




**HOCHSCHULE
MITTWEIDA**
University of Applied Sciences

MASTER THESIS

Mr.
Tim Käbisch

**Building a trustless connection
between the Lightning Network and
EVM-compatible blockchains**

Mittweida, May 2023



Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Building a trustless connection between the Lightning Network and EVM-compatible blockchains

Author:

Tim Käbisch

Course of Study:

Blockchain & Distributed Ledger Technologies

Seminar Group:

BC20w1-M

First Examiner:

Prof. Dr.-Ing. Andreas Ittner

Second Examiner:

M.Sc. Erik Neumann

Submission:

Mittweida, 14/05/2023

Defense/Evaluation:

Mittweida, 2023

Bibliographic Description

Käbisch, Tim: Building a trustless connection between the Lightning Network and EVM-compatible blockchains, 51 pages, Mittweida, Hochschule Mittweida – University of Applied Sciences, Faculty of Applied Computer Sciences and Biosciences.

Master Thesis, 2023

Abstract

As the cryptocurrency ecosystem rapidly grows, interoperability has become increasingly crucial, enabling assets and data to interact seamlessly across multiple chains. This work describes the concept and implementation of a trustless connection between the Bitcoin Lightning Network and EVM-compatible blockchains, allowing the transfer of assets between the two ecosystems. Establishing such a connection can significantly contribute to the growth of both ecosystems as they can benefit from each other's advantages and emerge new possibilities.

Contents

Contents	I
List of Figures	III
Listings	IV
1 Introduction	1
2 Fundamentals	3
2.1 EVM-compatible Blockchains	3
2.1.1 EVM	3
2.1.2 Gas	4
2.1.3 Nonce	5
2.2 Lightning Network	6
2.2.1 Scaling Bitcoin	7
2.2.2 Invoices	8
2.2.3 WebLN	10
3 Concept	13
3.1 Pre-signed Transaction	13
3.1.1 Protocol	13
3.1.2 Problems	16
3.1.3 Evaluation	19
3.2 Atomic Swaps	19
3.2.1 HTLC	19
3.2.2 Protocol	20
3.2.3 Evaluation	23
4 Implementation	25
4.1 HTLC in Solidity	26
4.2 Lightning - Native Coin	31
4.3 Lightning - ERC-20 Token	35
4.4 Native Coin - Lightning	38
4.5 ERC-20 Token - Lightning	41
5 Further Considerations	44
5.1 Faucet	44
5.2 Taro	45
5.3 Supported Wallets	46
5.4 ERC-721	49
6 Conclusion	51
A Implementation	52
Bibliography	53

List of Figures

2.1	Lightning Network Capacity [13]	8
2.2	WebLN Provider Functionality [18]	10
2.3	Alby Approve Payment and Create Invoice	12
3.1	Protocol: Pre-signed Transaction	14
3.2	Protocol: Atomic Swap	21
3.3	Locking and Unlocking HTLC	22
4.1	User Interface	25
4.2	Protocol: Lightning - Native Coin	32
4.3	User Interface: Offer Buy Coin	34
4.4	User Interface: Claim Coin	35
4.5	Protocol: Lightning - ERC-20 Token	36
4.6	User Interface: Buy ERC-20 Token	37
4.7	Protocol: Native Coin - Lightning	38
4.8	User Interface: Offer Sell Coin	39
4.9	User Interface: Waiting for Operator	40
4.10	Protocol: ERC-20 Token - Lightning	41
4.11	User Interface: Approve ERC-20 Token	42
5.1	User Interface: Faucet	45
5.2	User Interface: Wallet Connect [33]	47
5.3	User Interface: Display Lightning Invoice	48

Listings

2.1 LNbits Invoice Creation	9
2.2 WebLN sendPayment	11
2.3 WebLN makeInvoice	11
3.1 Smart Contract transferETH Function	14
3.2 Create Pre-signed Transaction	15
4.1 HTLC Struct + Mapping	26
4.2 HTLC haveContract	27
4.3 HTLC newContract	27
4.4 HTLC getContract	28
4.5 HTLC withdraw	29
4.6 HTLC refund	30
5.1 HTLC Struct for ERC-721	50

1 Introduction

The cryptocurrency ecosystem has grown significantly and evolved in recent years, with numerous new ideas and blockchain platforms emerging. However, two ecosystems continue to stand out as the pillars of the entire ecosystem: Ethereum and Bitcoin. This work aims to build a trustless connection enabling asset transfer between the two ecosystems.

The introduction of smart contracts by the Ethereum network [1] revolutionized the blockchain industry, enabling the development of various decentralized applications (dApps) across different domains, such as insurance, decentralized finance (DeFi), social platforms, and games. However, scalability issues have remained a persistent challenge for the Ethereum network, leading to the emergence of several other blockchain ecosystems, such as Binance Smart Chain, Polygon, and Avalanche [2]. Despite their differences, these ecosystems share the common feature of utilizing the Ethereum Virtual Machine (EVM) to change the state of their networks, enabling them to be compatible with the broader Ethereum ecosystem and facilitating interoperability. [3]

The second pillar, Bitcoin [4], is a widely-used global system that maintains a record of transactions on a publicly available ledger. The system's scalability faces challenges as each participating computer is responsible for validating, observing, and storing every transaction. In response to this scalability issue, Joseph Poon and Thaddeus Dryja proposed a solution in their paper *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments* [5]. The Lightning Network is a second layer on top of Bitcoin. It introduces off-chain payment channels enabling users to transact even the smallest amounts, i.e., micropayments, without publishing them to the Bitcoin blockchain. The concept was initially proposed in 2015, followed by the first implementation of the protocol in 2018. Despite being in its early stages, the Lightning Network has already drawn significant attention from developers and investors, with many applications currently under development. [6, p. 9-12]

As the cryptocurrency ecosystem continues to grow and evolve, the need for interoperability has become increasingly crucial. Interoperability in the context of blockchain technology refers to the ability of assets and data to interact across multiple chains seamlessly. When two parties use the same blockchain platform, such as Bitcoin, exchanging data and value is straightforward. However, this becomes more challenging when the parties use different blockchain platforms. EVM-compatible blockchains and the Bitcoin Lightning Network have unique advantages in their respective use cases. Enabling interoperability between the two could open up new possibilities and use cases, making it a significant area of research and development. [7]

There have been previous attempts to establish a trustless connection between the Lightning Network and EVM-compatible blockchains, as evidenced by various prototypes available on GitHub [8, 9]. However, most of these prototypes require the user to run a separate Lightning node, making them inaccessible to many. This work aims to connect the two ecosystems

while prioritizing security, speed, and user experience. Establishing such a connection can contribute to the growth of both ecosystems as they can benefit from each other's advantages and emerge new possibilities.

This work is organized into six chapters. In this chapter, the context and motivation of this work were introduced. Chapter two provides an overview of the necessary fundamentals of EVM-compatible blockchains and the Bitcoin Lightning Network. Based on these fundamentals, chapter three presents two concepts for achieving a trustless connection between these ecosystems. The implementation of the more promising concept, namely so-called atomic swaps, is presented in chapter four. Chapter five provides additional considerations for the protocol and its implementation, primarily focusing on potential extensions. Finally, chapter six concludes this work with a summary and an outlook on future development.

2 Fundamentals

The following chapter deals with the necessary fundamentals of EVM-compatible blockchains and the Lightning Network.

2.1 EVM-compatible Blockchains

The Ethereum network was the first to introduce the concept of smart contracts, revolutionizing the blockchain industry. However, the network has struggled with scalability issues, leading to the emergence of other blockchain ecosystems, such as Binance Smart Chain, Polygon, and Avalanche. Despite their differences, these ecosystems have one thing in common: they remain compatible with Ethereum by utilizing the Ethereum Virtual Machine (EVM) to change the state of their networks. This allows developers to write smart contracts that can run on multiple blockchains, creating a more connected and interoperable ecosystem. As the demand for blockchain technology continues to grow, the importance of interoperability and compatibility with Ethereum will only continue to increase. [3]

2.1.1 EVM

In the context of software development, programmers typically write code in high-level programming languages, such as Java, Python, or C++. While this code is understandable to humans, a computer's central processing unit (CPU) cannot execute it directly. Therefore, the code must be translated into machine-executable code, which is achieved through a process known as compilation. A compiler is a software program that translates the code written in a high-level programming language into bytecode, which is a low-level, machine-readable format. Once compiled, the CPU can execute the bytecode, allowing the computer to run the program as intended. Compiling is fundamental to software development, enabling developers to write code in a human-readable and machine-executable format. [3]

Blockchain networks, such as Ethereum, are decentralized networks that run on nodes distributed worldwide. This distributed architecture makes the so-called "world computer" unique compared to traditional computing systems, as it does not rely on a single central processing unit (CPU) to execute programs. Instead, the Ethereum network leverages the Ethereum Virtual Machine (EVM) as a software-based CPU to execute bytecode on each network node. Therefore, allowing developers to write smart contracts in high-level programming languages, such as Solidity, and compile them into bytecode that can be executed by the EVM. This approach to executing code enables the Ethereum network to operate without a central control point. [3]

Deploying a smart contract on a blockchain network results in a copy being distributed to every node in the network. Users can submit transactions to modify the contract's global state, and each node executes the transaction within its EVM and saves the resulting output, reflecting the changed state. This mechanism creates a distributed state machine where each

node maintains a copy of the global state. The EVM functions as the core component of the distributed state machine, orchestrating the execution of transactions to ensure the state remains consistent across all nodes in the network. [10, p. 297]

With the significant growth in the number of users on the Ethereum network, the cost of using the network has become prohibitively expensive. To address this issue, sidechains have emerged, typically code forks of Ethereum with different consensus mechanisms [11]. Sidechains offer faster and cheaper transaction processing, enabling developers to build decentralized applications that are more cost-effective for users. To ensure compatibility with Ethereum, sidechains leverage the Ethereum Virtual Machine (EVM) to change the state of their networks, enabling developers to reuse existing solutions and protocols rather than building them from scratch. Additionally, users are already familiar with the workings of the EVM, allowing them to utilize their preferred applications across different chains. [10, p. 297]

2.1.2 Gas

Gas is a critical component of EVM-based blockchains, providing a mechanism for measuring the computational and storage resources required to perform actions on the network. Unlike Bitcoin, whose transaction fees only account for the size of a transaction in kilobytes, the open-ended computation model in networks like Ethereum requires more extensive metering to avoid denial-of-service attacks or inadvertent resource-devouring transactions. Gas controls the number of resources a transaction can use, preventing them from consuming more than intended or necessary. This disincentivizes attackers from sending spam transactions, as they must pay for every computational step. [10, p. 106]

Gas is a separate virtual currency besides, e.g., ether on Ethereum. This separation is necessary to protect the system from the volatility that may arise from rapid changes in ether's value and to manage the sensitive ratios between the costs of various resources that gas pays for, including computation, memory, and storage. Each operation performed by a transaction or smart contract code execution costs a fixed amount of gas, which results in a variation of gas costs per transaction, depending on the complexity of the operations. [10, p. 106]

Essentially, four terms have to be taken into account when dealing with gas [10, p. 100]:

- Gas Cost: The amount of gas required to execute a transaction.
- Gas Price: The amount of ether the sender is willing to pay for each gas unit.
- Gas Limit: The max. amount of gas that the sender is willing to pay for a transaction.
- Block Gas Limit: The max. amount of gas that can be used in a block.

The gas cost of a transaction is determined by the number of computational steps required to execute the transaction. For example, simple payments that transfer ether between two accounts cost 21,000 gas. The gas price is the amount of ether the sender is willing to pay for each gas unit. The current network conditions determine a suitable gas price. During periods of high demand, the price of gas may increase, while during periods of low demand, the price of gas may even be zero, i.e., a transaction at zero cost. The gas limit is the maximum amount of gas that the sender is willing to pay for a transaction. If the transaction consumes more

gas than the gas limit, the transaction will fail and revert. This might happen if the gas limit is too low or the transaction is more complex than expected. In such cases, the sender will still be charged because the computational work already occurred. The block gas limit is the maximum amount of gas that can be used in a block. If the block gas limit is reached, the transactions not included in the block will be queued and included in one of the following blocks. [10, p. 314-317]

Understanding the concept of gas in EVM-compatible blockchains is essential for the following two things: First, it is crucial to understand the gas costs of transactions, as they are the main cost factor for using the network. For a service that programmatically creates transactions, it is essential to estimate the costs of transactions. This would allow the service to pass on the transaction fees incurred to its customers, for example. The web3.js library allows to estimate a transaction's gas costs and retrieve the current gas price. Due to that, a programmatic estimation of the gas costs is enabled. [10, p. 314-317]

Second, gas plays a crucial role in the confirmation speed of a transaction. Usually, transactions with a higher fee are included in a block faster (if the miners behave in a rational economic manner). This is especially important for time-critical transactions. For example, if a user sends a transaction to a smart contract to execute a specific action at a particular time. In that case, the transaction must be included in a block before the time is reached. Otherwise, the action will not be executed. In this case, the user has to pay a higher fee to ensure that the transaction is included in a block before the time is reached. [10, p. 314-317]

2.1.3 Nonce

The term nonce stands for *number used once*. In general, it is a number only used once for a specific purpose. A distinction must be made between two nonces: One used as part of the proof-of-work (PoW) consensus algorithm and one used for transactions in EVM-compatible blockchains. In the PoW consensus algorithm, miners repeatedly change the nonce and recalculate the hash value to find a value that meets specific criteria [12]. The nonce in the context of this work always refers to the nonce used for transactions in EVM-compatible blockchains.

A transaction nonce is a value generated dynamically "by counting the number of confirmed transactions that have originated from an address" [10, p. 101]. It is used to ensure that transactions sent from a specific address are unique and processed in the correct order. Although the nonce is an attribute of the sender's address, it is not explicitly stored in the blockchain's state of accounts. The transaction nonce is critical for an account-based blockchain protocol, in contrast to the Unspent Transaction Output (UTXO) mechanism used by the Bitcoin protocol. [10, p. 101]

The influence of the nonce on the correct transaction sequence can be seen in the following scenario: A user attempts to make two transactions in the Ethereum network, one for six ether and another for eight ether, but has only ten ether in their account. The user broadcasts the 6-ether transaction first, expecting it to be processed before the 8-ether transaction. However,

due to the decentralized nature of the Ethereum network, some nodes may receive the 8-ether transaction first, potentially causing the 6-ether transaction to fail. The transaction nonce ensures that the transactions are processed in the correct order, based on their nonce values, even if they arrive out of order to different nodes on the network. By including a nonce in the transactions, the first transaction sent will have a nonce value lower than the second transaction and, thus, will be processed first. [10, p. 101]

The nonce also plays a crucial role in preventing so-called replay attacks. A replay attack is when an attacker attempts to use a transaction the network has already processed. The following scenario illustrates how the nonce prevents such attacks: A user has an account with 100 ether and sends two ether to another account to purchase a product. Without a nonce value in the transaction, a malicious actor could copy and replay the transaction multiple times, potentially resulting in the loss of all the user's ether. However, by including a nonce value in the transaction data, every transaction is unique, even when sending the same amount of cryptocurrency to the same recipient address multiple times. This ensures that no one can duplicate a payment made by the user. [10, p. 101]

This results in two essential consequences that are relevant to the concept of this work: First, transactions are processed based on their nonce value, leading to sequential processing of transactions. If a transaction with a higher nonce value is broadcast before one with a lower nonce value, the latter is held in the mempool until the missing nonce is received. To fill the gap, a valid transaction with the missing nonce should be broadcast, which allows previously held transactions to be included. Transactions can unintentionally create gaps in the nonce sequence due to invalidity or insufficient gas, causing subsequent transactions to be held in the mempool until a valid transaction with the missing nonce is broadcast. Once a transaction with the missing nonce is validated, the transactions with higher nonces become valid. [10, p. 104]

Second, in case of a nonce duplication, such as transmitting two transactions with the same nonce but different transaction data, one of the transactions will be confirmed while the other will be rejected. The confirmation of the transaction will be determined by the sequence in which they arrive at the first validating node that receives them, resulting in a random selection of which transaction is confirmed. These consequences are discussed in more detail in section 3.1.2. [10, p. 104]

2.2 Lightning Network

The Lightning Network is a transformative protocol revolutionizing how individuals exchange value online. It operates as a second-layer technology on top of Bitcoin, facilitating fast, cheap, and scalable transactions between users. The Lightning Network uses Bitcoin in a novel way that enables micropayments, making it possible to transact even the smallest of amounts. The concept was originally introduced in 2015, and the first implementation of this protocol was released in 2018. Despite still being in its infancy, the Lightning Network has attracted significant attention from developers and investors, and many applications are being built today. [6, p. 1]

2.2.1 Scaling Bitcoin

Bitcoin is a widely-used global system that keeps a record of transactions on a publicly available ledger. Each transaction is validated, observed, and stored by each participating computer, posing challenges for scalability. The rise in popularity of Bitcoin and the demand for transactions have led to an increased number of transactions in each block. This resulted in reaching the block size limit regularly. Extra transactions are left to queue when blocks are full, and the fee competition increases. Consequently, lower-value transactions may become unprofitable to include during times of high demand. The block size limit could be raised to address this problem, providing additional space for transactions. Nevertheless, this would shift the expenses to node operators, necessitating increased resources to validate and save the blockchain. The larger the block size, the more critical the bandwidth, processing, and storage requirements for the network participants. As a result, the system becomes more centralized since fewer people can operate a node. [6, p. 9]

In terms of scalability, Bitcoin is often compared to Visa, which processes around 40,000 transactions per second at peak usage [6, p. 9]. For Bitcoin to process as many transactions as Visa, a block size limit of around eight gigabytes would be required. With a block being mined every 10 minutes on average, this results in more than one terabyte of transaction data daily. Operating a node could not be done by everyday people at home anymore but would exclusively be feasible for large companies with the necessary resources. Additionally, the capacity to process 40,000 transactions per second is only equivalent to the capacity of traditional financial payment networks. However, it may not be sufficient to meet future demands due to the emergence of new technologies such as microtransactions and machine-to-machine payments. [6, p. 9-10]

The paper *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments* by Joseph Poon and Thaddeus Dryja proposes a solution for Bitcoin's scalability problem. The proposed Lightning Network is a second layer on top of Bitcoin, introducing off-chain payment channels which allow users to make transactions without publishing them to the Bitcoin blockchain. This enables users to make as many payments as they want without waiting for confirmations or paying transaction fees. The payments are instead conducted off-chain in payment channels, which can be created and closed with a transaction on the Bitcoin blockchain. During the lifespan of a payment channel, users can make unlimited payments, subject only to the bandwidth and computing power available to them. Furthermore, payments can also be conducted between users that do not operate a payment channel with each other. This can be achieved by routing payments through the Lightning Network using other participants' payment channels. [6, p. 10-12]

