Portal Network: Swap Protocol

Portal Engineering & Research
San Francisco, California

2023-09-14

Abstract

Portal Swap Protocol is designed to enable any user to perform trustless and censorship resistant swaps of crypto assets on different blockchains without using wrappers, bridges or other custodian based solutions. Portal uses customized HTLC contracts or equivalent smart contracts on different blockchains to accomplish cross-chain atomic swaps. The architecture is designed modularly so that clients can use any of the following combinations : Layer 1 to Layer 1 swaps, Layer 1 to Layer 2 swaps, or Layer 2 to Layer 2 swaps on different blockchain networks. Portal Swaps are designed to be modular and independent of the underlying matching protocol, i.e. Peer-to-Peer or a Cross-chain AMM and thus maintain atomicity regardless of how the maker and taker are connected. We have also identified several race conditions and designed the protocol with proper incentives to mitigate some of the issues like lockup griefing etc. In this paper we focus exclusively on how swap contracts are designed and implemented in the Portal network and we illustrate several temporal flows using Lightning to Ethereum as an example but can be applied across any network that supports SHA256 hashing functionality.

Basics of Atomic Swaps

Atomic swaps represent peer-to-peer trading methods employed to transfer cryptocurrencies between distinct blockchains without relying on trusted third parties.

An atomic swap protocol ensures the following:

- When all parties adhere to the protocol, the swap occurs as intended.
- If any party deviates from the protocol, no compliant participant suffers negative consequences.
- No group has economic incentives to deviate from the protocol.

Atomic swaps make use of Hashed Time Lock Contracts (HTLCs), which are a category of smart contracts designed to facilitate a trustless exchange of digital assets. Smart contracts employ an automated process that self-executes once all predefined conditions embedded within the contract are met.

Bitcoin atomic swaps are possible thanks to two key components encoded in Bitcoin's HTLCs:

Hashlock

The hashlock mechanism enables the contract to be secured with a distinct cryptographic key that can solely be created by the depositor of the cryptocurrency. This unique key serves as a guarantee that the swap is completed when the party with the key (secret, pre-image) gives their consent to the transaction.

Timelocks

The time lock mechanism can be likened to a time limit for the swap. It guarantees that the transaction is finalized within a predefined timeframe, and if this condition is not met, it makes it possible for the depositor to reclaim their funds. Timelock plays a crucial role in ensuring the security of the swap transactions. It mandates that both parties must execute the swap within the stipulated time window for a successful swap

Example of a Simple Atomic Swap Flow

Alice and Bob agree to exchange 10 X tokens (on x blockchain) for 10 Y tokens (on y blockchain) using a Hashed Time Lock Contract (HTLC) set to expire in one hour.

First, Alice creates a contract address on x chain and deposits her 10 X tokens into it, after generating a private key accessible only to her, with the stipulation that anyone with the key can unlock and spend from the contract, within a specified time period, after which she can reclaim the funds. She then generates a cryptographic hash of this private key and sends it to Bob. Bob uses this hash to confirm that Alice has indeed deposited 10 X tokens into the contract address. However, Bob cannot access the funds since he possesses only the hash, not the actual private key.

Bob uses the hash to create a new contract address where he deposits his 10 Y tokens on the Y blockchain, with the stipulation that anyone with the secret key can spend from the contract, for a period of time which is less than the timelock in Alice's contract on X. Now, both parties have contributed their funds to the contract. Because Bob crafted the address using the hash of Alice's private key, Alice can claim the 10 Y tokens deposited by Bob. To do this, she reveals the private key on the y blockchain, which presumably Bob monitors and obtains. He then has a period of time to use that key to spend X tokens from Alice's contract to his own address. If Bob fails to complete the transaction between the time when Alice reveals the key on y and before the timelock expires on x, Alice can spend those coins back to herself.

If the swap is completed within the specified timeframe, the contract is successfully executed and Alice successfully swaps her 10 X tokens with Bob in exchange for his 10 Y tokens.

Atomic swaps vs. bridges

Atomic swaps enable peer-to-peer exchanges, allowing users to swap cryptocurrencies directly without the need for intermediaries. On the other hand, cross-chain bridges establish a link between different blockchain networks, enabling the seamless transfer of digital assets through tokenized representations.

