zkFold x Asterizm

Interoperability Solution

Project ID - 1300199

Version 2.0

Change Log

Date	Version	Details
01.04.2025	1.0	Milestone 1, All new
21.04.2025	2.0	Plutus script for relayer and the client servers (New) Milestone 2

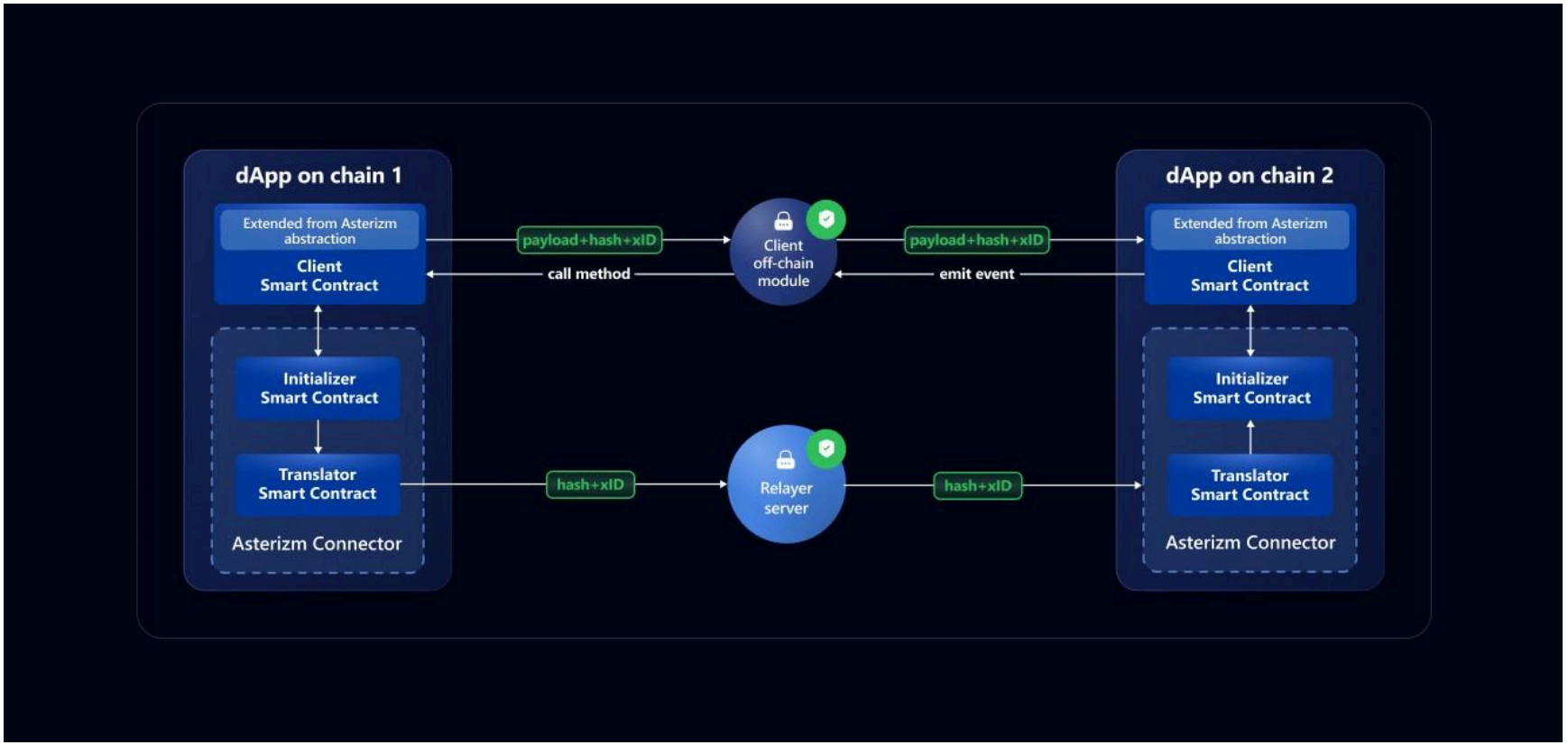
Functionality