Payment channels on the Bitcoin Lightning Network are implemented as 2-of-2 multi-signature addresses, which are a type of Bitcoin address that requires two signatures to authorize a transaction. The multi-signature address is funded with the opening transaction of the payment channel, and both participants hold one key of the multi-signature address. Payments on the Lightning Network are executed by signing a transaction that spends from the multi-signature address. However, the transaction is not broadcast to the Bitcoin blockchain but is kept by the channel partners. This allows the channel partners to conduct a series of payments, negotiating the balance of the channel. The latest transaction in the series defines

how the balance is split between both participants. Therefore, making a payment on the Lightning Network is equivalent to updating the balance of the payment channel by moving funds from one participant to the other. Closing the channel is done by broadcasting the latest transaction to the Bitcoin blockchain, which is then included in a block, and the funds are transferred to the participants accordingly. [6, p. 40]

The Lightning Network has experienced significant growth since its release in 2018. One significant indicator of this growth is the steady increase in cumulative Bitcoin capacity across all channels of the Lightning Network. This trend can be observed in figure 2.1, where the red line represents the total capacity of Bitcoin, while the blue line shows the capacity of the Lightning Network in USD. Bitcoin's all-time high at the end of 2021 also led to a record-high capacity in the Lightning Network in USD. At the time of writing, the Lightning Network has a capacity of nearly 5,500 Bitcoin, valued at approximately 120 million USD. [13]

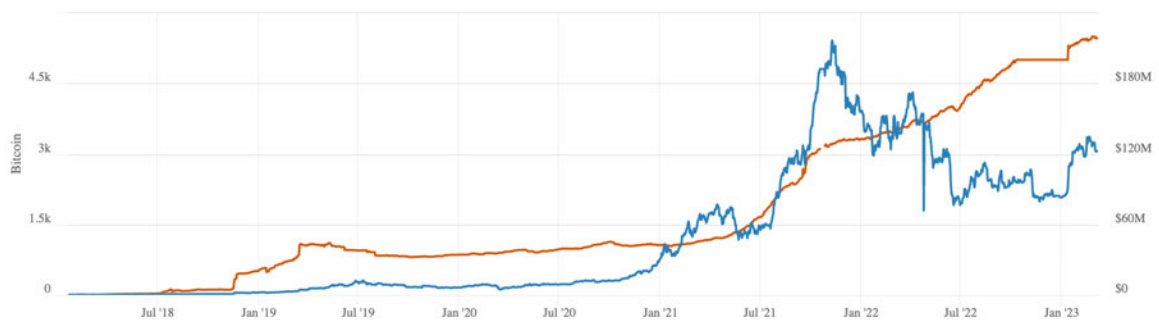


Figure 2.1: Lightning Network Capacity [13]

The Lightning Network's ability to facilitate instant payments with minimal fees has drawn the attention of developers interested in building applications on top of the network, known as Lightning Applications (LApps). One important use case for LApps would be pay-per-use, allowing users to pay only for the specific service they utilize rather than being charged for a subscription plan. This model could benefit newspaper subscriptions, where the user only pays for the articles they read, or for music streaming services, where they only pay for the songs they listen to. In addition, lightning payments could be used for everyday goods, such as coffee at a restaurant or snacks from a vending machine. Compared to traditional payment methods such as cash or credit cards, lightning payments offer faster and cheaper transactions, which could benefit both consumers and providers. [14, 15]

2.2.2 Invoices

Bitcoin transactions can be sent to the receiver's Bitcoin address, which only the owner of the corresponding private key can access. Furthermore, a Bitcoin address can be used an unlimited number of times. In contrast, the Lightning Network primarily utilizes invoices to initiate payments. Such invoices can only be used once since the recipient must reveal a secret to complete the payment. Once the secret is revealed, it should not be used again to prevent other participants from potentially stealing the funds. Payments on the Lightning Network are always atomic, meaning either the payment is completed or not, without any in-between state where the payment is partially successful. After being created by the recipient, an invoice can be transmitted to the payer using any communication channel, such as email, chat, or even a physical paper. [6, p. 55, 336]

An invoice on the Lightning Network contains the following information, among other things:

- Amount of the payment
- Payment hash
- Description of the payment
- Expiry time

The payment hash is the most critical element of an invoice, as it serves as the payment identifier and the key to completing the payment. The payment hash is constructed through two steps: firstly, a random number called the preimage or payment secret is selected. Secondly, the preimage is hashed using the SHA-256 algorithm to generate the payment hash. This payment hash is used to create the invoice, and only the creator of the invoice has access to the preimage, as there is no known way to reverse the SHA256-algorithm. To complete the payment, the preimage needs to be publicly revealed. Therefore, if the payer knows the preimage, the payment is completed, making the preimage a proof of payment. In other words, the payer can prove that the payment was successful when they can present a preimage that hashed with the SHA256-algorithm matches the payment hash. This fundamental concept is significant for the concepts discussed in this work and will be further elaborated in chapter 3. [6, p. 56]

The programmatic management of invoices can be done with different providers. One such provider is LNbits, which provides an application programming interface (API) for creating and paying invoices. The API can be called with HTTP requests, and authentication is achieved with an API key. The programmatic creation of an invoice can be done with a few lines of JavaScript and can be seen in code snippet 2.1. The programmatic payment of an invoice is as simple as slightly changing the body. [16]

```
1 let body = {
2   "out": false,
3   "amount": 1000, // 1000 satoshis
4   "memo": "This in an invoice."
5 }
6
7 let invoice = await fetch('https://lnbits.com/api/v1/payments', {
8   method: 'POST',
9   headers: {
10    'X-API-Key': '5cb01893ed699e' // API key
11  },
12   body: JSON.stringify(body)
13 }).then(res => res.json())
14
15 let paymentHash = invoice.payment_hash // 32 bytes hex string
16 let paymentRequest = invoice.payment_request // lnbc10u1pjq34...
```

Listing 2.1: LNbits Invoice Creation

2.2.3 WebLN

WebLN is a collection of guidelines intended for Lightning apps and client providers, designed to enable secure communication between web apps and users' Lightning nodes. The programmatic interface offered by WebLN enables applications to request payments from users, generate invoices to receive payments, and perform other related functionalities. The WebLN specifications, first published in 2018, are based on the web3.js concept, which allows developers to interact with a local or remote Ethereum node using HTTP, IPC, or WebSocket protocols [17]. Over time, several Lightning applications and client providers have been built using this standard. The way WebLN fits into the Lightning ecosystem is illustrated in figure 2.2. [18]

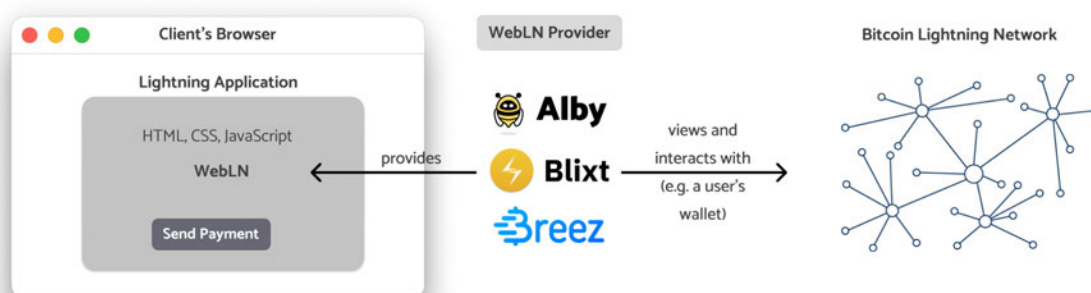


Figure 2.2: WebLN Provider Functionality [18]

By allowing for programmatic interactions, WebLN reduces the friction between Lightning apps and users' wallets, eliminating the need to switch contexts to scan a QR code or sign a message to prove ownership of a wallet. Additionally, when a WebLN client supports auto-payments, users can send payments with a single click without prompts. WebLN is a specification that describes the interaction with Bitcoin Lightning wallets and does not require third-party libraries. Over the years, it has become a recognized standard within the community. Initialization and execution of WebLN only require a few lines of JavaScript code, a widely used language for creating web apps. [18]

LNURL is an associated standard with WebLN, which provides a manual option for interacting with the Lightning Network. The correct implementation of WebLN can significantly enhance the user experience of Lightning apps. In instances where a user lacks a WebLN provider, LNURL can serve as an alternative option. [18]

One provider of WebLN is the versatile and open-source browser extension Alby. It has been specifically designed to enable deep integration of the Bitcoin Lightning Network with web applications. The primary focus of this extension is to facilitate the web payment process by implementing the WebLN standard as an interface between websites and Lightning Network nodes. This interface has enabled a seamless user experience for payments, authentication flows, and other related functionalities. With Alby, it is possible to set a budget for each website, allowing users to spend money without switching apps or constantly confirming payments while ensuring they do not overspend. [19]

Alby's goal is to provide a minimal web extension that allows browsers to interact programmatically with the Lightning Network, providing a reliable and efficient means for web applications to connect with Lightning Network nodes. In addition to supporting custodial and non-custodial setups, the Alby extension can connect to different node implementations, providing a high degree of flexibility for developers and users alike. As such, the extension has become a valuable tool for creating a seamless user experience for web payment and authentication flows. [19]

Given the extension's capabilities, it is an ideal candidate for the goal of this work: the development of a trustless connection between the Lightning Network and EVM-compatible blockchains. The use of Alby in this context can help to enhance the user experience further, making it more efficient and reliable. [19]

Due to the implementation of the WebLN standard in Alby, it only requires a few lines of code to send payments or create invoices. The code snippet in 2.2 shows how to send a payment using Alby. The function `sendPayment` takes a lightning invoice as an argument and returns a promise. The lightning invoice is a string that contains the payment hash and the amount to be paid, among other things. The promise resolves to an object or an error indicating whether the payment was successful. [20]

```
1 await webLn.enable();
2 const invoice = "lnbc10u1pjw9nnp...";
3 const result = await webLn.sendPayment(invoice);
```

Listing 2.2: WebLN sendPayment

Creating an invoice with Alby is just as easy. The code snippet in 2.3 shows how it is done programmatically. The function `createInvoice` takes an object as an argument, which contains the amount to be paid and an optional memo string, and returns a promise. The promise resolves to an object or an error indicating whether the invoice was created successfully. If the creation is successful, the object contains the invoice and the payment hash, a 32-byte string. The payment hash is used to identify the invoice and verify the payment. [20]

```
1 await webLn.enable();
2 const invoice = await webLn.makeInvoice({
3   amount: 1000,
4   defaultMemo: "This is an invoice."
5 });
```

Listing 2.3: WebLN makeInvoice

Executing the `sendPayment` or `createInvoice` function triggers the Alby extension showing up, as shown in figure 2.3. The user can then confirm or cancel the payment or invoice creation. This process happens in the user's browsers and does not require the user to switch the context, i.e., they do not need to use their phone to pay or create an invoice.

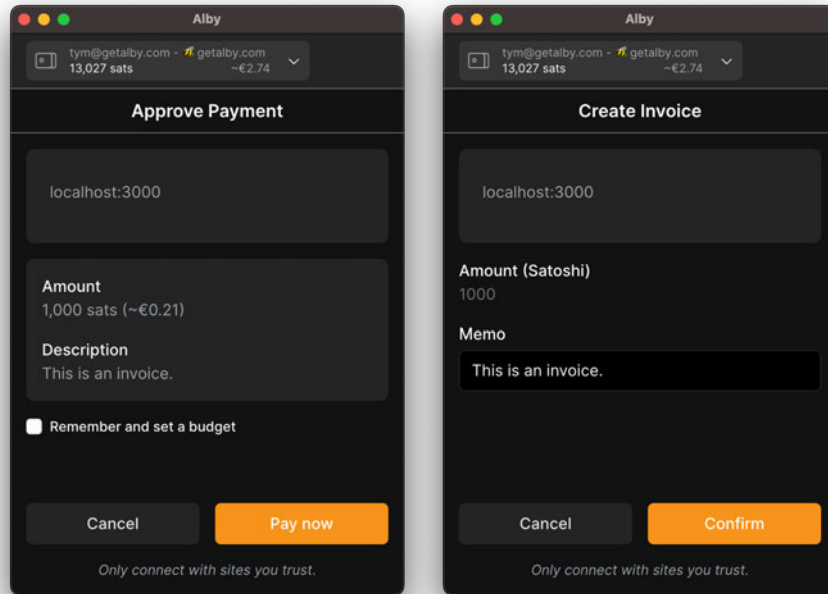


Figure 2.3: Alby Approve Payment and Create Invoice

Based on the fundamentals presented in this chapter, the following chapter describes two concepts for establishing a trustless connection between the Lightning Network and EVM-compatible blockchains.

3 Concept

This chapter presents two concepts for establishing a trustless connection between the Bitcoin Lightning Network and EVM-compatible blockchains. The objective is to transfer assets between the two ecosystems without the involvement of any third party. Each concept is described in detail, followed by an evaluation of its feasibility for implementation. The more promising of the two concepts is selected for implementation, which will be described in chapter 4.

3.1 Pre-signed Transaction

In the context of EVM-based blockchains, signing transactions without publishing them is possible, resulting in a valid transaction being held back. This transaction can be sent to the network to be mined anytime. This section presents a concept that aims to use pre-signed transactions that will only become valid once a payment has been made on the Lightning Network. This concept will be examined in detail, along with the challenges that arise during its implementation. Finally, the concept's suitability for implementation will be evaluated and assessed.

3.1.1 Protocol

The protocol is based on the idea of pre-signed transactions. A pre-signed transaction is a transaction that the sender has signed but not yet published to the network. The transaction can be published at any time and will be mined as soon as it is published. It does not have to be published by the sender itself but can instead be published by anyone else. Furthermore, this concept is based on the fact that upon a successful payment on the Lightning Network, the sender acquires the preimage linked to that payment. The payment hash of a payment is derived by applying the SHA256-algorithm to the preimage, as described in section 2.2.2. The approach of this concept is to use a pre-signed transaction that will only become valid with the knowledge of the preimage, i.e., after a successful payment on the Lightning Network. Figure 3.1 illustrates the protocol of this concept, which is described in detail below.

The protocol involves a customer seeking to purchase a certain amount of ether and pay via the Lightning Network. The operator manages a smart contract on the Ethereum network, which is used to transfer the requested ether to the customer. The core part of the smart contract is the `transferETH` function, which can be seen in code snippet 3.1. It can only be called by the operator of the smart contract due to the `onlyOperator` modifier. This modifier checks whether the transaction's sender is the operator or not. Essentially, the operator generates a pre-signed transaction that executes the `transferETH` function. This function requires four parameters: the receiver of the ether, the amount of ether to be sent, a payment hash, and the corresponding preimage to the payment hash. The function checks if the preimage and the payment hash belong together by applying the SHA256-algorithm to the preimage. It also checks if the specified amount of ether is available in the contract. If this is the case, the specified receiver will get their specified amount of ether.

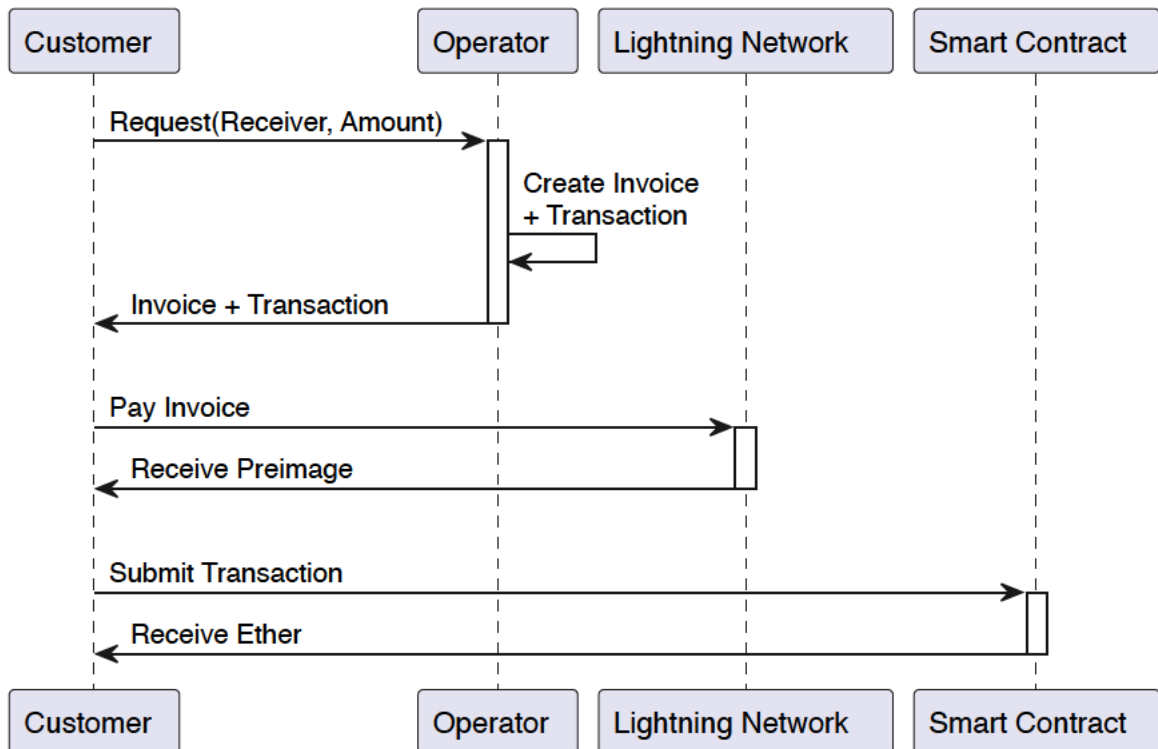


Figure 3.1: Protocol: Pre-signed Transaction

```

1 function transferETH (
2     address _receiver ,
3     uint256 _amount,
4     bytes32 _paymentHash,
5     bytes32 _preimage
6 ) public onlyOperator returns (bool) {
7
8     require(sha256(abi.encodePacked(_preimage)) == _paymentHash,
9         "ERR: preimage does not match payment hash");
10
11    require(_amount <= address(this).balance,
12        "ERR: balance insufficient");
13
14    (bool sent, bytes data) = _receiver.call{value: _amount}("");
15    require(sent, "ERR: unable to transfer eth");
16
17    return sent;
18 }
19
20 modifier onlyOperator() {
21     require(msg.sender == operator, "ERR: operator only");
22     _;
23 }
  
```

Listing 3.1: Smart Contract transferETH Function

The customer initiates the process by forwarding an HTTP request to the operator, which includes the specific amount of ether requested and the Ethereum address of the customer. The operator must ensure that the appropriate amount of ether is available in the smart contract. Then, the operator retrieves the current price of ether in Bitcoin from a suitable API to calculate the amount of Bitcoin required to purchase the requested amount of ether. Next, the operator creates a lightning invoice for the calculated amount of Bitcoin, which can be done programmatically as shown in 2.2.2. Upon generating the invoice, the operator acquires both the payment hash and preimage associated with the payment. Notably, while the creator of an invoice has knowledge of the preimage, this information remains unknown to any other party. At this stage, the operator possesses all the requisite parameters, namely the recipient, amount, payment hash, and preimage, to generate a transaction that calls the `transferETH` function. Finally, the operator signs the transaction using their private key, as illustrated in code snippet 3.2.

```
1  const transaction = contract.methods.transferETH(  
2    receiver , amount, preimage , paymentHash);  
3  
4  signedTransaction = await web3.eth.accounts.signTransaction(  
5    {  
6      from: accountFrom.address ,  
7      to: contractAddress ,  
8      value: amount ,  
9      data: transaction.encodeABI() ,  
10     gas: await transaction.estimateGas() ,  
11   } ,  
12   accountFrom.privateKey  
13 );  
14  
15 let rawTransaction = signedTransaction.rawTransaction;  
16 let X = "XXXXXX..." // 32 bytes of X's  
17 var result = rawTransaction.replace(preimage , X);
```

Listing 3.2: Create Pre-signed Transaction

The raw transaction appears as follows:

```
0xF9013481B18504A817C800828FB3946D7771FE079B0A3B9719052DA4F0C0E19A6023D6880  
DE0B6B3A7640000B8C43B9472B400000000000000000000000007BA055861DC7DB3BC36FE1ED  
D4D5CA06D8B5CAC000000000000000000000000000000000000000000000000000000000  
F42400000000000000000000000000000000000000000000000000000000000000000000  
CD146ECB131AA617337C85BA2EB01CB5E4DC3F2DFF46467915C554F8ED0000000000000000  
00000000000000000000000000000000000000000000000000000000201464FA47662A27C79EA5459275B4  
CB7671D68264974A41B9F4E5C5F5F52CBC5C820A96A0D22BB820BAF739808E5C9F75714C156  
AE6917A0522CEE4952D9140E67BC13782A07FE42E856DB9A1204BE9DA0EB8A0940FF720B8BB  
87018764AF3A61A1B49902A5
```


transaction will remain in the mempool until the operator creates a new valid transaction with nonce 11. At this point, the operator could withhold broadcasting the new transaction, causing the pre-signed transaction to remain stuck in the mempool indefinitely due to the gap in the nonce sequence. Thus, the customer would not receive the requested ether despite the operator receiving the lightning payment.