While both cross-chain bridges and atomic swaps contribute to improving blockchain interoperability and facilitate the movement of cryptocurrencies across various blockchain platforms, they operate differently. Cross-chain bridges act as connectors or intermediaries that facilitate asset transfers among multiple blockchain networks.

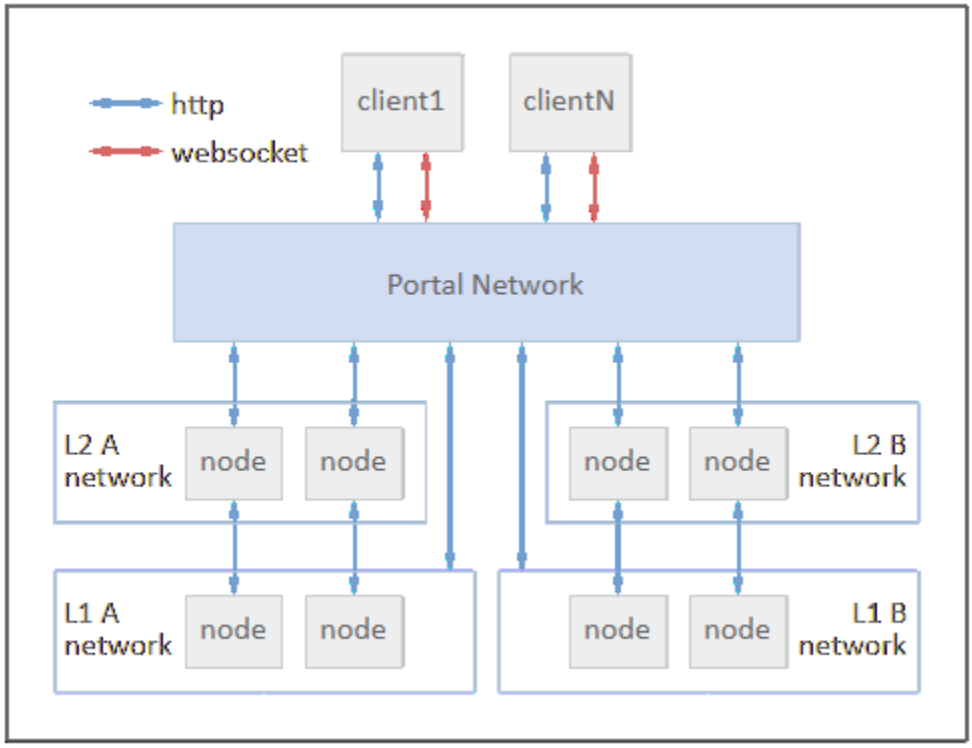
To use cross-chain bridges, a token must be locked on its original blockchain, after which a corresponding wrapped token is generated on the target chain. This wrapped token is then deposited into a liquidity pool on the target blockchain, making it accessible for trading, transfer, or redemption for the original assets from the source blockchain. Note that the two blockchains are independent, internally consistent systems that do not depend on external data to maintain or alter their contract states or order of transactions i.e, there is no enforcement of the peg or redeemability at the protocol level on either chain. Usually, either a single entity or a group of entities "guarantees the trust" in the peg between the wrapped token and the original token and its redeemability. There have been several instances of wrapped token attacks and

failures and there is a need for atomicity as opposed to bridges, which is why Portal chose the harder but right way of doing swaps.

Peer-to-Peer Swaps (V1)

V1 of the Portal Network executes atomic, non-custodial exchange of BTC with ETH, ERC20, and other currencies on L1s and L2s. The Network is composed of nodes/servers that act as an intermediary between multiple clients and multiple L1 and/or L2 networks as depicted in Figure 1 below.

Figure1: P2P Portal Swaps



Clients interface with a node using both HTTP and websocket connections. Clients create limit orders which are stored in an orderbook managed by the network. When two limit orders match (currently only exact matches are supported), both counterparties are notified via websocket whereupon they may then create a swap order whose lifecycle is managed by a node. The node sends messages to both counterparties at each phase of the swap lifecycle.
Atomic, non-custodial swaps are implemented using Hashed Time-Locked Contracts (HTLC's) residing on the L1's or L2's supporting the respective currencies being exchanged, i.e. there is an HTLC on each participating L1/L2. HTLC's contain the hash of a secret bit string (the "secret hash") and are unlocked either when presented with the secret that originated the hash or after a time limit has expired.