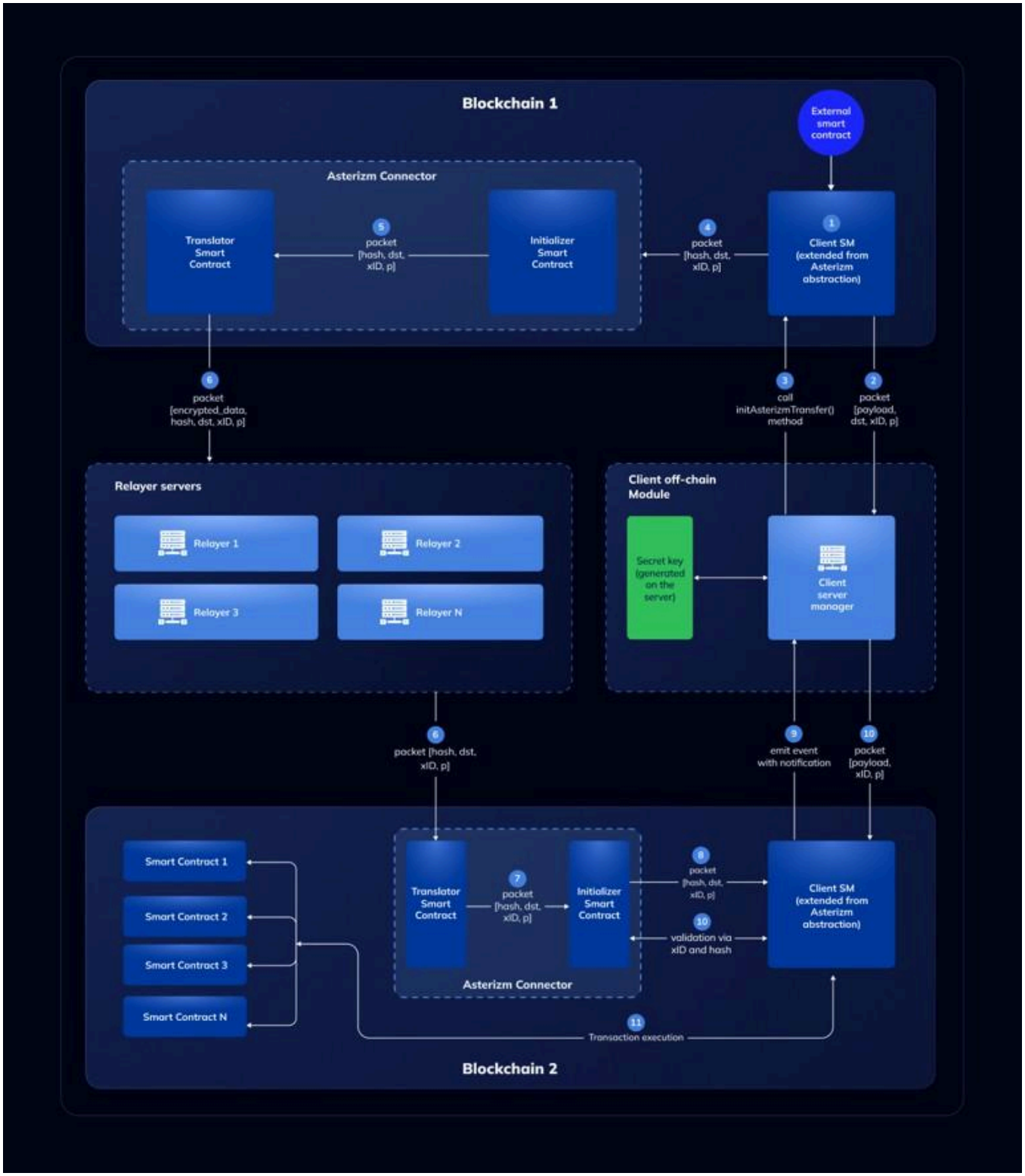
Basic Transfer Logic

This section provides a step-by-step description of the basic transfer logic, starting from initialization in the source network and ending with the execution of transfer instructions in the destination network.