One potential way to address this issue is by implementing the following approach: The most recent nonce value associated with an account is publicly accessible. By verifying the operator's latest nonce value, the customer can ensure that the nonce of the pre-signed transaction immediately follows the nonce of the most recent transaction. If this is not the case, the customer may choose not to pay the invoice, preventing the risk of being deceived by the operator. In response to such a situation, the customer could request a new pre-signed transaction with a valid nonce or discontinue using this service.

Duplicated Nonce

If two transactions with the same nonce but different transaction data are sent, one will be confirmed, and the other will be rejected. The transaction confirmation will depend on the order in which they arrive at the first validating node that receives them, resulting in a random selection of which transaction is confirmed, as explained in more detail in section [2.1.3](#).

In the context of the presented concept, the operator can deceive the customer by intentionally sending a pre-signed transaction with a nonce already in use. As transactions can be held back, the customer cannot verify if there is no second transaction with the same nonce about to be published. The customer may pay the invoice and publish the transaction, while at the same time, the operator publishes a second transaction with the same nonce. The second transaction will likely be confirmed faster if the operator pays a high gas price, as explained in section [2.1.2](#). Thus, making the transaction published by the customer invalid. In this scenario, the customer will not receive the requested ether, as the transaction will be rejected, even though they have paid the invoice.

To address the issue of possible nonce duplication, the protocol could be extended as follows: all operators must register themselves in a registry smart contract and deposit a specified amount of ether as a stake, ensuring they act honestly. In case of malicious activity, their stake will be taken. After creating the pre-signed transaction, the operator must commit to not using the same nonce for another transaction. Following creating the pre-signed transaction, the operator is aware of its hash, which is the transaction's unique identifier. Now the commitment can be created, essentially a digital signature of the transaction's hash, the used nonce, and a timeout timestamp. This commitment ties the transaction hash to the nonce. If the operator creates a second transaction with the same nonce, the transaction's hash will differ from the one in the commitment. The timeout timestamp signals how long the commitment is valid. If the customer does not pay the invoice and thus does not publish the transaction, the operator can use the nonce for a different transaction after the timeout has passed. This commitment is sent along with the invoice and the pre-signed transaction to the user.

The registry contract acts as a decentralized court. If the operator tries to deceive the customer by using the same nonce for a different transaction, the customer can call the court function of the registry. In this function, the customer provides the operator's commitment and the second transaction where the operator uses the nonce again. The court function verifies the signature of the operator and the second transaction to determine if the operator acted maliciously (i.e., they used the nonce for another transaction while the commitment tied the nonce to the transaction sent to the customer). If the operator is found to have acted maliciously, the customer can penalize them by taking their stake. As a result, the operator is economically incentivized to act honestly. Their stake is at risk of being taken away if they attempt to deceive the customer using the same nonce for a different transaction.

Available Funds

In the presented concept, funds are not locked in any way during the swapping process, raising the possibility of the operator sending the customer a pre-signed transaction with a higher value than the available funds in the smart contract. This would result in an invalid transaction, and the customer would not receive the requested ether. To mitigate this risk, the customer could verify the available funds in the smart contract before paying the invoice. However, a certain amount of time passes between the customer's verification and the publication and processing of the transaction, during which anything could happen. For example, the operator may withdraw the funds from the smart contract after receiving the payment on the Lightning Network, leading to insufficient funds for the transaction. Consequently, the customer may pay the invoice and submit the transaction, only to find that the transaction is not mined due to the lack of funds in the smart contract.

One critical drawback of the presented concept is the lack of a locking mechanism to ensure the security and reliability of the transaction process. Specifically, the customer requires the assurance that they will receive the requested ether after paying the invoice, which is not guaranteed without a proper locking mechanism. Without such a mechanism, the customer must rely on trust in the operator regarding the availability of funds in the smart contract. However, this work aims to build a trustless solution.

Transaction Validation

The customer can verify the signature of the pre-signed transaction by utilizing the operator's public key. However, the operator may deceive the customer by signing a transaction with the wrong preimage, which does not match the payment hash associated with the invoice. Although this transaction would still be valid, it would fail when calling the `transferETH` function due to the mismatch between the preimage and payment hash. The customer could not verify the transaction's validity beforehand since they only receive the preimage after paying the invoice. Thus, the customer once again must rely on trust in the operator to ensure the validity of the transaction sent, which is not in line with the principles of a trustless solution.

3.1.3 Evaluation

The presented concept is a promising approach to connecting the Lightning Network and EVM-based blockchains. It utilizes pre-signed transactions that are only valid upon a successful payment on the Lightning Network. This is possible because the preimage is known to the sender after a successful payment. However, the concept also has some significant issues, as shown in the previous section. The issue with the duplicated nonce could be solved as stated, but it requires a complex protocol and much effort. Moreover, there are two issues that cannot be solved with the presented concept. Firstly, the customer cannot verify that the requested funds will be available after paying the invoice on the Lightning Network. Secondly, the customer cannot verify the validity of the pre-signed transaction. As a result, the concept is not fully trustless and, therefore, does not meet the requirements of this work.

The use of pre-signed transactions has been shown to have significant issues that make a fully trustless implementation unfeasible. Therefore, a new concept needs to be designed to meet the requirements of an utterly trustless solution. The new concept must ensure that the funds are locked for a specific period so that the customer can verify that they can claim their funds after paying the invoice on the Lightning Network. Additionally, it must allow the customer to create transactions independently, eliminating any potential problems related to the nonce. Finally, the relationship between the customer and the operator must be fully trustless, meaning neither party should rely on the other. Such a concept is presented in the next section.

3.2 Atomic Swaps

The concept of atomic swaps allows for a trustless exchange of values between different blockchains. This is achieved due to the use of hashed timelock contracts (HTLCs). The following section presents an atomic swap-based concept for establishing a trustless connection between the Lightning Network and EVM-compatible blockchains. This includes an explanation of HTLCs, a detailed description of the protocol of the concept, and an evaluation of the feasibility of implementing this concept.

3.2.1 HTLC

Hashed timelock contracts (HTLCs) are a type of smart contract that establishes a transactional agreement to produce conditional payments between two parties. HTLCs combine two concepts - hashlock and timelock - to ensure that the transaction takes place as per the agreed terms. A hashlock is a restraint that restricts the spending of an output until a specified kind of data is available, while a timelock locks out a transaction until a preset or pre-determined time is reached. In simple terms, HTLCs involve a sender of a transaction, locking cryptocurrency up in a contract, and sharing a secret with the recipient. If the recipient can provide the correct secret before a set deadline, they can spend the funds. If not, the funds go back to the sender. HTLCs benefit users looking for alternative ways to interact with others without intermediaries, as they are efficient, trustless, and offer validation. [21]

Hashed timelock contracts (HTLCs) can be implemented both on-chain and off-chain. On-chain HTLCs are executed on the blockchain itself, while off-chain HTLCs are executed outside of the blockchain. Such off-chain HTLCs are used in the Lightning Network to process payments. It uses payment channels, which are smart contracts between two parties. These payment channels can be used to route payments across multiple channels, and HTLCs are used to ensure the secure transfer of funds without the risk of intermediaries stealing them. Routing across multiple channels is an essential part of the Lightning Network because it is not optimal for users to open a payment channel with everyone they transact with. This is why the sender of a payment knows the preimage after payment: the HTLCs along the payment route must be unlocked to settle the payment. This unlocking process necessitates the payment receiver disclose the preimage, which is then transmitted along the route to the sender, making it a proof of payment as described in section 2.2.2. [22]

HTLCs are also used for cross-chain transactions, so-called atomic swaps, enabling users to transfer funds across different currently isolated blockchain ecosystems without requiring centralized exchanges as intermediaries. The only limitation is that both blockchains must be compatible with the same hashing algorithm. Because the Lightning Network utilizes the SHA-256 hashing algorithm, and EVM-compatible blockchains also support this algorithm, atomic swaps can establish a trustless and seamless connection between the Lightning Network and EVM-compatible blockchains.

3.2.2 Protocol

This protocol involves a customer seeking to purchase a certain amount of ether and pay via the Lightning Network. The operator, in this case, deployed a smart contract on the Ethereum network that implements hashed timelock contracts (HTLCs). This contract allows the creation of new HTLCs, which lock up a certain amount of ether with a hashlock and a timelock. The funds can only be spent if someone provides the correct secret before a deadline. If no one provides a correct secret, the person that created the HTLCs receives their funds back by withdrawing them from the contract.

The process, illustrated in figure 3.2, begins with the customer initiating an HTTP request to the operator, which includes the specific amount of ether requested and the Ethereum address of the customer. The operator then retrieves the current price of ether in Bitcoin from a suitable API to calculate the amount of Bitcoin required to purchase the requested amount of ether. Once the operator calculates the amount, a lightning invoice for the calculated amount of Bitcoin is created. Upon generating the invoice, the operator acquires both the payment hash and preimage associated with the payment. Notably, while the creator of an invoice knows the preimage, this information remains unknown to any other party. With all the necessary information, namely the payment hash, the receiver, and the amount of ether, the operator can create the HTLC in the smart contract on the Ethereum network, effectively locking up the requested amount of ether. The following describes the selection of both the hashlock and the timelock for the HTLC.

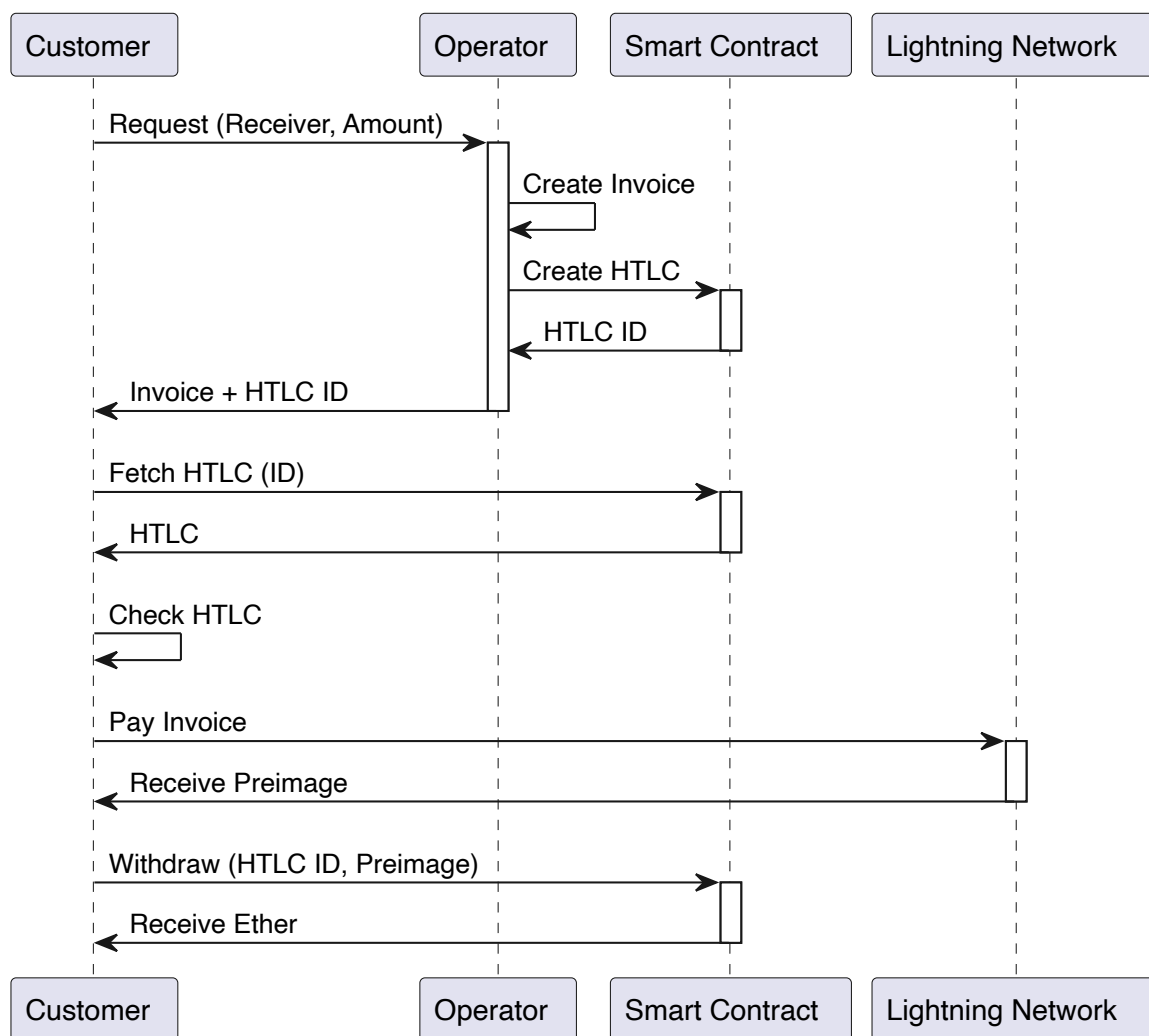


Figure 3.2: Protocol: Atomic Swap

The timelock component of the HTLC mechanism is a crucial aspect of its design. It acts as a protection measure to prevent the funds from being locked up indefinitely if the customer fails to complete the swap. The operator can choose an appropriate timelock period, such as 15 minutes, to ensure the customer has enough time to complete the transaction. This time window strikes a balance between providing enough time for the customer to complete the transaction and preventing the funds from being locked up for an extended period. If the customer fails to complete the transaction, the operator can withdraw the funds from the smart contract and use them for swaps with other customers.

The second crucial aspect of HTLCs is the hashlock. It is a cryptographic constraint that restricts the spending of the locked ether until the preimage to that hashlock is provided. The hashlock of an HTLC in this concept is the payment hash of the lightning invoice generated for the swap. The payment hash of an invoice is derived by applying the SHA256-algorithm to the preimage of this payment. Furthermore, the preimage of a payment is known to the sender after a successful payment, as explained in section 2.2.2. To unlock an HTLC, the presentation of the corresponding preimage to the hashlock is required, which is essentially a value that produces the hashlock when hashed using the SHA256-algorithm. Given that the hashlock used in the HTLC is the payment hash of the lightning invoice, the preimage of the lightning

invoice also serves as the preimage of the HTLC. Therefore, since the sender becomes aware of the preimage after successfully paying the lightning invoice, the sender can unlock the HTLC by presenting the obtained preimage after paying the invoice. This correlation is the basis for the concept presented in this section and is illustrated in figure 3.3.

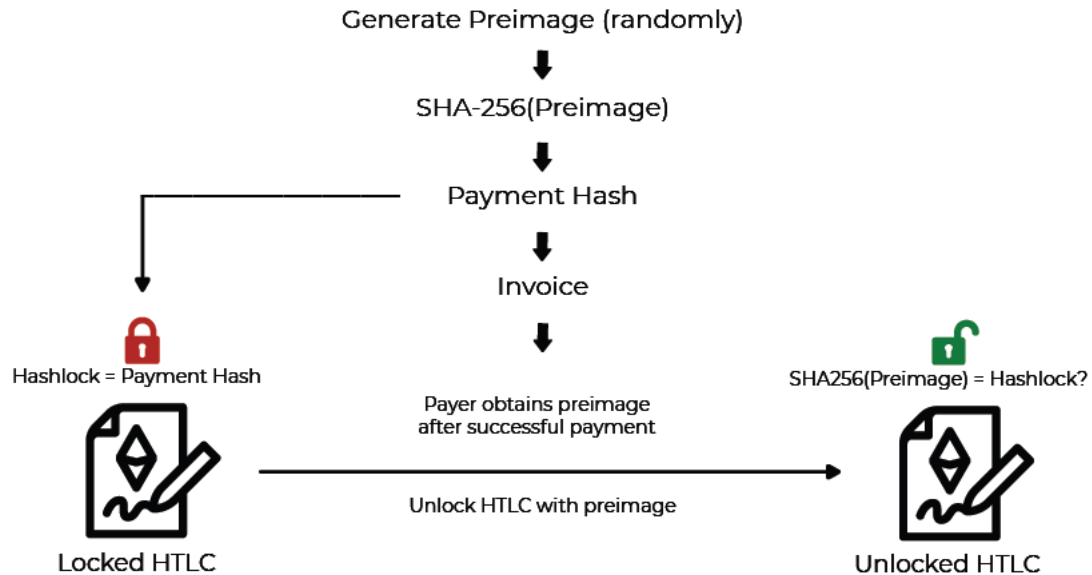


Figure 3.3: Locking and Unlocking HTLC

Upon successfully creating the HTLC in the smart contract, the operator is provided with an identifier (ID) for that specific HTLC by the smart contract. Subsequently, the operator sends the lightning invoice and the obtained ID to the customer as a response to the HTTP request. Upon receipt of the ID, the customer can retrieve all information regarding the HTLC by calling the smart contract. The customer should undertake several precautionary measures to ensure the integrity of the HTLC.

In order to ensure the integrity of the hashed timelock contract (HTLC), the customer must verify the payment hash of the received lightning invoice against the hashlock of the HTLC. If the two (i.e., the payment hash of the lightning invoice and the hashlock of the HTLC) do not match, the customer cannot unlock the HTLC after paying the lightning invoice. The operator may attempt to deceive the customer by utilizing a hashlock that differs from the payment hash of the invoice. However, the customer can easily detect this deception attempt by performing the verification check.