When creating a swap order, each party supplies a secret hash and the node selects one of these for constructing both HTLC's. The party whose secret is chosen is the "secret holder" and the other party is the "secret seeker". The secret holder is responsible for activating the HTLC using their in-the-clear, i.e. un-hashed, secret. Doing so reveals the secret to the secret seeker which may then activate their HTLC.

HTLC's are also constructed with a time limit by which they must be activated. After the time limit expires, the HTLC can no longer be unlocked with a secret and each party may retrieve their original currency any time thereafter.

The following sections describe the temporal sequence of a swap lifecycle and the specifics of the HTTP and websocket interfaces.

Example Swap Flow

1. SecretKnower/Maker initiates the process by generating a random 32-byte secret, which is then hashed to create hashOfSecret. An order is subsequently generated using this hashOfSecret.
2. SecretSeeker/Taker enters the picture by matching the order. Following this, SecretKnower/Maker establishes a locked deposit for SecretSeeker, utilizing the same hashOfSecret.
3. SecretSeeker/Taker acknowledges the deposit and, in response, generates a counter-deposit on the opposing chain, also utilizing the hashOfSecret lock.
4. Upon detecting an incoming payment on the opposing chain, SecretKnower/Maker proceeds to claim the original payment from SecretSeeker/Taker. Simultaneously, SecretKnower/Maker reveals the secret to SecretSeeker/Taker via chain data.
5. Subsequently, SecretSeeker/Taker employs this same secret to claim the funds on the original chain.

A complete picture of how the temporal flow works is illustrated in Figure 2.