If a relay or the client server is attacked, compromising one part of the system (either the relay or the client off-chain module) will not yield results, since the other part of the system will remain intact. Even if the relay is hacked, as mentioned above, the client off-chain module still remains. The relay alone cannot do anything. It doesn't even see the transfer instructions, since only the hash of these instructions is transmitted through it. For the client, a compromised relay does not pose a significant threat. In the worst case, they can use some external relays.

If we are talking about repeated executions of the same transfer, we have protection at the contract level — it is impossible to execute the instructions of the same transfer twice, as there are checks before execution. The point is that the transfer instructions will not be executed without the transfer going through the relay. In turn, the transfer will only reach the relay if it is sent by the client off-chain module, which, again, will not be able to perform this operation unless the transfer was previously initiated on the client contract. The transfer hash logic does not allow the transfer to be executed twice or tampered with in any way.


dApp on chain 1
Extended from Asterizm abstraction
Client Smart Contract
Initializer Smart Contract
Translator Smart Contract
Asterizm Connector
payload+hash+xID
call method
Client off-chain module
payload+hash+xID
emit event
hash+xID
Relayer server
hash+xID
dApp on chain 2
Extended from Asterizm abstraction
Client Smart Contract
Initializer Smart Contract
Translator Smart Contract
Asterizm Connector


Blockchain 1
External smart contract
Asterizm Connector
Translator Smart Contract
5 packet [hash, dst, xID, p]
Initializer Smart Contract
4 packet [hash, dst, xID, p]
1 Client SM (extended from Asterizm abstraction)
6 packet [encrypted_data, hash, dst, xID, p]
3 call initAsterizmTransfer() method
2 packet [payload, dst, xID, p]
Relayer servers
Relayer 1
Relayer 2
Relayer 3
Relayer N
Client off-chain Module
Secret key (generated on the server)
Client server manager
6 packet [hash, dst, xID, p]
9 emit event with notification
10 packet [payload, xID, p]
Smart Contract 1
Smart Contract 2
Smart Contract 3
Smart Contract N
Translator Smart Contract
7 packet [hash, dst, xID, p]
Initializer Smart Contract
8 packet [hash, dst, xID, p]
10 validation via xID and hash
Client SM (extended from Asterizm abstraction)
Asterizm Connector
11 Transaction execution
Blockchain 2

Step-by-step transfer logic

1. **Initialization** — Initialization occurs on the client contract in the source network. It does not matter who the initializer is, as long as the `_initAsterizmTransferEvent` function is called.
After all checks and the generation of the transfer data hash, the transfer is added to the `outboundTransfers` pool with the `successReceive = true` parameter, and the `InitiateTransferEvent` is triggered.

2. **Client Off-chain Module** — The client's off-chain module listens to the `InitiateTransferEvent` on the client contract.
Once triggered, the transfer data is received, validated, processed, and stored. At this stage, the relay fee is calculated and sent along with the function call.

3. **Client Off-chain Module** — The client's off-chain module calls the `initAsterizmTransfer` function on the client contract, sending the minimum set of data required to send the confirmation through the protocol.

4. **Initializer Contract** — After all validations, the client contract sends the transfer data to the initializer by calling the `initTransfer` function. At this stage, before sending to the initializer, a check is performed to determine in which currency the relay accepts fees. If it is a token, the specified amount of tokens is approved in favor of the initializer; if it is native currency, the native currency is sent along with the function call to the initializer.
Along with the transfer data, the external relay address is sent if the client has set it.
After successfully sending the transfer data to the initializer contract, the `successExecute = true` parameter is updated in the `outboundTransfers` pool.