Additionally, the customer ensures that the locked amount corresponds to the requested amount and that the price of the lightning invoice corresponds to the value of the requested amount of ether. The operator may attempt to deceive the customer by either locking up an amount of ether that is less than the requested amount or by submitting a lightning invoice worth more than the requested amount. Nevertheless, the customer would be able to detect such deception attempts as well.

The customer should also confirm the following aspects: if the recipient address specified in the HTLC corresponds to the customer's address. If it does not match, the customer cannot claim the funds. Furthermore, the customer should ensure that the timelock of the HTLC is set to a few minutes in the future, allowing adequate time to pay the invoice and claim the funds. Given the possibility of high blockchain utilization, it is not uncommon for transactions to require some time before being mined. At such periods claiming the funds from the HTLC could take a significant amount of time. If the timelock is set too close, the customer may pay the lightning invoice, which transfers the bitcoin to the operator. However, they may be unable to withdraw their funds due to the transaction taking too long to be confirmed. Meanwhile, the HTLC may have already expired. Even if the timelock is set sufficiently in the future, the customer should select a high gas fee when initiating the claiming transaction to mitigate the risk of the transaction being stuck in the mempool for an extended period. As discussed in section 2.1.2, high gas fees generally lead to quicker transaction processing times, thereby minimizing the potential delay in claiming the funds.

If any of the verification checks fail, the customer will decline to pay the invoice, which results in the cancellation of the swap. In this case, the operator will keep their ether while the customer retains their Bitcoin. Thus, no trust is required from either party as all relevant details can be verified prior to transferring funds, ensuring the safety of both parties' funds.

Assuming the operator acts honestly, all verification checks by the customer should be successfully passed. Consequently, the invoice can be paid without the risk of deception. Upon payment of the invoice, the customer obtains the preimage used for the payment. As the hashlock of the HTLC is equivalent to the invoice's payment hash, obtaining the invoice's preimage also results in possessing the preimage for the HTLC. Therefore, the customer can claim the funds by calling the `withdraw` function (see 4.1) of the smart contract, providing the HTLC identifier and preimage. The smart contract then verifies the preimage for the corresponding HTLC by applying the SHA256-algorithm, ensuring that it matches the hashlock. If the customer has provided the correct preimage, the smart contract transfers the funds to the customer's specified address.

Finally, the customer has obtained their requested ether, while the operator has received the Bitcoin. Hence, the customer has effectively exchanged their Bitcoin for ether in a trustless manner. The presented concept leverages atomic swaps to enable fast and secure cross-chain swaps between the Lightning Network and EVM-compatible blockchains without requiring the parties involved to place trust in one another.

3.2.3 Evaluation

The presented concept offers a solution to the issues faced by the pre-signed transaction concept, as described in section 3.1.2. One of the key advantages of the proposed approach is that the customer initiates the transaction to claim the funds themselves, eliminating the difficulties associated with the nonce. Additionally, the operator locks the funds for a specific period, allowing the customer to verify the funds' availability before paying the lightning invoice. The utilization of hashed timelock contracts effectively resolves all the issues that render the pre-signed transaction concept unfeasible.

The proposed approach ensures that no party is required to trust one another. Before making any payments, the customer can retrieve the hashed timelock contract (HTLC) from the smart contract and verify all the relevant information, e.g., whether the hashlock matches the payment hash of the invoice. Only when all verification checks are passed will the customer pay the invoice, with the assurance that they can claim the funds afterward. If the operator attempts to deceive the customer, the customer would detect this during the verification checks and decline to pay the invoice, resulting in the cancellation of the swap. Conversely, the customer cannot claim the funds without paying the invoice, protecting the operator from fraud. If the operator chooses the preimage randomly, the customer cannot guess or compute it since the SHA256-algorithm is irreversible. If the customer cancels the swap, i.e., not paying the invoice although the verification checks passed, the operator can retrieve their funds back from the HTLC after the timelock expires. Thus, ensuring that the operator is not exposed to any risks of being deceived by the customer. Notably, the swap is atomic, meaning that once one party receives their funds, the other party also receives their funds, or neither party receives any funds. [23]

Using WebLN enables the development of a seamless user experience. The presented approach does not require the users to run their own lightning node or a node on their desired EVM-compatible blockchains. Furthermore, it allows for abstracting away the complexity associated with the process, thus making it easy to use for everyone. Finally, the security level of this approach is high since it eliminates the need for a complicated smart contract architecture, and the smart contract only contains funds during ongoing swaps.

The proposed approach is ideal for creating a trustless connection between the Lightning Network and EVM-compatible blockchains. The next chapter will provide a comprehensive description of the implementation of this concept.

4 Implementation

This chapter describes the implementation of the proposed concept in section 3.2 in detail. At its core, the implementation consists of three components:

- User Interface (Customer)
- Backend (Operator)
- Smart Contract (HTLC)

In order to effectively interact with an EVM-compatible blockchain and the Lightning Network, a suite of tools is necessary. Customers must utilize two browser extensions connected to the user interface, as shown in figure 4.1. Namely, Alby is utilized for interaction with the Lightning Network, while Metamask is employed for interaction with EVM-compatible blockchains. Conversely, the operator utilizes LNbits to create and pay invoices on the Lightning Network and leverages the web3.js-library to interact with EVM-compatible blockchains.

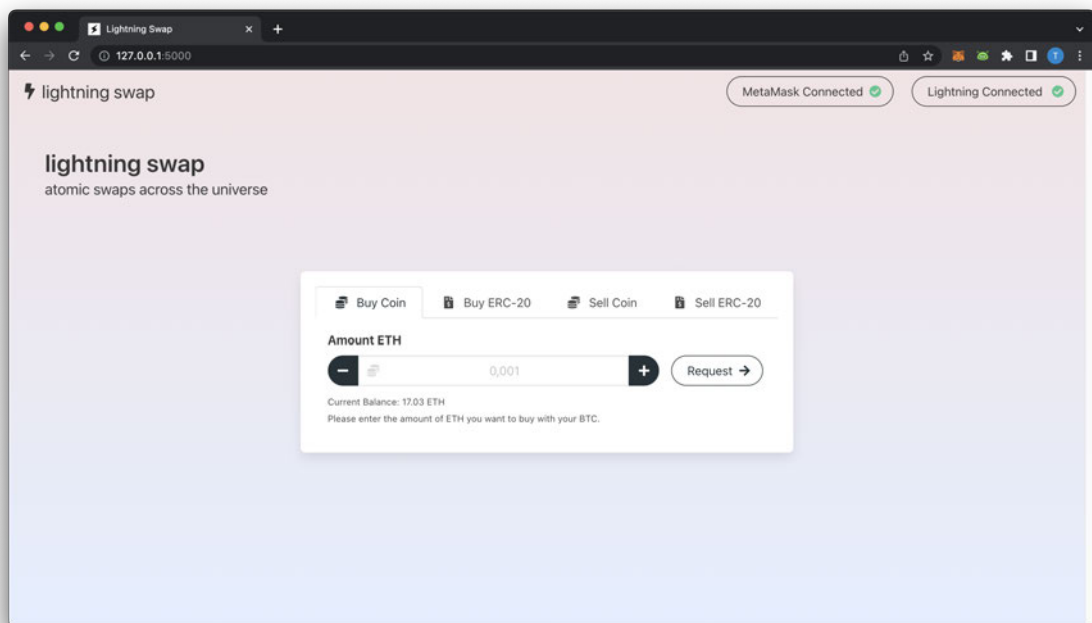


Figure 4.1: User Interface

The smart contract can be deployed onto any EVM-compatible blockchain, facilitating connectivity between the Lightning Network and the respective blockchain. It is important to distinguish between the native coin of a blockchain and tokens. The native coin is the currency of the blockchain system, such as ether on Ethereum, BNB on Binance Smart Chain, or MATIC on Polygon. Conversely, tokens do not possess their blockchain but function on top of other blockchains. These tokens are typically established following a particular standard, such as the ERC-20 standard. An example of such a token is Tether (USDT) on the Ethereum blockchain, which represents the value of one US dollar. [24, 25]

The term *native coin* is utilized in this work instead of specifying a particular cryptocurrency, such as ether, since the proposed connection is not restricted to the Lightning Network and Ethereum alone. Instead, the connection can be established between the Lightning Network and any EVM-compatible blockchain.

During development, the only network supported is the Ganache test network [26]. Consequently, the displayed coin in figure 4.1 is called *ETH*, and the user interface shows the prompt *Amount ETH*. However, once the application is deployed on other networks, the corresponding coin name will be displayed to the customer, such as *MATIC* on Polygon. A drop-down menu will be added next to the Metamask and Lightning buttons, allowing customers to choose their desired network. This will enable the customer to interact with their preferred network, and the user interface will adapt to reflect the specifics of the chosen network.

This chapter describes the smart contract, which implements the hashed timelock contract (HTLC) functionality. Subsequently, the four implemented directions, which include buying native coins and ERC-20 tokens with Bitcoin on the Lightning Network (LN) and selling native coins and ERC-20 tokens for Bitcoin on the LN, are described in detail.

4.1 HTLC in Solidity

Implementing a hashed timelock contract (HTLC) within a smart contract is a crucial component of the proposed concept and can be achieved using the Solidity programming language. The basis for the HTLC implementation presented in this work is a library available on GitHub [27]. Although the HTLC mechanism is similar for native coins and ERC-20 tokens, there are slight variations in the implementation for ERC-20 tokens. This section is split into two parts: firstly, a comprehensive description of the HTLC implementation for a native coin is provided, and secondly, the necessary modifications that need to be made to enable support for ERC-20 tokens are elaborated.

The data for a hashed timelock contract (HTLC) is stored in a struct, as illustrated in code snippet 4.1. The struct consists of several key elements, including the addresses of the sender, who created the HTLC, and the receiver, who is intended to receive the funds from the HTLC. It also contains the two components fundamental to an HTLC: the `hashlock` and `timelock`. The `hashlock` is a 32-byte hash, while the `timelock` is a timestamp in the future. The `amount` parameter specifies the number of native coins associated with the HTLC. The boolean variables `withdrawn` and `refunded` ensure that the HTLC can only be withdrawn or refunded once. Finally, the preimage of the HTLC is stored once the receiver publishes it to withdraw the funds. The data for each HTLC is stored in a map called `contracts`, which uses a 32-byte identifier (ID) to map to the corresponding HTLC data stored in the struct.

```
1 struct LockContract {
2     address payable sender;
3     address payable receiver;
4     bytes32 hashlock;
5     uint timelock;
6     uint amount;
```

```

7     bool withdrawn;
8     bool refunded;
9     bytes32 preimage;
10  }
11
12  mapping (bytes32 => LockContract) contracts;

```

Listing 4.1: HTLC Struct + Mapping

In order to efficiently verify the existence of a hashed timelock contract (HTLC) associated with a particular identifier (ID), the smart contract features a `haveContract` function, as illustrated in figure 4.2. The function accepts a 32-byte ID as input and determines whether an HTLC is associated with that ID. The function returns a boolean value of `true` if the HTLC exists and `false` otherwise. Using the internal keyword within the function's definition restricts the function from being called outside of the smart contract. As such, the `haveContract` function is called by other functions within the smart contract to confirm the existence of an HTLC for a given ID.

```

1  function haveContract(
2     bytes32 _contractId
3  ) internal view returns (bool exists) {
4     exists = (contracts[_contractId].sender != address(0));
5  }

```

Listing 4.2: HTLC `haveContract`

To create a new hashed timelock contract (HTLC), the smart contract includes a `newContract` function, which can be seen in code snippet 4.3. This function requires three parameters: the intended receiver, who will withdraw funds from the HTLC by providing the preimage, the hashlock, and the timelock. The amount of native coins locked up in the new HTLC is defined by the `msg.value`, which represents the number of native coins included in the transaction that executes the `newContract` function. In the beginning, the function verifies two critical aspects: firstly, whether the amount of native tokens sent with the transaction is greater than zero, and secondly, whether the timelock of the to-be-created HTLC is in the future. The identifier (ID) for the new HTLC is generated by hashing the function call parameters, which include the sender of the transaction, the receiver, the number of native coins, the hashlock, and the timelock. Subsequently, the `haveContract` function is used to check whether an HTLC already exists for that ID. Finally, if no HTLC exists for that ID, all the relevant information is stored in the struct and mapped to the ID using the `contracts` variable. Anyone can call this function, which returns the 32-byte ID for the HTLC.

```

1  function newContract(
2     address payable _receiver,
3     bytes32 _hashlock,
4     uint _timelock
5  ) external payable returns (bytes32 contractId) {
6     require(msg.value > 0, "msg.value must be > 0");

```

```

7   require(_timelock > block.timestamp,
8       "timelock time must be in the future");
9
10  contractId = sha256(
11      abi.encodePacked(
12          msg.sender, _receiver, msg.value, _hashlock, _timelock)
13  );
14
15  if (haveContract(contractId)) revert("Contract already exists");
16
17  contracts[contractId] = LockContract(
18      payable(msg.sender), _receiver, msg.value, _hashlock,
19      _timelock, false, false, 0x0
20  );
21 }

```

Listing 4.3: HTLC newContract

Access to information associated with an HTLC can be obtained through the `getContract` function by providing the relevant HTLC identifier (ID), as illustrated in code snippet 4.4. This function is declared with the `view` keyword, meaning it only reads data from the blockchain and does not modify any data. Before returning any information, this function uses the `haveContract` function to verify whether an HTLC exists for the given ID. If the HTLC exists, it returns all the information associated with the HTLC that is stored in the struct.

```

1  function getContract(
2      bytes32 _contractId
3  ) public view returns (
4      address sender, address receiver, uint amount, bytes32 hashlock,
5      uint timelock, bool withdrawn, bool refunded, bytes32 preimage
6  ) {
7      if (haveContract(_contractId) == false)
8          return (address(0), address(0), 0, 0, 0, false, false, 0);
9
10     LockContract storage c = contracts[_contractId];
11
12     return (c.sender, c.receiver, c.amount, c.hashlock,
13         c.timelock, c.withdrawn, c.refunded, c.preimage
14     );
15 }

```

Listing 4.4: HTLC getContract

The primary objective of an HTLC is to enable the receiver to unlock and claim their funds once they possess the preimage that unlocks the hashlock. This functionality is provided by the `withdraw` function of the smart contract, as illustrated in code snippet 4.5. The function accepts the identifier (ID) of the HTLC and the preimage to that HTLC as parameters and

returns a boolean value indicating whether the withdrawal was successful or not. Several checks are performed to ensure the caller can claim the funds. Firstly, the `haveContract` function is used to verify the existence of the HTLC for the given ID. Secondly, the provided preimage is checked by applying the SHA256-algorithm to ensure it is the preimage to the hashlock. Other checks include verifying if the caller of the function is the receiver of the HTLC funds, ensuring that the HTLC has not been withdrawn, and that the timelock of the HTLC has not yet expired. If all checks are successful, the preimage is stored in the struct, and the status of the HTLC is updated to *withdrawn*. Finally, the amount of native coins locked up in the HTLC, as specified in the `amount` variable, is transferred to the designated receiver.

```
1 function withdraw(  
2     bytes32 _contractId ,  
3     bytes32 _preimage  
4 ) external returns (bool) {  
5     require(haveContract(_contractId), "contractId does not exist");  
6     require(  
7         contracts[_contractId].hashlock ==  
8         sha256(abi.encodePacked(_preimage)),  
9         "hashlock hash does not match"  
10    );  
11    require(  
12        contracts[_contractId].receiver == msg.sender,  
13        "withdrawable: not receiver"  
14    );  
15    require(  
16        contracts[_contractId].withdrawn == false ,  
17        "withdrawable: already withdrawn"  
18    );  
19    require(  
20        contracts[_contractId].timelock > block.timestamp,  
21        "withdrawable: timelock time must be in the future"  
22    );  
23  
24    LockContract storage c = contracts[_contractId];  
25    c.preimage = _preimage;  
26    c.withdrawn = true;  
27    c.receiver.transfer(c.amount);  
28    return true;  
29 }
```

Listing 4.5: HTLC withdraw

If the intended receiver of the HTLC funds fails to unlock the HTLC, the creator of the HTLC needs to reclaim the funds. However, this cannot be achieved using the `withdraw` function since the creator of the HTLC is not the intended receiver. In such scenarios, the `refund` function is required, which can be seen in code snippet 4.6. The creator of the HTLC can call this function after the timelock has expired to reclaim their funds. The function conducts

several checks, including verifying the existence of the HTLC for the given ID, ensuring that the caller of the function is the creator of the HTLC, and confirming that the funds for the HTLC have not been refunded or withdrawn. Finally, the function ensures that the timelock has expired. If these checks are successful, the status of the HTLC is updated to *refunded*, and the designated amount of native coins is transferred back to the creator of the HTLC.

```
1 function refund(  
2     bytes32 _contractId  
3 ) external returns (bool) {  
4     require(haveContract(_contractId), "contractId does not exist");  
5  
6     require(  
7         contracts[_contractId].sender == msg.sender,  
8         "refundable: not sender"  
9     );  
10    require(  
11        contracts[_contractId].refunded == false,  
12        "refundable: already refunded"  
13    );  
14    require(  
15        contracts[_contractId].withdrawn == false,  
16        "refundable: already withdrawn"  
17    );  
18    require(  
19        contracts[_contractId].timelock <= block.timestamp,  
20        "refundable: timelock not yet passed"  
21    );  
22  
23    LockContract storage c = contracts[_contractId];  
24    c.refunded = true;  
25    c.sender.transfer(c.amount);  
26    return true;  
27 }
```

Listing 4.6: HTLC refund

As mentioned previously, the HTLC mechanism is applicable to both native coins and ERC-20 tokens. To enable the usage of ERC-20 tokens, slight modifications must be made to the smart contract presented earlier. Firstly, the ERC-20 standard must be imported, enabling the smart contract to handle all ERC-20 tokens by providing the token's address. The ERC-20 standard comprises a set of functions that all compliant tokens must implement. By importing this standard, the smart contract can interact with ERC-20 tokens in a standardized manner. The following describes the modifications to the smart contract that enable it to lock up and transfer ERC-20 tokens similarly to native coins.

In order to handle ERC-20 tokens, the struct from code snippet 4.1 must be extended by one element: the `tokenAddress`. This element must store the ERC-20 token address associated with the HTLC. The token address and the number of tokens must be passed to the `newContract` function when creating the HTLC. Unlike native coins, ERC-20 tokens cannot be sent along with a transaction. Therefore, the smart contract must collect the tokens from the creator of the HTLC when creating the HTLC. To accomplish this, a feature of the ERC-20 standard must be used that allows others to spend one's tokens. Thus, the creator of the HTLC must grant permission for the smart contract to spend a certain amount of tokens before calling the `newContract` function. If the so-called *allowance* exists, the smart contract transfers the specified amount of ERC-20 tokens from the creator of the HTLC to the smart contract upon creating a new HTLC, effectively locking them up.