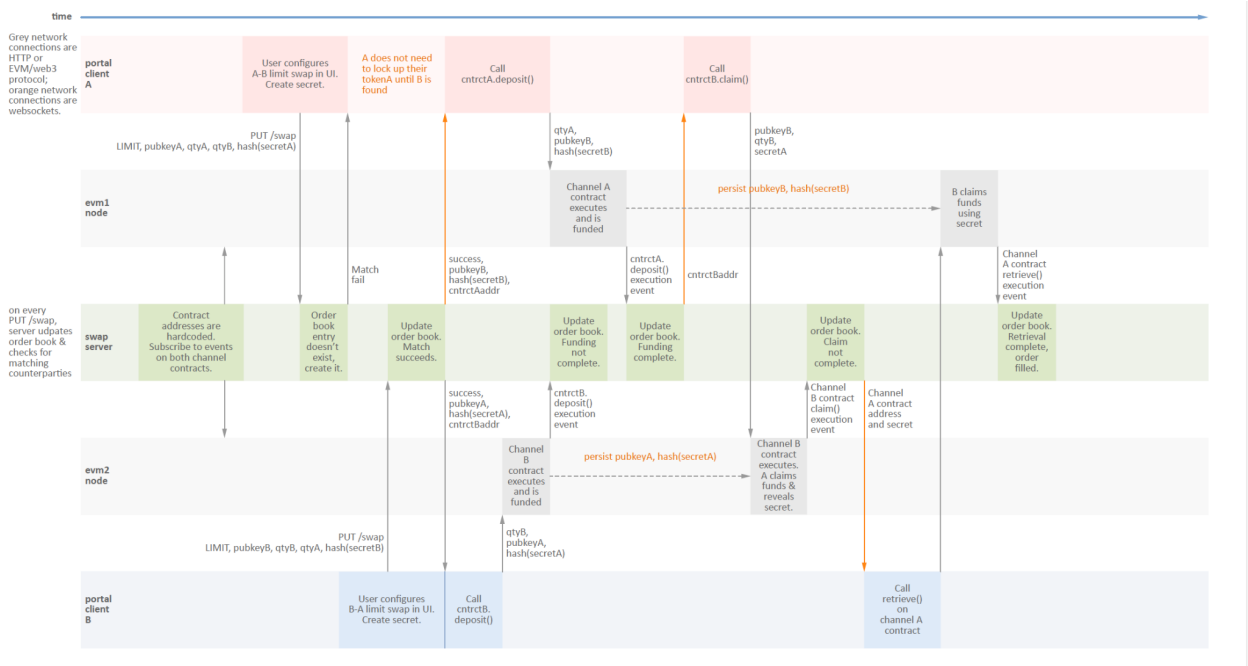


Figure 2: P2P Swap Temporal Flow

Lightning and Ethereum Swaps

A simple swap sequence between Lightning (Bitcoin L2) to Ethereum L1 are as shown below. The Relay node described is used to communicate between clients and provide swap intent matching (like Orderbook) similar to Nostr relay nodes.

```
Unset
actor: Alice
relay node: OrderBook
actor: Bob
—----
Alice -> OrderBook: Place swap order of 15 ETH for 1 BTC)
Bob -> OrderBook: Place swap order of 1 BTC for 15 ETH)
activate OrderBook
OrderBook --> Alice: EventEmitter: swap.matched
OrderBook --> Bob: EventEmitter: swap.matched
Alice -> OrderBook: Opens swap order (and locks 15 ETH in contract)
OrderBook --> Alice: EventEmitter: swap.opening
OrderBook --> Bob: EventEmitter: swap.opening
```

```
Bob -> OrderBook: Opens swap order (and locks 1 BTC in contract)
OrderBook --> Alice: EventEmitter: swap.opened
OrderBook --> Bob: EventEmitter: swap.opened
Alice -> OrderBook: Commits swap order
OrderBook --> Alice: EventEmitter: swap.committing
OrderBook --> Bob: EventEmitter: swap.committing
Bob -> OrderBook: Commits swap order
OrderBook --> Alice: EventEmitter: swap.committed, secret revealed
OrderBook --> Bob: EventEmitter: swap.committed, secret revealed
OrderBook -> Bob: Receives 1 BTC from swap completion
OrderBook -> Alice: Receives 15 ETH from swap completion
```

Lightning to Ordinal Swaps

Alice and Bob are the two parties to the ordinal swap. Alice is selling the ordinal to Bob. Alice sends the ordinal to Bob on L1 and receives payment on L2.

Alice knows a secret and generates a hash of that secret. This is the swap hash. The secret is the swap secret.

On the lightning network, a Bolt 11 invoice is used. Alice creates the Bolt 11 invoice, and Bob pays it later under certain conditions. Alice uses the swap hash as the invoice hash when creating the invoice. The invoice has an expiration time T_1.

On the bitcoin network, an on-chain transaction is used, with an HTLC locking script, modeled on an example by the Bitcoin Studio tutorial on bitcoinjs-lib.

The HTLC locking script is written into a witness script. Bob generates an address based on this witness script. Let’s call this the swap address.

Alice learns the swap address, and to set the swap in motion she sends the ordinal to the swap address using Sparrow, which suits such ordinal sending.

The HTLC in the witness script uses the swap hash and the public keys for both Alice and Bob. This HTLC offers Bob the chance to obtain the bitcoin sent to the swap address, including the ordinal, once Bob learns the swap secret. After a certain time $T_2 > T_1$, Alice is able to get a refund from the swap address. The length of time between T_1 and T_2 must be large enough for Bob to take action in time, once Alice settles the invoice and before Alice can get the refund.

Once Alice has sent the ordinal to the swap address, Bob can check the invoice to see if it's for the proper amount and has an expiration T_1 sufficiently earlier than T_2. After a satisfactory number of confirmations, Bob pays the invoice and waits for Alice to settle the invoice.

Alice settles the invoice, revealing the swap secret to Bob. Bob then has until time T_2 to obtain the ordinal, and accompanying bitcoin. Bob creates a transaction, using the swap secret and her signature to unlock the UTXO Alice had created when paying to the swap address. Bob then broadcasts it. If Alice does not pay the invoice in time, then Bob does nothing.
Alice then gets her ordinal, and accompanying bitcoin, refunded to her after time T_2. Alice creates a transaction, using her signature to unlock the UTXO she had created. Alice then broadcasts it. See below for the swap sequence:

```
Unset
actor: Alice
relay node: OrderBook
actor: Bob
Alice -> OrderBook: Place swap order of 1 BTC (on lightning) for an Ordinal Inscription
OrderBook --> Alice: EventEmitter: swap.creating with swap.id sent
Bob -> OrderBook: Inputs swap.id into swap order
activate OrderBook
OrderBook --> Bob: EventEmitter: swap.created
Bob -> OrderBook: Opens swap order (and locks 1 BTC (on lightning) in contract)
OrderBook --> Alice: EventEmitter: swap.opening
OrderBook --> Bob: EventEmitter: swap.opening
Alice -> OrderBook: Opens swap order (and locks 1 BTC (on lightning) in contract)
OrderBook --> Alice: EventEmitter: swap.opened, receives address to send inscription to
OrderBook --> Bob: EventEmitter: swap.opened
Alice -> OrderBook: Sends the Ordinal Inscription to the receive address
OrderBook --> Alice: Notifies BTC transaction has reached minimum confirmations
OrderBook --> Bob: Notifies BTC transaction has reached minimum confirmations
Bob -> OrderBook: Commits swap order
OrderBook --> Alice: EventEmitter: swap.committed, secret revealed
OrderBook --> Bob: EventEmitter: swap.committed, secret revealed
OrderBook -> Bob: Receives the Ordinal Inscription from swap completion
OrderBook -> Alice: Receives 1 BTC (on lightning) from swap completion
```