5. **Initializer Contract** — During the `initTransfer` call, the client contract addresses are checked against the blocklist. If the address of the contract in the source network or the address of the client contract in the destination network is on the blocklist, the transfer is rejected.
After this check, the provided external relay address is validated. If it is not `address(0)`, the

transfer is sent to the external relay contract (at the same time, the `logExternalMessage` function is called on the internal relay of the protocol to notify the base off-chain relay module that the transfer was sent through an external relay and to receive the fee for such a transfer) by calling the `sendMessage` function on the external relay contract. If the provided external relay address is `address(0)`, the transfer is sent directly to the base relay of the protocol by calling the `sendMessage` function on the relay contract.

6. **Relay Contract** — Upon receiving the transfer data through the `sendMessage` function, the relay contract sends the received transfer fee to the contract owner, collects the payload for the off-chain module (which has the following structure: `srcChainId`, `srcAddress`, `dstChainId`, `dstAddress`, `txId`, `transferResultNotifyFlag`, `transferHash`), and triggers the `SendMessageEvent`, passing the collected payload and the fee value received by the contract. If `srcChainId == dstChainId`, the off-chain relay module is bypassed, and the `SuccessTransferEvent` is triggered instead.
The transfer data is then processed according to the logic described in step 8 (skipping step 7), with the difference that this occurs in the source network, not the destination network.

7. **Relay Listens to `SendMessageEvent`** — The relay listens to the `SendMessageEvent` on the relay contract. Once triggered, the off-chain relay module receives the data, validates it, processes it, and stores it. After this, the `transferMessage` function is called on the relay contract in the destination network. If the client has set the notification flag, after calling the function on the relay contract in the destination network, the `transferSendingResultNotification` function is called on the relay contract in the source network, passing the transfer data and its status to notify the client's off-chain module in the source network.

8. **`transferMessage` Function in Destination Network** — When the `transferMessage` function is called on the relay contract in the destination network, the transfer data is unpacked and sent to the initializer contract in the destination network by calling the `receivePayload` function.

9. **`receivePayload` Function in Destination Network** — When the `receivePayload` function is called on the initializer contract in the destination network, the client contract addresses (in the source and destination networks) are validated.
If the validation fails, the contract throws an error. Otherwise, a DTO is assembled to send the transfer to the client contract in the destination network by calling the `asterizmIzReceive` function.
If the client contract returns an error, the `PayloadErrorEvent` is triggered on the initializer contract, passing the transfer data and the error reason. If the call to the client contract is successful, the transfer is added to the `outgoingTransfers` pool, and the `SentPayloadEvent` is triggered.

10. **`asterizmIzReceive` Function in Destination Network** — After the transfer data is sent to the client contract in the destination network by calling the `asterizmIzReceive` function, multiple validations of the transfer data are performed. If all checks pass, the received transfer is added to the `inboundTransfers` pool with the `successReceive = true` parameter, and the `PayloadReceivedEvent` is triggered, which is listened to by the client's off-chain module in the destination network.