The `withdraw` and `refund` functions remain mostly unchanged from those for native coins. The only modification required is in the way the funds are transferred. To transfer ERC-20 tokens, the smart contract must call the `transfer` function of the corresponding ERC-20 token contract. This function transfers the specified amount of ERC-20 tokens to the intended recipient. The same checks and conditions that apply to the native coin version of these functions also apply to their ERC-20 counterpart, such as verifying the existence of an HTLC with the given ID, checking whether the caller is authorized to withdraw or refund the funds, and ensuring that the funds have not already been withdrawn or refunded. Once these checks are completed, the smart contract transfers the appropriate number of ERC-20 tokens to the intended recipient.

The two smart contracts, one for HTLCs involving native coins and one for HTLCs involving ERC-20 tokens, are among the core components of this implementation, along with the user interface and the backend. The interaction between all these components is described in detail in the following sections.

4.2 Lightning - Native Coin

This section outlines the scenario where a customer purchases native coins using Bitcoin on the Lightning Network. The protocol for this scenario is illustrated in figure 4.2 and will be described in detail in this section.

Before initiating the swap, the customer must have their Metamask and Alby wallet connected to the user interface. To begin the process, the customer selects the network and desired amount of coins they wish to purchase with Bitcoin on the Lightning Network. Next, they send an HTTP POST request to the operator at the `offerCoinBuy` endpoint, which includes the desired amount of coins, the customer's address, and the desired network.

Before conducting a swap, the operator sends the customer an offer for that swap. The offer contains a lightning invoice that the customer must pay to accept the offer. Although this adds complexity to the protocol, it is necessary to protect the operator from spam. The operator incurs gas costs for creating the HTLC and may incur additional costs if the customer does not conduct the swap. In this case, the operator has to pay extra gas fees for a refund. Without this additional step with the additional invoice, a customer could request a swap

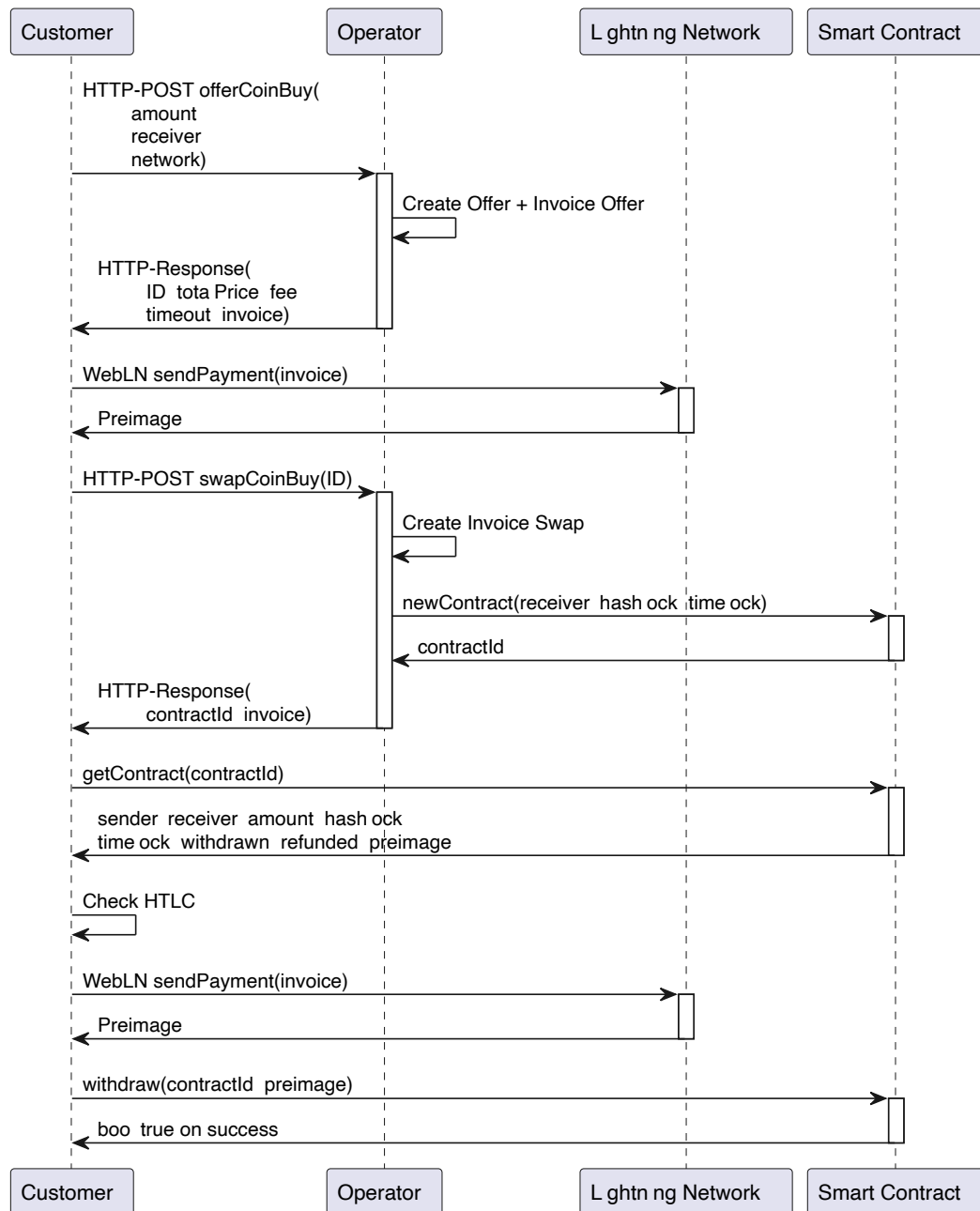


Figure 4.2: Protocol: Lightning - Native Coin

without any intention of conducting it, leaving the operator to pay for two transactions and lose liquidity for some time. To prevent fraud, the operator sends the customer an offer and a lightning invoice that must be paid to accept the offer.

The operator creates an offer for the customer by fetching the current exchange rate of the requested coin to Bitcoin, which is used to calculate the price for the swap. Next, the operator estimates the gas costs for creating a new hashed timelock contract (HTLC) and the refund for that HTLC. This can be done programmatically using the web3.js library, as described in section 2.1.2. The sum of the costs for the creation and refund is the amount of the lightning invoice the customer must pay to accept the offer. This protects the operator from losses even if the customer accepts the offer but does not fully conduct the swap, which requires the operator to refund the funds from the HTLC. However, if the customer successfully conducts

the swap, they would pay for the refund transaction, although this transaction would not take place in a successful swap scenario. This has to be considered in the price calculation, which can be illustrated using the following example.

Assuming the following numbers: the customer requests one coin that costs 100 sats¹, the costs for creating and refunding the HTLC are ten sats each, and the operator charges a fee of 1 percent for each swap. As mentioned earlier, the cost of accepting the offer is the combined expenses for creating and refunding the HTLC. In this scenario, it would be 20 sats. The total price of the swap amounts to 100 sats plus a 1% fee, resulting in a total of 101 sats. If the customer pays the calculated price, the swap will be conducted successfully, and the refund transaction will not be necessary for the operator. However, the customer has already paid for the refund transaction in the invoice for accepting the offer. To make the price fair for both parties, the costs of the refund transaction (10 sats in this scenario) can be subtracted from the total price. In this case, the total price for the requested coin would be 91 sats. In total, the customer paid 111 sats, 100 sats for the coin, ten sats for the creation of the HTLC, and one sat in fees.

The offer also includes a timeout typically set for a few minutes. This timeout is designed to provide enough time for the customer to consider the offer while preventing the operator from committing to a specific exchange price for an extended period. If the timeout were set for a longer period, the customer could wait for the exchange rate between the coin and Bitcoin to improve before conducting the swap, potentially resulting in a better exchange rate for the customer. Finally, the operator saves the offer to their database and sends it to the customer. The offer includes an identifier, the total price, the fee, the timeout, and the lightning invoice that must be paid to accept the offer.

The offer is displayed to the customer in the user interface, as shown in figure 4.3. The customer may accept the offer within the given timeout by clicking the *Accept Offer* button. This action opens the connected Alby wallet and prompts the user to pay the invoice, equivalent to the costs for creating and refunding the HTLC (20 sats in this scenario). The customer has to approve the payment within the wallet, and the Lightning Network will process the payment.

Upon payment of the lightning invoice, the user interface sends an HTTP POST request to the `swapCoinBuy` endpoint of the operator, containing the identifier (ID) of the offer to initiate the creation of the hashed timelock contract (HTLC). The operator performs several checks, including verifying if an offer exists for the given ID (i.e., looking for an offer with the given ID in their database), if the customer responded within the specified timeout, and if the customer paid the invoice. If all conditions are met, the operator proceeds to create the lightning invoice in the amount of the total price. The payment hash of the invoice is used as the hashlock for creating the HTLC, as described in detail in section 3.2.2. The operator creates a new HTLC by executing the `newContract` function of the smart contract. This function returns a 32-byte identifier for the HTLC, referred to as the *contractId*. The user interface receives the lightning invoice and the *contractId* as a response to the HTTP request.

¹Satoshis [the smallest unit of Bitcoin]

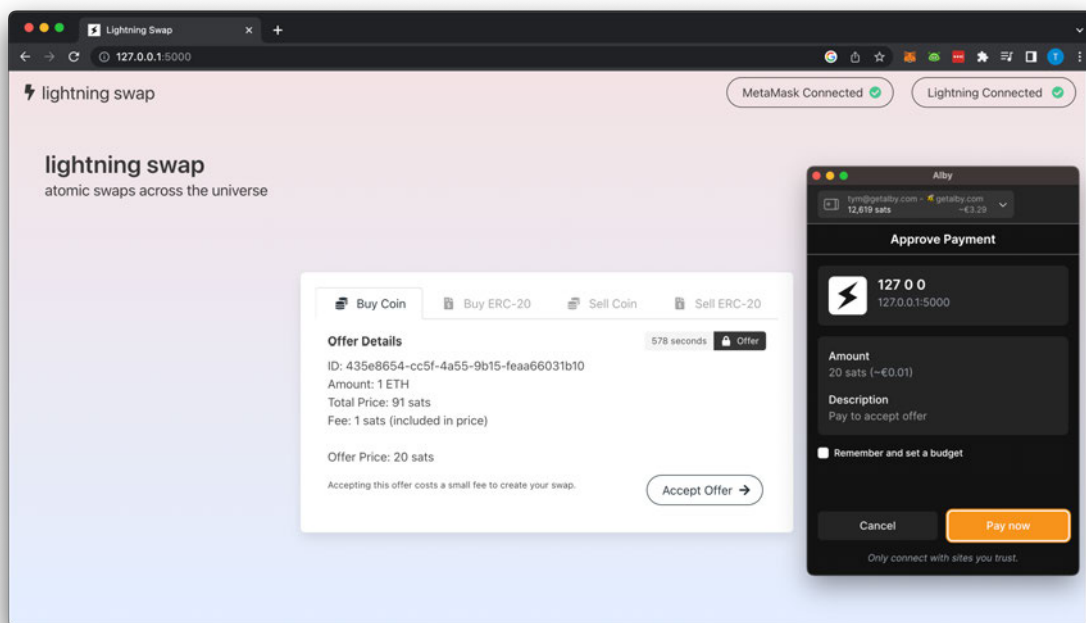


Figure 4.3: User Interface: Offer Buy Coin

Before paying the lightning invoice, the customer must ensure the integrity of the hashed timelock contract (HTLC). The user interface fetches the HTLC from the smart contract using the `getContract` function. This function returns all relevant information associated with the HTLC, including the creator and receiver of the HTLC, the amount, the hashlock, the timelock, and two boolean values indicating whether the HTLC has already been withdrawn or refunded. Furthermore, the preimage of the HTLC is returned, which is an empty string at this point. In addition to fetching the HTLC information, the user interface decodes the lightning invoice to reveal the payment hash, among other details.

The user interface performs several verification checks before prompting the customer to pay the lightning invoice. These checks include verifying whether the payment hash of the invoice matches the hashlock of the HTLC, if the timelock of the HTLC is more than five minutes in the future, if the locked amount in the HTLC corresponds to the requested amount, and if the receiver of the HTLC is the customer's address. The verification checks are described in more detail in section 3.2.2.

After ensuring the integrity of the HTLC, the customer is ready to pay the lightning invoice by clicking on the *Pay Invoice* button on the user interface. This opens the Alby wallet and prompts the customer to confirm the payment. Once the Lightning Network successfully processes the payment, the preimage of the payment is known to the customer and is displayed on the user interface. The customer uses this preimage to withdraw their requested coins from the HTLC. By clicking on the *Claim* button, the user interface creates a transaction and prompts the customer to confirm it in their MetaMask wallet, as shown in figure 4.4. The transaction calls the `withdraw` function of the smart contract, submitting the identifier of the HTLC and the preimage. The smart contract then verifies if the preimage matches the hashlock of the HTLC, as described in more detail in 4.1. Finally, the smart contract

transfers the coins locked in the HTLC to the customer, completing the swap. As a result, the customer received their requested coins, and the operator received their Bitcoin on the Lightning Network in exchange.

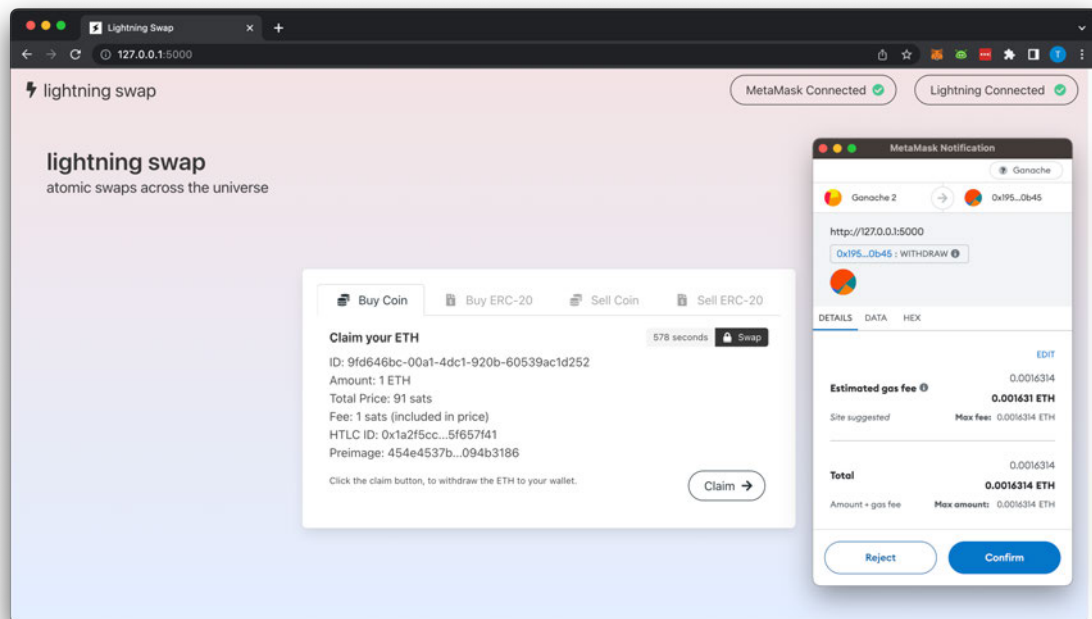


Figure 4.4: User Interface: Claim Coin

In this section, the protocol for conducting a successful swap has been described. However, it should be noted that not all swaps may be successfully conducted. If the customer fails to accept the offer, the operator will delete the offer once the timeout period has expired. The customer cannot accept the offer after the timeout has elapsed. If the customer accepts the offer but fails to pay the invoice for conducting the swap, the operator must wait for the HTLC to expire due to timelock. Once it expires, the operator can initiate a refund by calling the refund function of the smart contract. In this scenario, the only funds lost are the gas fees for creating and refunding the HTLC. However, as previously described, the fees associated with the creation and refund of the HTLC are paid by the customer upon acceptance of the offer.

4.3 Lightning - ERC-20 Token

In addition to buying native coins, customers can also purchase ERC-20 tokens that the operator on the relevant network supports. The protocol for buying ERC-20 tokens requires several modifications compared to native coins. The modified protocol is illustrated in figure 4.5. The necessary changes are discussed in detail in this section, while aspects of the protocol that remain unchanged from buying native coins are only briefly addressed.

Every blockchain only has one native coin, but it may support many ERC-20 tokens, which are specified by their unique address. Therefore, the customer's first step is to select their desired ERC-20 token. To achieve this, the user interface provides a drop-down menu that

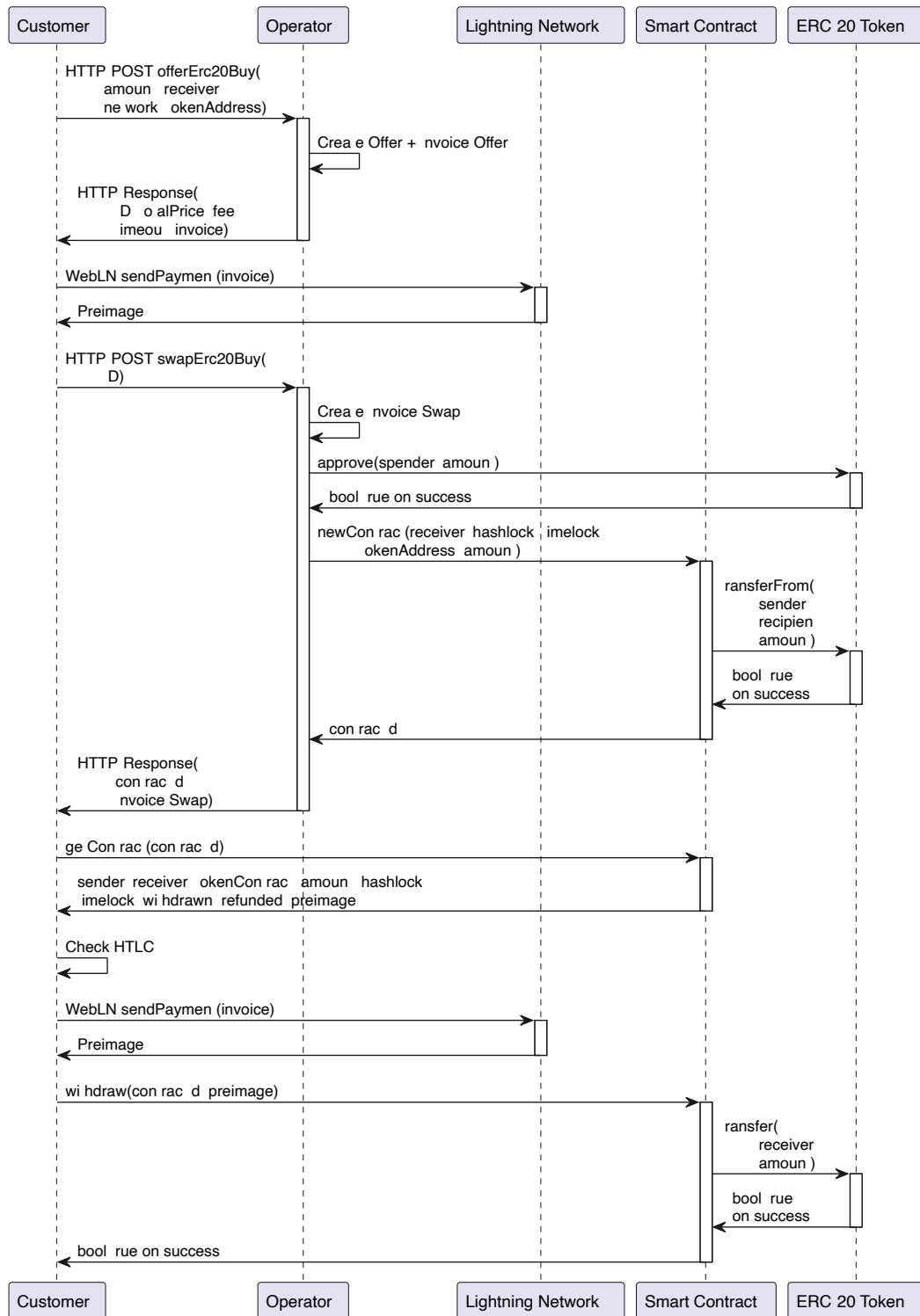


Figure 4.5: Protocol: Lightning - ERC-20 Token

displays all supported tokens for the particular network, as shown in figure 4.6. Once the customer has selected the token and the desired amount, they initiate the buying process by sending an HTTP POST request to the `offerErc20buy` endpoint of the operator. Compared to the request for buying native coins, this request contains one additional parameter: the address of the desired ERC-20 token.