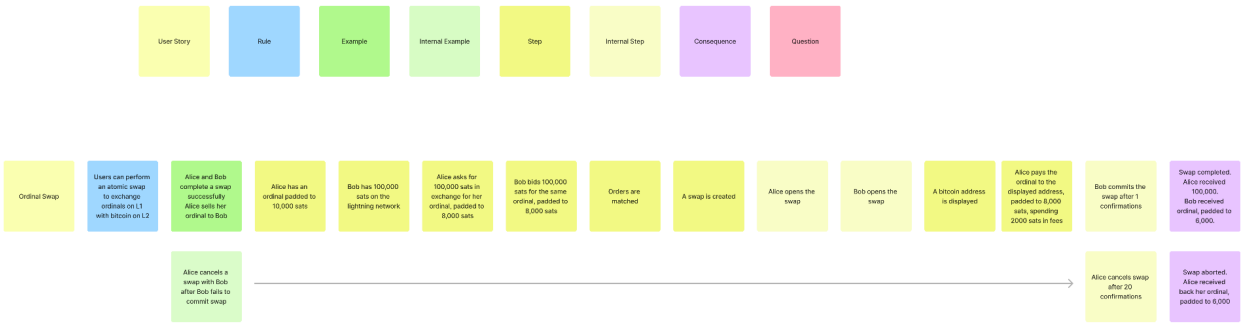


Figure 3: P2P Ordinal Swaps

Bitcoin to Lightning Submarine Swap

On the lightning network, a hodl invoice is used. Bob creates the hodl invoice, and Alice pays it. This is the same as what we use on the lightning side of the lightning<>eth swap. On the bitcoin network, an on-chain transaction is used, with an HTLC locking script that uses an additional pubkey hash check as described by Alex Bosworth.

Below we describe 2 cases of asymmetry between Alice and Bob.

Case 1: Alice is weak, and Bob is strong

1. Bob knows the swap secret and makes the first move to engage money.
2. Bob creates the on-chain tx
3. Bob creates HODL invoice using the hash of swap secret.
4. Bpb sends the invoice request (Bolt 11) and the pubkey for the on-chain tx to Alice.
5. Alice confirms the use of the same hash in both the on-chain tx and in the hodl invoice, and she confirms the amounts.
6. Alice waits for sufficient confirmation of the on-chain tx
7. Alice pays the invoice, gets the swap secret, and uses the secret to unlock the on-chain UTXO.

Case 2: Alice is strong, and Bob is weak

1. Alice knows the swap secret and makes the first move to engage money.
2. Alice sends Bob the swap hash.
3. Bob creates HODL invoice using the swap hash.
4. Alice pays the invoice. Bob cannot settle the invoice yet since he does not know the swap secret.
5. Once the invoice is held, Bob creates the on-chain tx with the same hash and the proper amount.
6. After confirming the hashes and amounts, Alice can now unlock the on-chain UTXO. This reveals the swap secret.
7. Bob uses the swap secret to settle the held invoice.

Exceptions: Areas of Improvement in User Experience

If Alice broadcasts a bitcoin transaction using the HTLC tx before it is confirmed, then there is a risk of Bob double-spending the UTXO fed into the HTLC tx. Alice should wait until there is sufficient confirmation. If there is no confirmation and double-spending occurs, or if the blockchain forks after confirmation, then there needs to be an exception alerting the users.