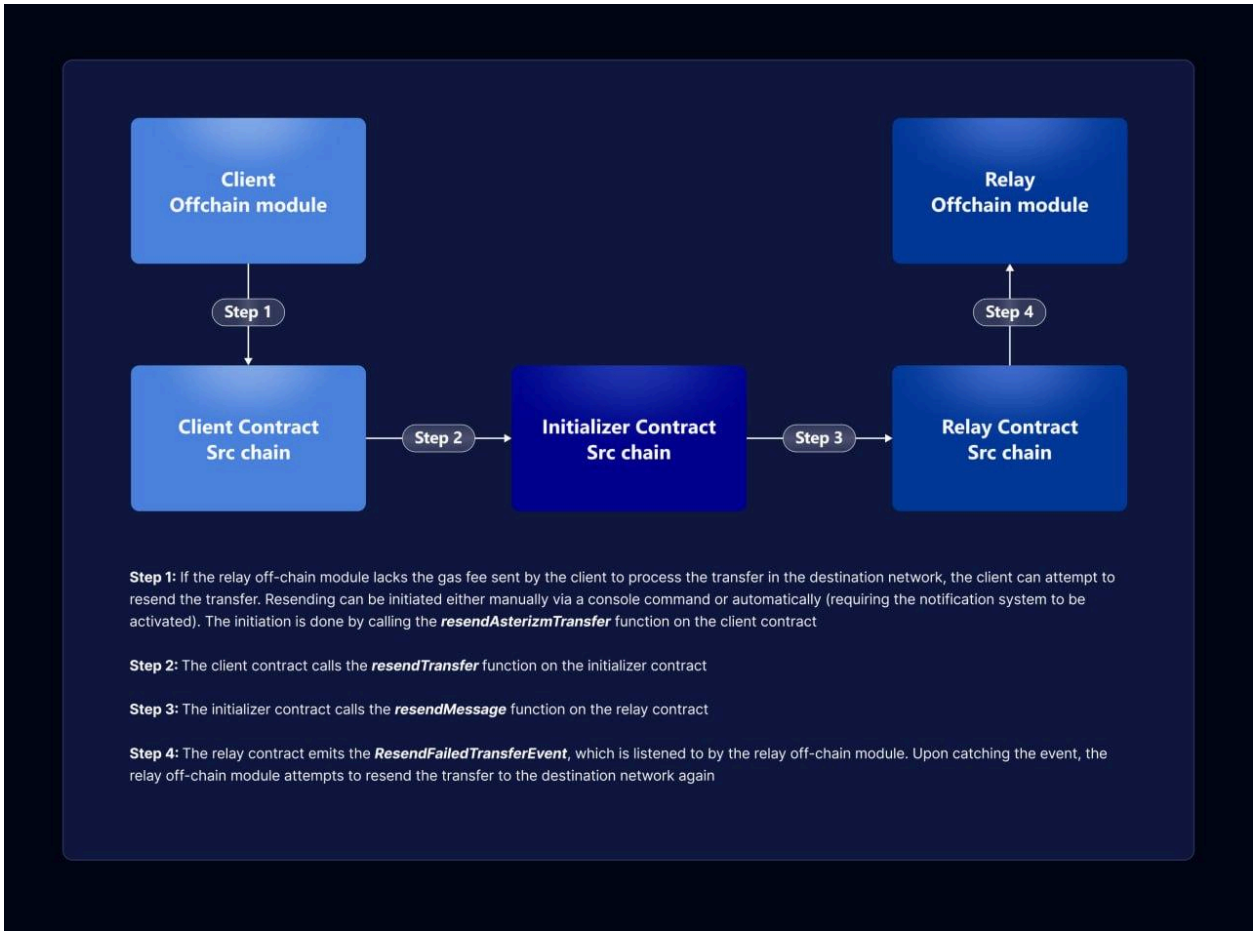
11. **Client Off-chain Module in Destination Network** — The client's off-chain module in the destination network listens to the `PayloadReceivedEvent` on the client contract.
Once triggered, it receives the data, validates it, processes it, and stores it.
After this, the `asterizmCiReceive` function is called on the client contract in the destination network, passing the necessary transfer data.

12. **`asterizmCiReceive` Function in Destination Network** — When the `asterizmCiReceive` function is called, multiple validations of the transfer data are performed, including a check for the completeness and integrity of the transfer data.
After successful validation, the transfer logic is executed by calling the internal `_asterizmReceive` function, which is implemented by the client (as mentioned earlier, the

protocol does not care about the specific logic implemented by the client).
This is the final step of the basic transfer logic.

Transfer Resend Logic

This functionality allows resending transfers that are stuck on the relay (either the base relay or an external relay).



There can be several reasons for a transfer to get stuck, the most common being insufficient coverage of the relay's costs for sending the transaction to the destination network (this can happen due to gas price spikes or exchange rate fluctuations of the native currency in either the source or destination network).

For such cases, the transfer resend functionality was developed, which allows "pushing" the transfer or adding additional fees to the already sent fees to mitigate the impact of native currency exchange rate fluctuations.

Anyone can resend transfers; it is not necessary to be the owner of the client contract that initiated the transfer.

There are two ways to use this logic:

1. Manual Resend — By calling functions on the contracts or through console commands of the client's off-chain module.

2. Automatic Resend — Implemented in the client's off-chain module. However, for this functionality to work, the notification functionality must be enabled; without it, automatic resend will not work.