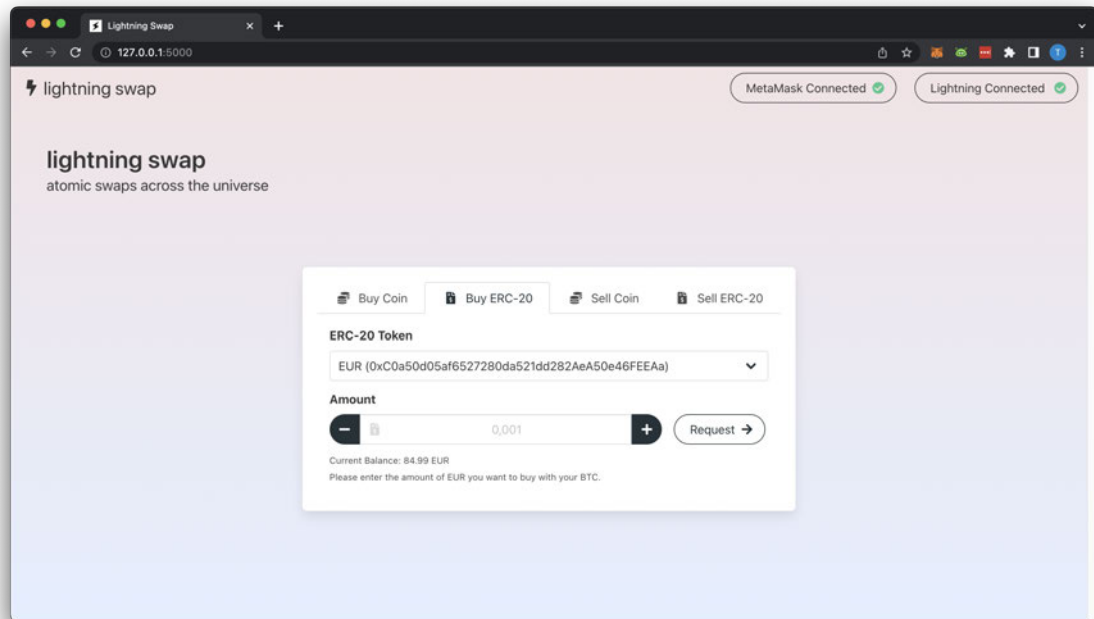


Figure 4.6: User Interface: Buy ERC-20 Token

At this point, the protocol is the same as the one for buying native coins. The operator fetches the current exchange rate between Bitcoin and the ERC-20 token and creates an offer along with a lightning invoice to be paid by the customer. The offer and lightning invoice are then sent to the customer in response to the HTTP request, and the customer pays the lightning invoice to accept the offer. Upon payment, the user interface sends an HTTP POST request to the `swapErc20Buy` endpoint of the operator, containing the identifier (ID) of the offer to initiate the creation of the hashed timelock contract (HTLC).

First, the operator creates the lightning invoice for the swap, as the payment hash is needed to create the HTLC. Creating an HTLC for ERC-20 tokens differs from native coins in a few key ways. As discussed in section 4.1, unlike native coins, ERC-20 tokens cannot be sent along with a transaction. Therefore, the smart contract must collect the ERC-20 tokens from the operator upon creating a new HTLC. To accomplish this, the operator must create an *allowance* for the smart contract by calling the `approve` function of the ERC-20 token. After creating the allowance, the operator can call the `newContract` function, which collects the tokens from the operator using the `transferFrom` function of the ERC-20 token and locks them up in an HTLC. The operator receives an identifier, the *contractId*, for the newly created HTLC, which is sent to the user interface along with the lightning invoice for the swap in response to the HTTP request.

The remaining steps of the protocol are essentially the same as for buying native coins with Bitcoin on the Lightning Network. The customer fetches and verifies the HTLC, pays the lightning invoice, and calls the `withdraw` function to claim their ERC-20 tokens. However, the `withdraw` function differs slightly for ERC-20 tokens because they cannot be sent along with a transaction. Instead, the tokens must be transferred to the customer using the `transfer`

function of the ERC-20 token. Once the transfer is complete, the customer has successfully obtained their requested ERC-20 tokens, and the operator has received their Bitcoin on the Lightning Network in exchange.

4.4 Native Coin - Lightning

The presented approach in section 4.2 can be used in reverse, allowing the customer to sell their native coins for Bitcoin on the Lightning Network, illustrated in figure 4.7.

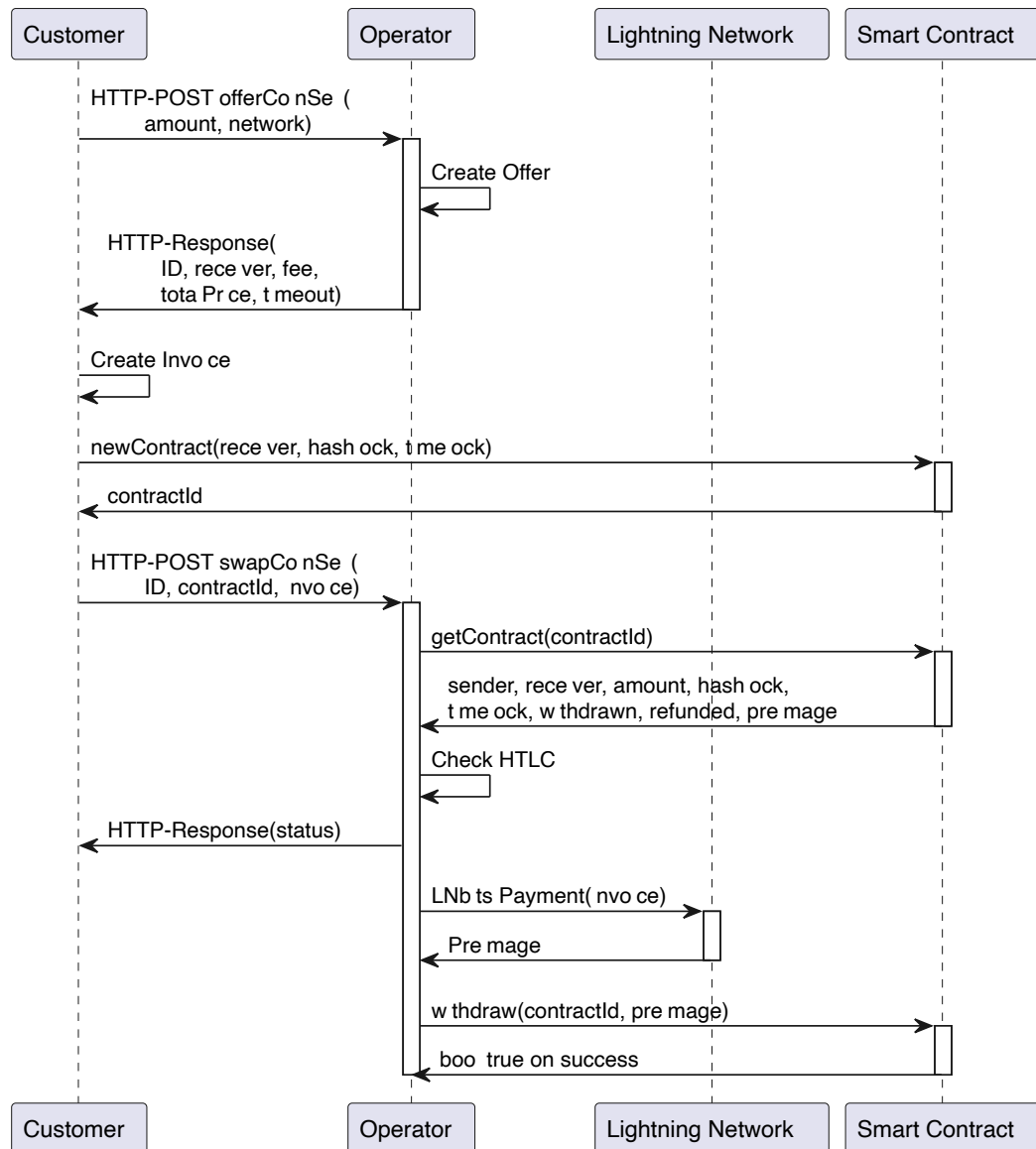


Figure 4.7: Protocol: Native Coin - Lightning

The customer initiates the process by selecting the desired amount of tokens they want to sell. The user interface for this looks the same as the one shown in figure 4.1. The customer then requests an offer from the operator by sending an HTTP POST request to the `offerCoinSell` endpoint. This request contains the amount the customer wants to sell and the desired network. The operator fetches the current exchange rate between the coin and Bitcoin to

create an offer. The operator can then calculate how much they will pay for the number of coins the customer wants to sell. The amount the operator is willing to pay is slightly lower than the actual exchange rate due to a fee that the operator raises. Furthermore, the operator chooses how long they will pay this amount by setting the timeout accordingly. Finally, the operator sends the offer to the customer in response to the HTTP request. This response contains an identifier for the offer, the operator's address, the fee, the amount of Bitcoin the customer receives for their coins, and the timeout for the offer. The operator's address is required to create the hashed timelock contract (HTLC) because the operator unlocks the HTLC later, i.e., is the receiver of the HTLC's funds.

Upon receiving the offer from the operator, the customer can review it in the user interface, as shown in figure 4.8. If the customer accepts the offer, they must do so within the given timeout. To initiate the swap, the customer must create the lightning invoice and HTLC accordingly. The lightning invoice is created using Alby by clicking the *Accept offer (Create Invoice)* button. The invoice amount is the price the operator is willing to pay, which was sent with the offer. The customer can confirm the creation of the lightning invoice by clicking the *Confirm* button.

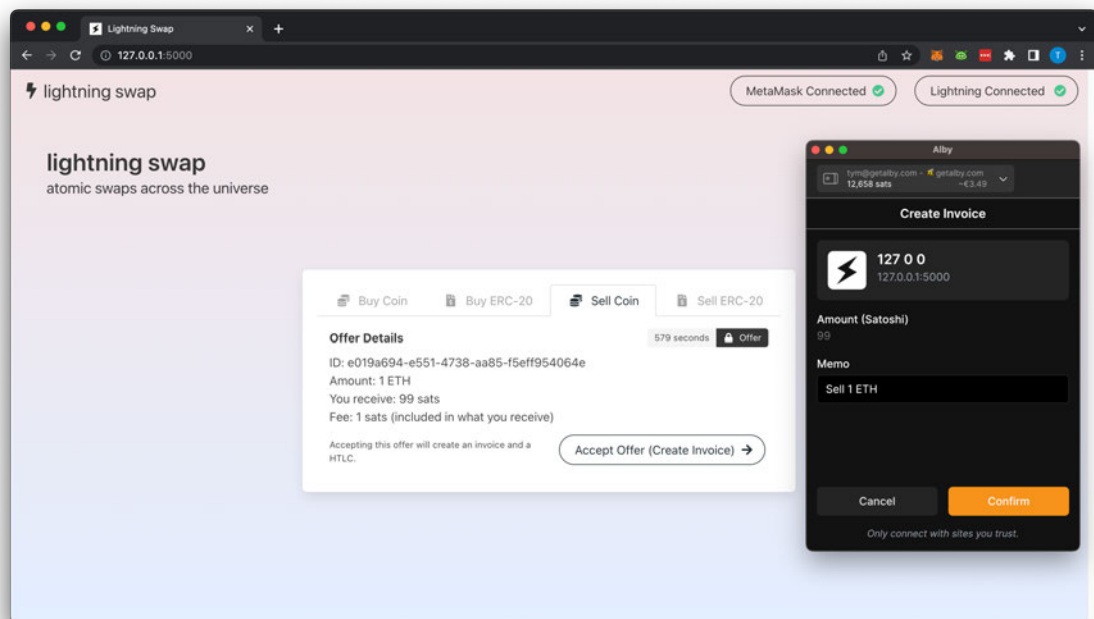


Figure 4.8: User Interface: Offer Sell Coin

After creating the lightning invoice, the customer has everything they need to proceed with the creation of the HTLC. The operator is set as the receiver of the HTLC, with the payment hash of the prior created lightning invoice serving as the hashlock. The HTLC timeout is set to a few minutes in the future. The user interface then prompts the customer to confirm the creation of a new HTLC by executing the `newContract` function of the smart contract via their Metamask wallet. Upon execution, the function returns a 32-byte identifier for the HTLC, referred to as the *contractId*.

Upon creation of the hashed timelock contract (HTLC), the user interface sends an HTTP POST request to the `swapCoinSell` endpoint of the operator, which includes the ID of the offer, the `contractId`, and the lightning invoice. To ensure the integrity of the HTLC, the operator fetches the HTLC from the smart contract using the `contractId` and performs several verification checks. These include verifying if the hashlock of the HTLC matches the payment hash of the lightning invoice and performing other checks, as described in more detail in section 3.2.2. If all checks are passed, the operator sends a positive status update to the user interface in response to the HTTP request. The user interface then updates accordingly, informing the customer that they must wait for the operator to pay the invoice and claim their funds from the HTLC, which can be seen in figure 4.9.

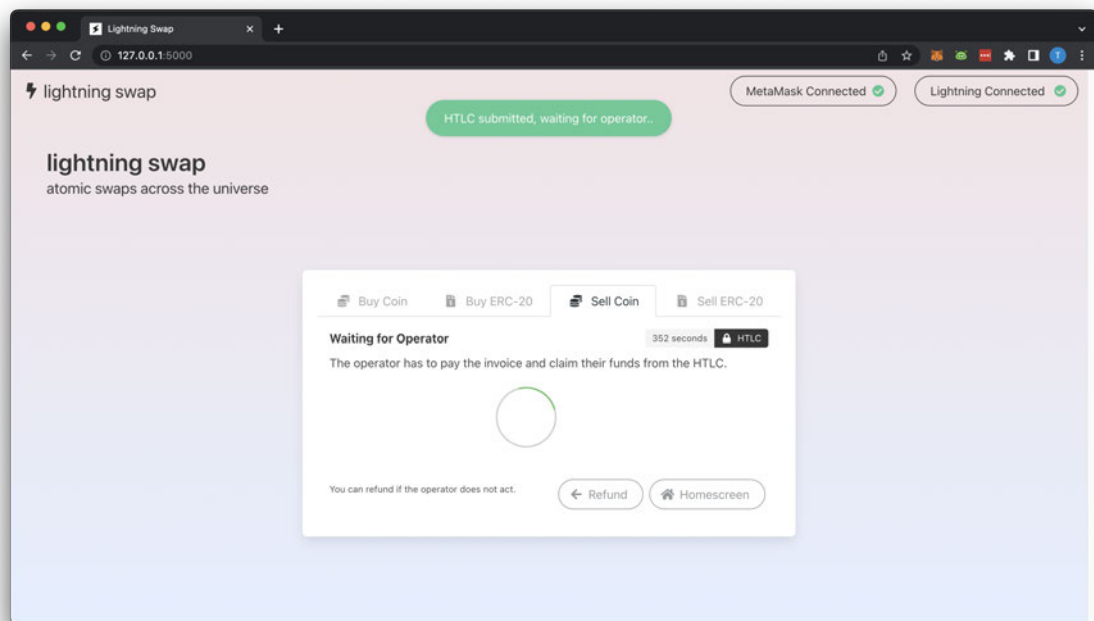


Figure 4.9: User Interface: Waiting for Operator

After the operator successfully pays the lightning invoice using LNbits, they receive the preimage of the invoice. This preimage allows the operator to claim their funds from the HTLC by calling the `withdraw` function of the smart contract and providing the `contractId` and the preimage. The user interface listens for an event on the blockchain, signaling that the operator claimed their funds. Once the operator has claimed their funds, the user interface informs the customer accordingly, and the swap is considered complete. The operator has received their coins, and the customer has received their Bitcoin on the Lightning Network. However, if the operator does not act within the specified timeout period, the customer must refund their funds from the HTLC. This can be achieved by clicking the *Refund* button on the user interface and confirming the transaction in the Metamask prompt. This transaction calls the `refund` function of the smart contract, effectively transferring the coins back to the customer.

4.5 ERC-20 Token - Lightning

The customer can also sell ERC-20 tokens for Bitcoin on the Lightning Network. The protocol, illustrated in figure 4.10, differs slightly from selling native coins. The necessary changes are discussed in detail in this section, while aspects of the protocol that remain unchanged from selling native coins are only briefly addressed.