Examples

Alice doesn’t have enough BTC in her lightning channel

- She decides to deposit X BTC (ignoring fees at the moment)
- Enters 'X' into the deposit form and clicks "deposit"
- A popup emerges with an address for depositing, Alice sends X BTC to the address
- X BTC is deposited via submarine swaps, she is now ready to create a swap

Alice deposits X BTC via web interface

- Alice views balance, sees she does not have enough balance on the lightning channel to create a swap
- Alice clicks deposit, a input form appears, she enters 'X' BTC
- A Popup with a deposit address appears
- Countdown timer is shown below the address, and/or a final deadline block number, is shown to indicate this address is temporary and to be only used for this specific deposit transaction
- Swap confirmation: bitcoin is deposited into Alice’s lightning channel balance and now she is able to swap with using X BTC

```
Unset
actor: Alice
relay node: OrderBook
actor: Bob
Alice -> OrderBook: Place swap order of 1 BTC (on lightning) for 1 BTC on mainnet
OrderBook --> Alice: EventEmitter: swap.creating with swap.id sent
Alice --> Bob: Alice shares swap.id with Bob
Bob -> OrderBook: Inputs swap.id into swap order
activate OrderBook
OrderBook --> Bob: EventEmitter: swap.created
Bob -> OrderBook: Opens swap order (and locks 1 BTC (on lightning) in contract)
OrderBook --> Alice: EventEmitter: swap.opening
OrderBook --> Bob: EventEmitter: swap.opening
Alice -> OrderBook: Opens swap order (and locks 1 BTC (on lightning) in contract)
OrderBook --> Alice: EventEmitter: swap.opened, receives address to send inscription to
OrderBook --> Bob: EventEmitter: swap.opened
Alice -> OrderBook: Sends 1 BTC on mainnet to the receive address
```

```
OrderBook --> Alice: Notifies BTC transaction has reached minimum confirmations
OrderBook --> Bob: Notifies BTC transaction has reached minimum confirmations
Bob -> OrderBook: Commits swap order
OrderBook --> Alice: EventEmitter: swap.committed, secret revealed
OrderBook --> Bob: EventEmitter: swap.committed, secret revealed
OrderBook -> Bob: Receives 1 BTC on mainnet from swap completion
OrderBook -> Alice: Receives 1 BTC (on lightning) from swap completion
```

Failure Mode Scenarios of Swaps

In order to determine the requirements for atomic swaps under exceptional situations, along unhappy paths, this section explores the complex behaviors based on examples.[1]

Initially let's consider examples of behavior involving the following types of swaps

- Bitcoin L2 <> Ethereum L1
- Bitcoin L2 <> Ethereum L2
- Bitcoin L2 <> Bitcoin L1

For Lightning, we assume the LND implementation for simplicity.

The examples below are created to describe some user scenarios which impact security and mitigations, mainly form client side.

Case 1: One party to a lightning swap creates a hodl invoice

No examples known of failures in this scenario

Case 2: One party to a lightning swap pays a hodl invoice

Handle failure or delay

1. When a path cannot be found to the other party's node, alert both parties, cancel the invoice, and stop the swap without fault.
2. When a payment is stuck, alert both parties, and consider allowing either party to cancel the swap without fault. Ensure that payment is not retried until after expiry. Warn the paying party to not retry payment on his own. Do not allow the swap to be recreated until after expiry.
3. Normally payments are made only after both parties have committed and each can no longer cancel the swap. If other exceptional situations arise that are similar to example 2 in that they occur after a payment is made and involve a special cancellation, handle the cancellation in a similar fashion, cognizant of the expiry. One such case may be if a cancellation in example 1 can be stuck.

Case 3: One party to a lightning swap settles a hodl invoice

Handle delay

If the peer in the final HTLC channel does not cooperate and the recipient's lightning node needs to move the HTCL on-chain as a result, alert both parties and accommodate the delay.

Case 4: One party to a lightning swap cancels the swap

Cancel hodl invoice

1. After a swap has been opened and before both commitments, allow the swap to be canceled.
2. When the swap is canceled, cancel the hodl invoice.
3. If cancellation of the hodl invoice gets stuck, notify both parties, and offer to perform a hard shutdown.

Cancel swap activity on ethereum side

After a swap has been opened and then canceled, stop activity and remove state on the ethereum side as necessary.

Case 5: One party to a lightning swap commits the swap

Handle delinquent commits

1. After a swap is opened fully and not canceled, if the swap is not committed within a certain time in line with existing invoice expiry (or anything similar on the ethereum side), then both parties are alerted. Any delinquent party is also notified about how much time is left before the swap is automatically canceled with fault assigned.
2. After an alert is sent, then after another period of time the swap is automatically canceled with fault assigned to one or both parties.