Step-by-step Resend Logic

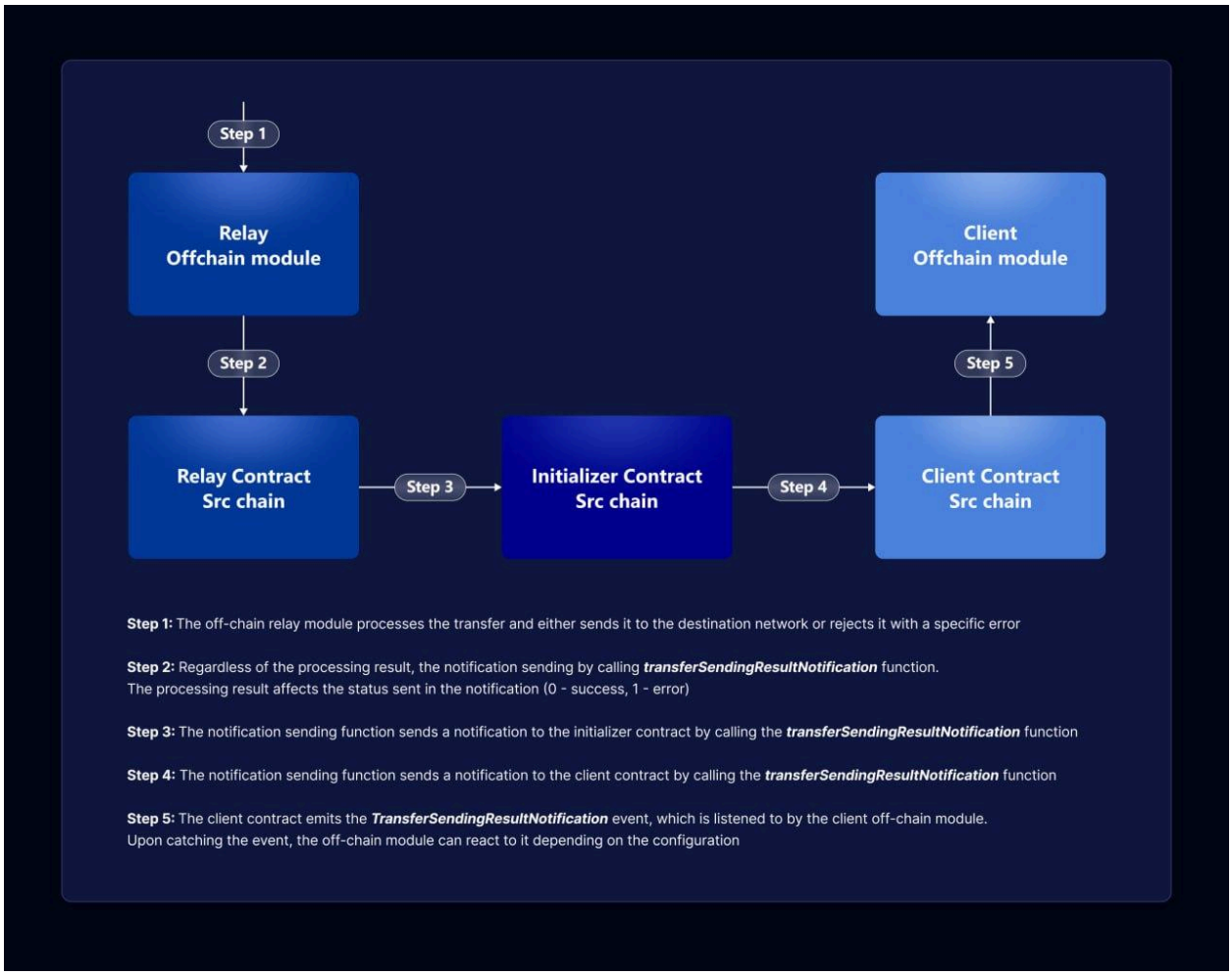
1. **Client Contract** — The `resendAsterizmTransfer` function must be called on the client contract, passing the hash of the transfer to be resent. This method accepts native currency, which will be sent to the relay to cover the insufficient fees.
At the same time, the resend data is sent to the initializer by calling the `resendTransfer` function, and the `ResendAsterizmTransferEvent` is triggered.
This step can only be performed by the sender of the client contract. If the initiator is a third-party address, the process must start from step 2.

