,” Specification 5.2, 2019, p. 1475 (cit. on p. 28).

[22] D. Booth, H. Haas, F. McCabe, *et al.* (2020). “Web services architecture,” [Online]. Available: `https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/` (visited on Nov. 8, 2020) (cit. on p. 37).

[23] C. Bormann and P. Hoffman, “Concise binary object representation (cbor),” RFC Editor, RFC 7049, Oct. 2013 (cit. on p. 37).

[24] FasterXML, LLC. (2020). “Jackson core: Streaming,” [Online]. Available: `https://github.com/FasterXML/jackson-core/wiki` (visited on Nov. 8, 2020) (cit. on p. 41).

[25] Taler Systems SA. (2020). “GNU taler merchant backend operator manual,” [Online]. Available: `https://docs.taler.net/taler-merchant-manual.html` (visited on Nov. 8, 2020) (cit. on p. 43).

[26] ——, (2020). “Merchant.git,” [Online]. Available: `https://git.taler.net/merchant.git/tree/` (visited on Nov. 8, 2020) (cit. on p. 48).

[27] Oracle. (2020). “Java native interface specification,” [Online]. Available: `https://docs.oracle.com/en/java/javase/14/docs/specs/jni/index.html` (visited on Nov. 8, 2020) (cit. on p. 48).

List of Figures

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, November 10, 2020

J. Florian Kimmes