Case 6: When multiple orders are created using the same secret hash

Handle failure

Check that the hash of the secret has not been used previously, should be a newly created hash. Given a user creates an order with a secret hash that had been used on a previous order, then the order is matched on the server, it will receive an event swap.created, for which returns an object including the swap id and secret hash. Then the secret hash will be used to match with previously created orders and statuses to “invalid”.

Case 7: When user exits / clears previously create orders

Then order/swap would not complete successfully

Handle failure

Then the relay node should receive a timeout and cancel the order if no response is received within a specified time period. Given a user creates an order and has refreshed the page, When the order is

matched on the server ,the user will not receive an event swap.created. Then the order would stall and will not be able to match with any counter party's request.

Case 8: Invalid parameters in swap order creation

Incorrect numbers are supplied

Handle failure

1) disable input of negative number 2) add bounds to input according to token decimal places and available balance

Case 9: Insufficient funds

Users cannot swap currencies if they have insufficient funds.

Handle failure

Given a user wants to swap 100 tokens of currency A for currency B and the user has only 80 tokens of currency A. When the user tries to initiate the swap then the swap should be rejected and the user should be notified about insufficient funds and unsuccessful swap operation

Case 10 : Time lock expired

Users cannot complete the swap if the time lock has expired

Handle failure

Given a user has initiated a swap and the time lock for the swap has expired. When the user tries to commit the swap, then the swap should be canceled instead and both users should be notified about the expired time lock and swap cancellation

Case 11: Network Issues

Users cannot complete the swap due to network issues

Handle failure

Given a user has initiated a swap and there are network issues resulting in at least one party being disconnected. When the user tries to commit the swap, then the swap should be canceled if no response is received and the user should be notified about the network issues
and both users should be notified about the swap cancellation

Case 12: Griefing/ Free Option Issue

To facilitate atomic exchanges, both parties secure their assets and permit their counterpart to withdraw them upon receipt of a secret. This situation can give rise to a challenge known as a "griefing attack" or the creation of an "American Call option." In this scenario, one party ceases their involvement in the exchange, causing their counterpart to endure a waiting period until a timelock expires before they can access their locked funds.

Handle failure

There are two ways to mitigate the unfairness of an inadvertent call option inherent in atomic swaps:

1. Eliminate the unintended option, or
2. Internalize the cost of an option so that the option is “priced in” with the cost of the swap.

Eliminating the inadvertent option on Layer 1 is not possible, as separate blockchains do not communicate with each other. Instead, PortalX internalizes the price of the option in the form of a “reclaimable deposit/bond”.

In our construction, the cost of the option is included in Alice’s Bitcoin HTLC. This internalizes the cost and eliminates inefficiency inherent to Tier Nolan atomic swaps (refer to ZK-Swaps White Paper - Part 1 v0.02 for details and analysis).

Cross-chain AMM Swaps (V2)

As stated in the previous sections, Portal Swap Protocol is designed to be independent of how parties are matched and who the parties are. The swap protocol works seamlessly regardless of Peer-to-Peer matching or an AMM is used. Essentially, AMM acts as a counterparty or a peer but the swap contracts and mechanism remain the same across the blockchains. We intend to publish the module of the whitepaper describing how Portal’s cross-chain AMM works in the coming weeks.

Complex swap contracts

Swap contracts can be easily extended to support partial fills, derivatives and a variety of complex contracts. We will describe these complex transactions in more detail in a separate module, to be published after the AMM module.

References

---

1. The approach here is loosely based on behavior-driven development. See BDD in Action by John Ferguson Smart and Jan Molak (2nd edition, 2023) and Formulation by Seb Rose and Gaspar Nagy (2021)
2. Mastering the Lightning Network (MLN) by Andreas Antonopoulus, Olaoluwa Osuntokun, and Rene Pickardt (2022)
3. See MLN, p 264, and . Stuck payments are payments that are neither fulfilled nor cancelled by an error. This is reportedly rare but possible due to the nature of onion routing when HTLCs are used. Point Time-Locked Contracts (PTLCs) should remedy this, as facilitated with the advent of Taproot and Schnorr. See also Payment Points – Part 2: "Stuckless" Payments.