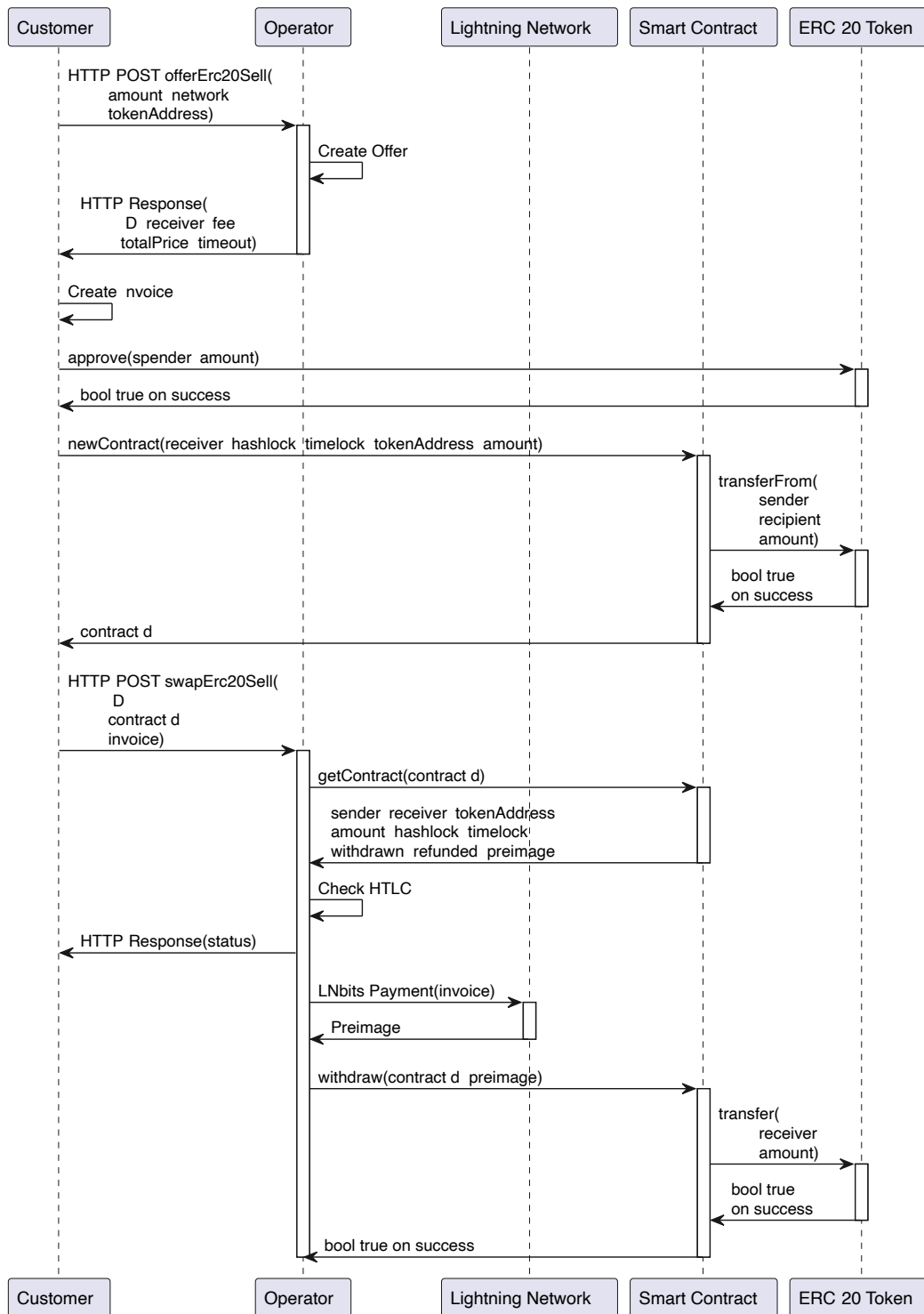


Figure 4.10: Protocol: ERC-20 Token - Lightning

In the process of selling ERC-20 tokens for Bitcoin on the Lightning Network, the customer starts by choosing which ERC-20 token they want to sell from the drop-down menu. Afterward, they choose the amount they wish to sell. Upon selecting the desired token and amount, the customer initiates the process by sending an HTTP POST request to the `offerErc20Sell` endpoint of the operator. This request contains an extra parameter, the address of the desired ERC-20 token, compared to the request for selling native coins.

Like the protocol for selling native tokens, the operator obtains the current exchange rate between the ERC-20 token and Bitcoin and calculates the price they are willing to pay for the number of tokens the customer wants to sell. Notably, the price is lower than the actual exchange rate due to the operator's fee. The operator sends the offer to the customer in response to the HTTP request, which includes the offer identifier, the operator's address, the fee, the amount of Bitcoin the customer will receive for their ERC-20 tokens, and the offer's timeout period. Upon receiving the offer, the customer reviews it in the user interface and creates a lightning invoice using Alby. As shown in figure 4.8, the customer only needs to click the *Confirm* button in the Alby prompt to create the lightning invoice.

The subsequent step in the protocol differs from the one for selling native coins. As previously stated, ERC-20 tokens cannot be sent along in a transaction, and thus, the `approve` and `transferFrom` functions of the ERC-20 token have to be used. Therefore, before creating a new HTLC, the customer must authorize the smart contract to transfer their token (the number of tokens they want to sell). The customer can do so by clicking the *Create Token Allowance* button on the user interface, which can be seen in figure 4.11. This generates a transaction that calls the `approve` function of the ERC-20 token and must be confirmed by the user in the Metamask prompt. After the transaction is successfully processed, the customer can proceed with creating the HTLC.

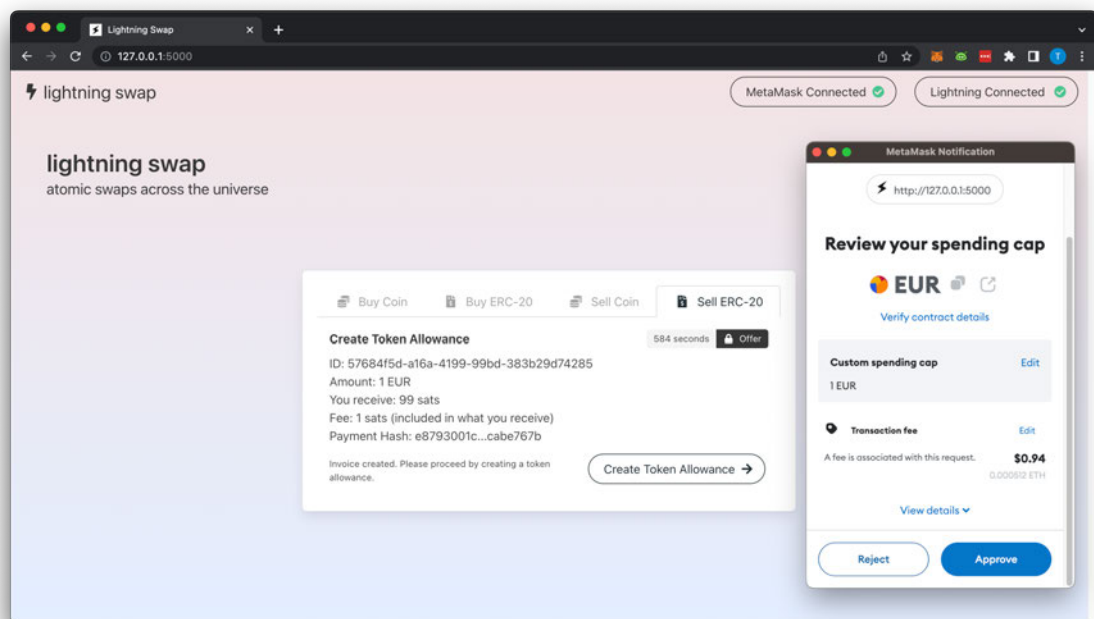


Figure 4.11: User Interface: Approve ERC-20 Token

After the lightning invoice and the allowance for the smart contract have been created, the `newContract` function of the smart contract is used to create a new HTLC. During the creation of the HTLC, the smart contract utilizes the `transferFrom` function of the ERC-20 token to transfer the token to the smart contract and lock it up in the HTLC. If the HTLC creation is successful, the customer receives a unique identifier for the HTLC, referred to as the *contractId*.

The remaining steps of the protocol for selling ERC-20 tokens for Bitcoin on the Lightning Network largely follow the same steps as selling native coins. The user interface sends an HTTP POST request to the `swapErc20Sell` endpoint of the operator, which contains the offer identifier, the *contractId* of the HTLC, and the lightning invoice. The operator fetches the HTLC from the smart contract using the *contractId* and performs the verification checks described in 3.2.2. If all checks pass, the operator sends a positive status update to the user interface, informing the customer that they must wait for the operator to pay the invoice and claim their funds from the HTLC. Afterward, the operator pays the lightning invoice using LNbits to obtain the preimage of the payment. Knowing the preimage allows the operator to claim their ERC-20 tokens by calling the `withdraw` function of the smart contract. The `withdraw` function leverages the `transfer` function of the ERC-20 token to transfer the tokens to the operator. As a result, the customer received their Bitcoin on the Lightning Network, and the operator received their ERC-20 tokens. However, if the operator does not act in time, as described in the previous section, the customer has to use the `refund` function to claim their funds back.

5 Further Considerations

This chapter serves to provide additional considerations for the protocol as a whole. Firstly, implementing a faucet is described, which could improve customer onboarding by providing customers with a small number of native coins to begin interacting with the protocol. Secondly, the significance of Taro is outlined, as Taro aims to bring stablecoins and assets in general to Bitcoin and the Lightning Network. Thirdly, supporting other wallets beyond MetaMask and Alby is discussed, which would increase accessibility and attract a broader range of users to the platform. Finally, the extension of the protocol to include ERC-721 tokens is described, which would allow for buying and selling non-fungible tokens on the platform.

5.1 Faucet

A faucet is a mechanism in the cryptocurrency space that is used to distribute small amounts of coins or tokens to users. In the early days of Bitcoin, faucets were used to distribute Bitcoin ownership widely. Faucets are an easy way for new users to receive their first coins or tokens without registering at an exchange or any other platform. Nowadays, faucets are widely used to distribute coins and tokens on test networks. This allows developers and new users to start using the network without much effort, thus facilitating the process of testing and experimentation. [28]

Creating transactions on any EVM network requires a certain amount of gas fee to be paid in the coin of the corresponding network, such as ether in the case of Ethereum, as explained in section 2.1.2. Customers wishing to use the platform introduced in chapter 4 are expected to create transactions for claiming their funds or creating new hashed timelock contracts (HTLCs). Therefore, customers must have certain coins on the desired network to cover the gas fees. While customers already using the EVM ecosystem should have some coins, this may not be the case for new customers.

The entry barrier for new customers is relatively high, as they need to acquire coins on their desired network before using the presented platform. Typically, this requires much effort as they need to register at an exchange or a similar service. Furthermore, they only need a small amount of coins to be able to pay for their gas fees. To address this issue and make the platform easier accessible for new users, the platform could be extended by a faucet operated by the operator that sells small amounts of coins on the supported networks. New customers could use this service to acquire small amounts of coins on their desired network to start using the platform immediately, lowering the entry barrier for new customers.

The faucet could be included in the user interface, as shown in figure 5.1. This would allow customers to paste their address and the desired amount they want to receive. The user interface would suggest a specific amount based on the current network conditions for the customer to be able to pay the gas fees for at least one transaction (e.g., a transaction that executes the `withdraw` function). The operator would fetch the current exchange rate

between the coin and Bitcoin, following which a lightning invoice would be created for the requested coins' value. After the customer paid the invoice, the operator would send the desired amount of coins to the customer.

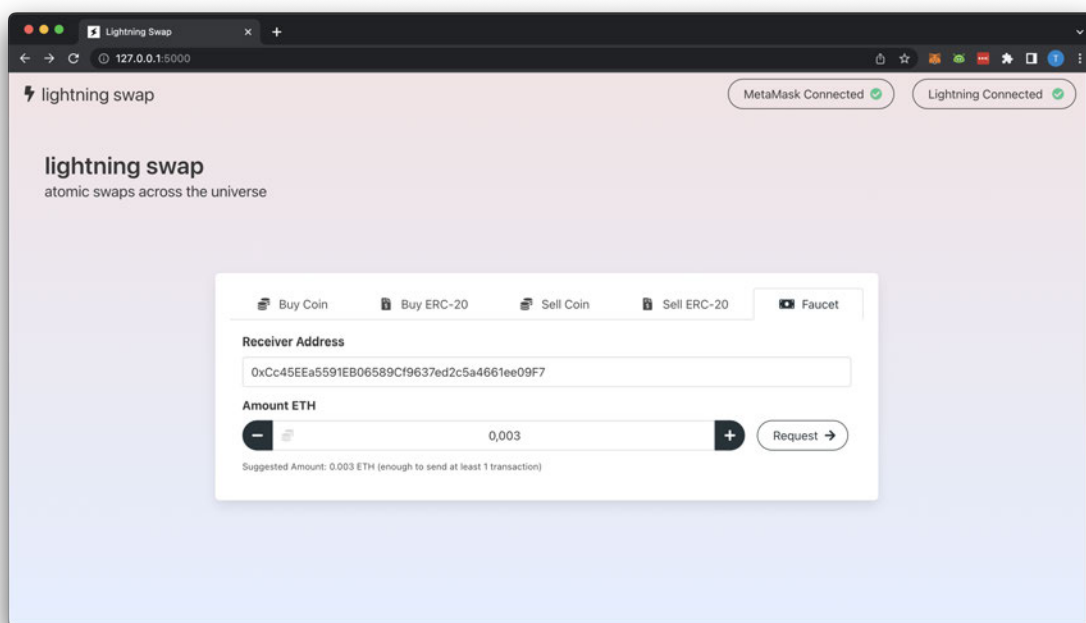


Figure 5.1: User Interface: Faucet

This proposed solution of including a faucet to acquire small amounts of coins has one downside in terms of trust. The customer must fully trust that the operator will send the coins after paying the lightning invoice. However, there are several reasons why this solution makes sense. Firstly, it is optional for the customer and only serves as a convenience. Secondly, the faucet's number of coins is usually minimal, i.e., only a few cents. Even if the operator acts dishonestly, the customer will lose only a tiny amount. Thirdly, if the operator cheats on their customers, their reputation would be damaged, leading to loss of future customers and earnings. Thus, the operator is incentivized to behave honestly. While the faucet does require trust from the customer, it is essential to note that the rest of the platform remains trustless.

The main goal of the faucet is to simplify the process of obtaining small amounts of coins for new customers. By offering this service, customers can start using the platform immediately, even if they are unfamiliar with the ecosystem. Once they have acquired enough coins to pay the gas fees for a single transaction, they can purchase additional coins in a trustless manner, as described in section 4.2. They only require enough coins to execute the `withdraw` function and claim their funds from the hashed timelock contract (HTLC).

5.2 Taro

Taro is a newly introduced Taproot-powered protocol that enables the issuance of assets on the Bitcoin blockchain that can be transferred over the Lightning Network. The platform introduced in chapter 4 takes advantage of the security and stability of the Bitcoin network

as well as the speed, scalability, and low fees offered by Lightning. Taro employs Taproot, the most recent upgrade of Bitcoin, to create a tree structure that allows developers to embed arbitrary asset metadata within an existing output. The assets issued on Taro can be diverse and can represent stablecoins, tickets, shares, art, or ownership rights, among others. Depositing Taro assets into Lightning Network payment channels enables users to hold a balance in their wallet different from BTC, such as a stablecoin, and transact instantly. At the time of writing, Taro's development focuses primarily on the stablecoin use case. [29]

A stablecoin is a cryptocurrency designed to maintain a stable value relative to another asset or group of assets, such as a fiat currency or commodity. This is achieved by pegging the value of the stablecoin to the value of the chosen asset or assets. Stablecoins offer several benefits, such as holding and transferring value without being subject to the volatility of other cryptocurrencies and offering a potential alternative to traditional payment systems. There are several types of stablecoins, including fiat-collateralized, crypto-collateralized, and algorithmic stablecoins. [30]

As of today, the stablecoin ecosystem predominantly relies on EVM-compatible blockchains. At the time of writing, stablecoins contribute to approximately ten percent of the global cryptocurrency market capitalization. Most stablecoin market capitalization, i.e., approximately 55%, is based on Ethereum. The remaining is distributed over various other blockchain networks like Tron and Polygon. The widespread use and adoption of stablecoins demonstrate their significant influence and importance in the broader cryptocurrency ecosystem. [31]

The development of the Taro protocol has brought the prospect of stablecoins on the Bitcoin blockchain and the Lightning Network. This can be seen as a milestone because the stablecoin ecosystem is currently predominantly based on EVM-compatible blockchains. This development may result in a demand for transferring stablecoins from an EVM-based blockchain to the Lightning Network. The platform developed in this work could be an alternative for users not wanting to use a centralized exchange. It offers a solution for users to transfer their stablecoins quickly, cheaply, and in a trustless manner.

The Lightning Network and EVM-compatible blockchains offer distinct advantages in the crypto space. While the Lightning Network is well-suited for instant, low-cost payments for everyday goods, the EVM ecosystem provides users with a wide range of decentralized applications. As such, users may want to swap funds back and forth between the two ecosystems depending on their needs. This creates a demand for a trustless connection between the Lightning Network and EVM-compatible blockchains, which the platform developed in this work aims to provide.

5.3 Supported Wallets

A suite of requisite tools is necessary to interact with an EVM-compatible blockchain and the Lightning Network effectively. The implementation presented in chapter 4 utilizes the browser add-on Metamask to interact with EVM-compatible blockchains, while the browser

extension Alby is used for the interaction with the Lightning Network. However, customers who do not have Metamask or Alby installed cannot use the platform. Thus, supporting multiple wallet options would increase accessibility and attract more users to the platform.

There are a variety of wallets available for interaction with EVM-compatible blockchains. *WalletConnect* can be utilized to support multiple wallets without the need for individual implementation of each wallet. It offers a toolkit that enables the connection of decentralized applications (dApps) to multiple blockchain wallets. At the time of writing, more than 170 different wallets are supported. By integrating WalletConnect, only this one integration is required to support all these wallets. [32]

WalletConnect provides support for mobile wallets, desktop wallets, and hardware wallets. For mobile wallets that support WalletConnect, the customer scans the QR code shown in figure 5.2 to connect their wallet. After a successful connection, the customer can confirm transactions in their mobile wallet. Similarly, desktop and hardware wallets can also be connected. The benefit of leveraging WalletConnect is that the customers are not restricted to using a specific wallet, such as MetaMask, but still have access to the platform presented in chapter 4.