2. **Initializer Contract** — When the resend data is sent to the initializer by calling the `resendTransfer` function, the relay through which the client's transfer was sent is determined. After this, the resend data is sent to the relay by calling the `resendMessage` function on the relay contract.
At the same time, the native currency sent by the resend initiator is also sent to cover the relay's insufficient fees.

3. **Relay Contract** — When the resend data is sent to the relay by calling the `resendMessage` function, the fee is sent to the relay owner, and the `ResendFailedTransferEvent` is triggered. This event is listened to by the off-chain relay module, which attempts to resend the specified transfer.

Notification Logic

This functionality allows the client's off-chain module to receive information about what is happening with the transfer on the off-chain relay module through which the transfer is sent.



The client decides whether they need this functionality or not. It is configured by setting the `notifyTransferSendingResult` flag in the client abstraction constructor (upon deployment).

The automatic transfer resend system is based on this logic, so if the client needs this functionality, the notification functionality must be enabled.

More details on this functionality can be found in the documentation.

Step-by-step Notification Logic

(when the client has enabled notifications, i.e., `notifyTransferSendingResult = true`):

1. Relay Off-chain Module — After the off-chain relay module sends (or fails to send due to insufficient fees) the transfer to the destination network, a notification with the transfer status is sent to the client's off-chain module by calling the `transferSendingResultNotification` function on the relay contract in the source network.

2. Relay Contract — When the notification is sent by calling the `transferSendingResultNotification` function on the relay contract in the source network, the notification is sent to the initializer by calling the `transferSendingResultNotification` function on the initializer contract.

3. Initializer Contract — When the notification is sent to the initializer by calling the `transferSendingResultNotification` function, the notification is sent to the client contract by calling the `transferSendingResultNotification` function.

4. Client Contract — When the notification is sent to the client contract by calling the `transferSendingResultNotification` function, the `TransferSendingResultNotification` event is triggered.

5. Client Off-chain Module — The client's off-chain module in the source network listens to the `TransferSendingResultNotification` event.
Once triggered, it receives the transfer data, which was sent as part of the notification. If the `_statusCode` parameter has a value of 1, the client's off-chain module will attempt to automatically resend the transfer by adding the missing fee in native currency to the resend initialization.

Trusted Address Logic

This functionality was developed to allow client contracts to define trusted addresses of client contracts located in other networks.

This helps avoid executing phishing transfers sent from unauthorized contracts and simplifies transfer initialization by eliminating the need for the client to specify the destination address each time. For transfer initialization, it is sufficient to specify only the `dstChainId`, and the destination address will be obtained from the client's trusted address pool.

In order for the protocol to interact with addresses from other networks, it was decided to store addresses in the uint format. The address of any connected network is converted to the uint format and added to the list of trusted addresses.

Step-by-step Trusted Address Logic

1. Add Trusted Address — To add a trusted address, the client contract owner must call the `addTrustedAddress` function, passing the required parameters. The address is then added to the `trustedAddresses` pool.
Only one trusted address can be added per network!

2. Add Multiple Trusted Addresses — To add a pool of trusted addresses, the client contract owner must call the `addTrustedAddresses` function, passing the required parameters.
The addresses are then added to the `trustedAddresses` pool.

3. Remove Trusted Address — To remove a trusted address, the client contract owner must call the `removeTrustedAddress` function, passing the required parameters.
The address is then removed from the `trustedAddresses` pool.

Refund Logic

This functionality was developed for specific client contract implementations, such as OmniChain Token, and allows users to recover burned or staked coins in the source network if the transfer fails to execute in the destination network for any reason.

For more detailed information on how this logic works and how to integrate it on the client side, please refer to the documentation.