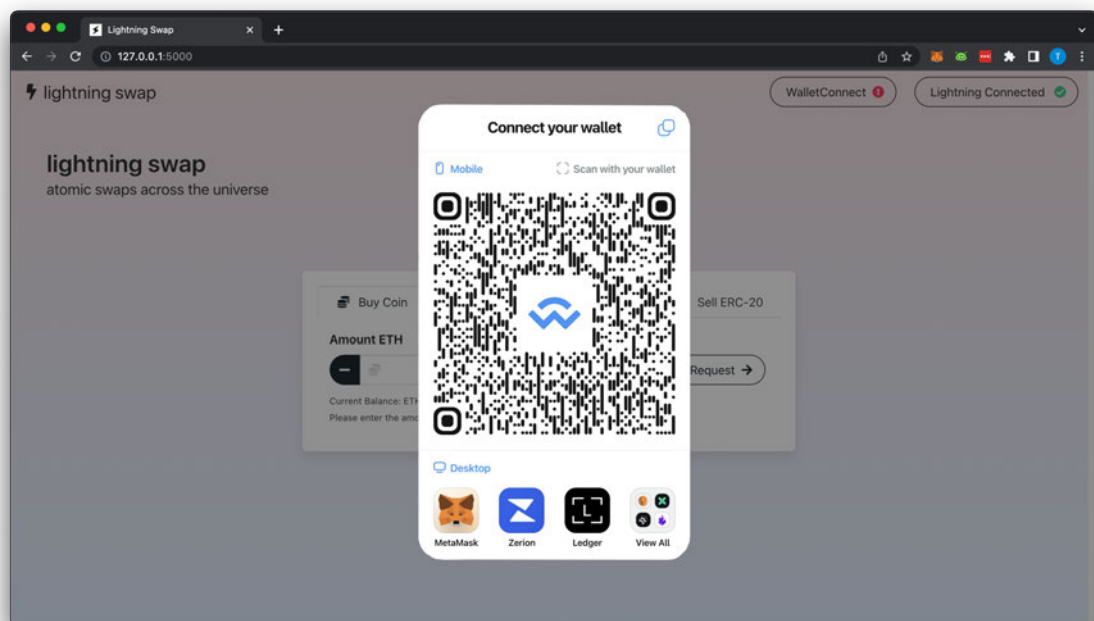


Figure 5.2: User Interface: Wallet Connect [33]

The implementation presented in chapter 4 utilizes the WebLN standard for interaction with the Lightning Network, which is explained in detail in section 2.2.3. Alby, a browser-based lightning wallet, is used for the implementation, as it supports the WebLN standard. Additionally, other wallets, including Joule, Kollider, and BlueWallet, also support the WebLN standard. Customers without a wallet that supports WebLN cannot use the platform and would need to set up a corresponding wallet first, which can be a significant obstacle. Therefore, supporting all widely used lightning wallets would reduce the entry barrier for customers to use the platform. [18]

In order to purchase native coins and ERC-20 tokens on the platform, customers must pay lightning invoices. The current implementation leverages the WebLN sendPayment method, which prompts users in their Alby wallet to confirm the payment. The user interface must display the lightning invoice to support other wallets, as shown in figure 5.3. For customers with mobile wallets, the lightning invoice can be paid by scanning the displayed QR code. Alternatively, the customer can copy the lightning invoice into their clipboard and paste it into their desired wallet to pay the invoice. The lightning invoice can now be paid using any lightning wallet.

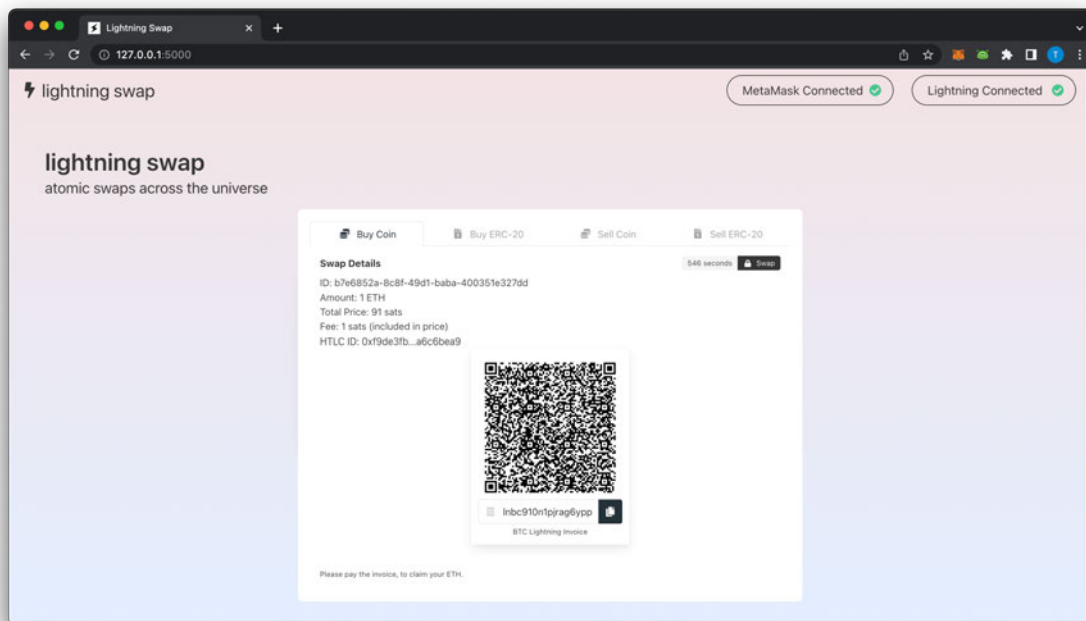


Figure 5.3: User Interface: Display Lightning Invoice

After paying the lightning invoice, the customer must claim their funds from the hashed timelock contract (HTLC) as described in section 4.2. The preimage from the successfully paid lightning invoice is required to claim the funds. Hence, the user must manually copy the preimage from the lightning wallet they used to pay the invoice into the user interface. Typically, the preimage is revealed in the payment details following a successful payment. However, this process can lead to a poor user experience, as the customer must manually handle the preimage. The customer may also have to use third-party software to transfer the preimage between their lightning wallet and the user interface. In contrast, the WebLN standard simplifies the process significantly, as the customer only needs to click one button to confirm the payment. The preimage is automatically handled programmatically in the background.

Selling native coins and ERC-20 tokens requires the customer to create a lightning invoice, which can be done programmatically using the makeInvoice method in WebLN. The customer only needs to confirm the creation of the invoice by clicking one button. However, supporting any lightning wallet means the customer must manually create the lightning in-

voice and paste it into the user interface. Similar to buying native coins and ERC-20 tokens, this creates a poor user experience for selling, especially when compared to the seamless experience offered by WebLN.

Supporting all commonly used lightning wallets can be accomplished with some modifications to the implementation. However, this approach has one significant drawback: the user experience will be notably worse than using the WebLN standard. Moreover, there is an increased probability of errors during the manual copy-pasting of the preimage to the user interface or while manually creating the lightning invoice. With the WebLN standard, the user does not need to perform manual actions but only confirm payments and invoice creations with a button click. The rest is handled in the background, invisible to the customer. Therefore, careful consideration must be given to determine if supporting other lightning wallets is worth sacrificing the seamless user experience offered by the WebLN standard.

In order to increase accessibility and attract a broader range of customers to the platform, supporting other wallets besides Metamask and Alby is a valuable consideration. WalletConnect is particularly useful in this regard, as it allows customers to use not only Metamask but also a variety of other wallets for interacting with EVM-compatible blockchains. The implementation of WalletConnect is straightforward and provides substantial benefits. Therefore, it should be considered for future development of the platform. However, the support of lightning wallets apart from WebLN-compatible wallets must be well considered. Although such support would offer advantages in terms of accessibility to customers without WebLN-compatible wallets, it would also result in a significantly worse user experience due to the need for manual intervention. Hence, it may be preferable to onboard customers to WebLN-compatible wallets to retain a seamless user experience. For instance, in Alby, existing wallets can be added, allowing customers to continue using their current wallet while also being able to access the platform with the WebLN-compatible Alby wallet.

5.4 ERC-721

Like ERC-20 tokens, ERC-721 tokens are a type of digital asset that exists on a blockchain like Ethereum. However, while ERC-20 tokens are fungible, meaning each token is interchangeable with another token of the same type and value, ERC-721 tokens are non-fungible, meaning each token is unique and has its distinct value. Each token can be traced back to its original owner, making it ideal for representing assets such as digital art, collectibles, and event tickets. [34]

The platform introduced in chapter 4 can be extended to support ERC-721 tokens. Since these tokens are similar to ERC-20 tokens, only a few modifications to the smart contract are required. Supporting ERC-721 tokens provides a range of use cases and makes the platform more appealing to a broader audience. For instance, if an event issues tickets on the Ethereum blockchain, one can purchase a ticket using the Lightning Network in a fast and trustless manner.

The implementation of hashed timelock contracts (HTLCs) presented in section 4.1 can be extended to support ERC-721 tokens by making a few modifications to the smart contract. An ERC-721 token can be identified by its unique identifier and the token address. Consequently, the struct of the HTLC must be adjusted to accommodate these changes. The struct no longer needs to store the `amount` since ERC-721 tokens are always unique, and an HTLC handles one token. To specify the token, the token address and identifier must be stored. The updated struct can be seen in code snippet 5.1. The sender and receiver of the HTLC remain unchanged, as well as the other properties of the struct, which are explained in detail in section 4.1. [27]

```
1 struct LockContract {
2     address sender;
3     address receiver;
4     address tokenContract;
5     uint256 tokenId;
6     bytes32 hashlock;
7     uint256 timelock;
8     bool withdrawn;
9     bool refunded;
10    bytes32 preimage;
11 }
```

Listing 5.1: HTLC Struct for ERC-721

The token address and the unique identifier must be passed to the `newContract` function when creating the HTLC. Like ERC-20 tokens, ERC-721 tokens cannot be sent along with a transaction. Therefore, the smart contract must collect the token from the creator of the HTLC when creating the HTLC. To accomplish this, the ERC-721 standard also provides the functionality to spend one's token. Thus, the creator of the HTLC must grant permission for the smart contract to transfer their token. Upon creating the HTLC, the smart contract transfers the token to the smart contract, effectively locking it up in the HTLC. The `withdraw` and `refund` functions remain mostly unchanged from those for ERC-20 tokens. To transfer the token, the smart contract must call the `transfer` function of the corresponding ERC-721 token contract.

The platform's user interface needs to be modified to support ERC-721 tokens. A crucial aspect of this modification is enabling customers to select a specific ERC-721 token. This can be done by including options for entering the token address and unique identifier, similar to how ERC-20 tokens are selected. Since the underlying mechanism of the platform remains the same for ERC-721 tokens, most of the user interface can remain unchanged.

In the context of this work, the main focus has been on native coins and ERC-20 tokens. However, it is essential to note that the presented platform is versatile and can be extended to support ERC-721 tokens as well. As explained in this section, supporting ERC-721 tokens opens up a variety of use cases and expands the potential user base of the platform.

6 Conclusion

In this work, a trustless connection between the Bitcoin Lightning Network and EVM-compatible blockchains has been developed, enabling asset transfer between the two ecosystems. The proposed solution is highly versatile, as it can establish a connection between Bitcoin Lightning and any blockchain that supports the EVM. After describing the necessary fundamentals, two concepts were proposed to achieve this connection. Finally, the protocol for the more promising concept, using atomic swaps, was described in detail and implemented. It is now possible to buy and sell native coins and ERC-20 tokens for Bitcoin on the Lightning Network in a trustless manner. The prioritized objectives for the solution - security, speed, and user experience - have been achieved.

The focus of this work has been on the technical aspects of the solution. However, to operate the developed platform in production, some non-technical aspects still need to be clarified, including economic and security considerations. To ensure the platform can operate adequately, enough liquidity must be available on both the Bitcoin Lightning Network and the EVM-compatible blockchain side. Furthermore, a rebalancing mechanism is required to prevent situations where customers are unable to complete their trades due to a lack of available assets on one side. Additionally, the development of a suitable monetization model is a crucial aspect of operating the platform sustainably. One possible option is to charge a fee for every swap, such as a percentage of the swapped amount. However, the exact fee structure must be carefully considered to ensure that it remains competitive in the market and remains attractive to customers. Finally, an in-depth analysis of the costs of using the platform for both the customer and operator is required. The costs depend on the payment route on the Lightning Network and the level of demand on the corresponding EVM blockchain.

The security of atomic swaps relies on the use of hashed timelock contracts (HTLCs) which ensure that neither party can cheat. The implementation of HTLCs in Solidity is straightforward and not very complex, which contributes to the high level of security of this solution. Furthermore, the smart contract never holds a large number of funds, only the funds that are involved in an ongoing swap. Thus, in the event of a security vulnerability in the smart contract, there would only be a small number of funds accessible to a potential attacker. Apart from the smart contract, the operator has to consider common security-relevant aspects when running a service, such as denial of service (DoS) protection.

This work provides a foundation for developing a production-ready service that enables a trustless connection between the Bitcoin Lightning Network and EVM-compatible blockchains using atomic swaps. Initially, the service could begin with a limited scope, e.g., by supporting only native coins on Ethereum. Once the service is considered stable, it can be expanded to include other networks and ERC-20 tokens. Furthermore, the service can be expanded to include a faucet for better user onboarding, support ERC-721 tokens, and support other wallets, as described in chapter 5. By establishing such a service, this work can contribute to the growth of both ecosystems and open up new possibilities for both of them to benefit from each other's advantages.

Appendix A: Implementation

The implementation can be found on the CD that accompanies this work.

Bibliography

- [1] Ethereum.org. "What is Ethereum?" (2023), [Online]. Available: <https://ethereum.org/en/what-is-ethereum>. (last visted on 09/05/2023).
- [2] Kearney, Leal. "What are EVM Compatible Blockchains? A Guide to the Ethereum Virtual Machine". (2023), [Online]. Available: <https://blog.thirdweb.com/evm-compatible-blockchains-and-ethereum-virtual-machine/>. (last visted on 09/05/2023).
- [3] GoCrypto. "What are EVM-compatible blockchains?" (2022), [Online]. Available: <https://medium.com/eligma-blog/what-are-evm-compatible-blockchains-64f91c97038e>. (last visted on 23/02/2023).
- [4] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". (2008), [Online]. Available: <https://bitcoin.org/bitcoin.pdf>. (last visted on 09/05/2023).
- [5] Joseph Poon, Thaddeus Dryja. "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments". (2016), [Online]. Available: <https://lightning.network/lightning-network-paper.pdf>. (last visted on 09/05/2023).
- [6] Andreas Antonopoulos, Olaoluwa Osuntokun, René Pickhardt, *Mastering the Lightning Network*. O'Reilly Media, Inc., 2021, ISBN: 978-1-492-05486-3.
- [7] Cointelegraph. "What is blockchain interoperability: A beginner's guide to cross-chain technology". (2023), [Online]. Available: <https://cointelegraph.com/learn/what-is-blockchain-interoperability-a-beginners-guide-to-cross-chain-technology>. (last visted on 19/04/2023).
- [8] Leon Do. "submarine-swaps". (2020), [Online]. Available: <https://github.com/leon-do/submarine-swaps>. (last visted on 26/04/2023).
- [9] Aleksey Bykhun. "ln-eth-swap". (2018), [Online]. Available: <https://github.com/cafeinum/ln-eth-swap>. (last visted on 26/04/2023).
- [10] Andreas Antonopoulos, Dr. Gavin Wood, *Mastering Ethereum*. O'Reilly Media, Inc., 2018, ISBN: 978-1-491-97194-9.
- [11] Ethereum.org. "Sidechains". (2013), [Online]. Available: <https://ethereum.org/en/developers/docs/scaling/sidechains/>. (last visted on 09/05/2023).
- [12] Data Conomy. "Unlocking the secrets of the blockchain nonce". (2022), [Online]. Available: <https://dataconomy.com/2022/12/15/blockchain-nonce-explained/>. (last visted on 09/05/2023).
- [13] Bitcoin Visuals. "Lightning Network Capacity". (2023), [Online]. Available: <https://bitcoinvisuals.com/ln-capacity>. (last visted on 09/03/2023).
- [14] Blocktrainer. "Bitcoin zum Anfassen: Der Lightning-Snackautomat". (2023), [Online]. Available: <https://www.blocktrainer.de/bitcoin-zum-anfassen-der-lightning-snackautomat>. (last visted on 09/03/2023).
- [15] Coincharge. "Payment per newspaper article with Bitcoin Lightning". (2023), [Online]. Available: <https://coincharge.io/en/payment-per-newspaper-article-with-bitcoin-lightning>. (last visted on 09/03/2023).

- [16] LNbits. "Free Open-Source Bitcoin Lightning Accounts System with Extensions". (2023), [Online]. Available: <https://lnbits.com>. (last visted on 09/03/2023).
- [17] web3.js. "web3.js - Ethereum JavaScript API". (2023), [Online]. Available: <https://web3js.readthedocs.io/en/v1.8.2/>. (last visted on 10/05/2023).
- [18] Alby. "WebLN Guide". (2023), [Online]. Available: <https://github.com/getAlby/webln-guide>. (last visted on 06/03/2023).
- [19] Alby. "lightning-browser-extension". (2023), [Online]. Available: <https://github.com/getAlby/lightning-browser-extension#lightning-web-extension>. (last visted on 06/03/2023).
- [20] Joule Labs. "WebLN Developer Documentation". (2023), [Online]. Available: <https://webln.dev>. (last visted on 07/03/2023).
- [21] Minima. "Understanding Hashed Time-locked Contracts (HTLCs)". (2022), [Online]. Available: <https://www.minima.global/post/understanding-hashed-time-locked-contracts-htlcs>. (last visted on 17/03/2023).
- [22] Lightning Labs. "Hashed Timelock Contract (HTLC)". (2023), [Online]. Available: <https://docs.lightning.engineering/the-lightning-network/multihop-payments/hash-time-lock-contract-htlc>. (last visted on 17/03/2023).
- [23] Muhammad Zubair. "How is SHA-256 used in blockchain, and why?" (2023), [Online]. Available: <https://www.educative.io/answers/how-is-sha-256-used-in-blockchain-and-why>. (last visted on 17/03/2023).
- [24] Liquid. "Crypto Coin vs. Token: Understanding the Difference". (2023), [Online]. Available: <https://blog.liquid.com/coin-vs-token>. (last visted on 22/03/2023).
- [25] Tether. "Tether Token USDT". (2023), [Online]. Available: <https://tether.to/en>. (last visted on 22/03/2023).
- [26] Truffle Suite. "Ganache - One Click Blockchain". (2023), [Online]. Available: <https://trufflesuite.com/ganache/>. (last visted on 10/05/2023).
- [27] C. Hatch. "hashed-timelock-contract-ethereum". (2021), [Online]. Available: <https://github.com/chatch/hashed-timelock-contract-ethereum>. (last visted on 26/04/2023).
- [28] Binance Academy. "Crypto Faucet". (2022), [Online]. Available: <https://academy.binance.com/en/articles/what-is-a-crypto-faucet>. (last visted on 13/04/2023).
- [29] Lightning Labs. "Taro". (2023), [Online]. Available: <https://docs.lightning.engineering/the-lightning-network/taro>. (last visted on 14/04/2023).
- [30] Investopedia. "Stablecoins". (2022), [Online]. Available: <https://www.investopedia.com/terms/s/stablecoin.asp>. (last visted on 14/04/2023).
- [31] DefiLlama. "DefiLlama - Stables - Chains". (2023), [Online]. Available: <https://defillama.com/stablecoins/chains>. (last visted on 14/04/2023).
- [32] WalletConnect Docs. "What is WalletConnect?" (2023), [Online]. Available: <https://docs.walletconnect.com/2.0>. (last visted on 17/04/2023).
- [33] WalletConnect Twitter. "advertisement Post". (2023), [Online]. Available: <https://twitter.com/WalletConnect/status/1636427862096969735>. (last visted on 17/04/2023).
- [34] OpenZeppelin Docs. "ERC721". (2023), [Online]. Available: <https://docs.openzeppelin.com/contracts/3.x/erc721>. (last visted on 17/04/2023).

Eidesstattliche Erklärung

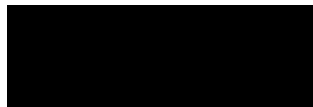
Hiermit versichere ich – Tim Käbisch – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Mittweida, 14.05.2023

Ort, Datum



Tim Käbisch