You can get more information about refund logic here

Sender Logic (for Clients and Relays)

This functionality allows owners of client contracts (referred to as senders) and owners of relay contracts (referred to as relayers) to add addresses to pools so that transactions are sent from multiple addresses, not just the owner's address.
This increases the stability and speed of transfer execution across the entire protocol.

Addresses in these pools are only allowed to perform specific actions on the contracts; they do not replace the contract owners and do not gain full control over the contracts.
However, it is important to clearly understand which addresses are added to the sender pool, as this can be critical.

Step-by-step Sender Logic:

1. Add Sender — To add a sender to the client contract, the client owner must call the `addSender` function on the client contract. To add a sender to the relay contract, the relay owner must call the `addRelayer` function on the relay contract.

2. Remove Sender — To remove a sender from the client contract, the client owner must call the `removeSender` function on the client contract. To remove a sender from the relay contract, the relay owner must call the `removeRelayer` function on the relay contract.

Plutus script for relayer and the client servers

The documentation created in the first milestone essentially contains the description of both the on-chain and off-chain business logic. Indeed, to implement the protocol on EVM, one may need 3 separate contracts. However, Cardano smart contracts are very different. On-chain scripts on Cardano are typically much shorter, as they are merely specifications that transactions must satisfy.

As the previous sections describe the workflow of the protocol in EVM terms, at zkFold, we had to adapt it to the Cardano architecture. We have optimised the validation logic that must be performed on-chain. As a result, we concluded that those 3 contracts can be implemented with just a single Plutus script that enables the posting of signed messages on Cardano. This script will be used by both the relayer and the client servers.

One key design decision that allowed us to unify the contracts is that we perform message filtering completely off-chain as opposed to a hybrid model described in the docs. We believe that this decision not only simplifies the on-chain logic but also makes the protocol more modular and robust. For example, different applications that use Asterizm messaging can be working with different (sets of) relayer servers.

Products

This section describes the logic and implementation of products built on the Asterizm Protocol.

Omnichain Token

A product built on the protocol that allows working with a single token across all networks supported by the protocol.

The idea is that to send a token from one network to another, the tokens are "burned" (or staked, depending on the implementation of the omnichain token) in the source network and "minted" (or sent, depending on the implementation) in the destination network. Thanks to Asterizm Protocol's validation, it is impossible to receive tokens in the destination network without sending them in the source network, thus maintaining the total token supply even if the tokens are in different networks.

There are three implementations of Omnichain Token:

1. Basic Implementation — This implementation involves burning tokens in the source network and minting them in the destination network.

2. Native Coin Staking Implementation — This implementation involves staking native coins on the contract in the source network and sending them from the contract's balance in the destination network.

3. Token Staking Implementation — This implementation involves staking a specific token on the contract in the source network and sending it from the contract's balance in the destination network.

These implementations share a common set of transfer data, allowing them to be combined. For example, an Omnichain Token can be created for three networks: Ethereum, BSC, and Base. To receive tokens in BSC and Base, users must stake ETH in the Ethereum network. In this case, the user stakes ETH in the Ethereum network on the client contract and receives tokens in BSC or Base through minting.
When sending tokens from BSC or Base back to Ethereum, the tokens in BSC or Base are burned, and ETH is sent from the contract's balance in the destination network (which was previously staked there).

All Omnichain Token contracts must implement a unified interface. The logic of these contracts is similar to the implementation of a standard client contract of the protocol, with the difference that during transfer initialization in the source network, tokens must be burned/staked, and during transfer execution in the destination network, tokens must be minted/sent.

IMPORTANT!
In the case of the second and third implementations (staking coins/tokens), it is important to prohibit direct withdrawal of staked assets from the contract. For this purpose, the `coinWithdrawalsDisable` and `tokenWithdrawalsDisable` parameters exist (examples of usage are provided).

Additionally, since real assets are involved, it is crucial to enable the refund logic on client contracts, as described earlier. This ensures that users can recover their burned/staked funds if the transfer fails to execute in the destination network for any reason.