Crosschain Interoperability

Adhara Labs

Version 1.0.4, 2023-09-20

Table of Contents

Chapter 1. Introduction

There are certain instances of software deployments which may involve multiple blockchains.
For example, two blockchains could contain different currencies that need to be exchanged, or one blockchain could deliver securities while another handles the payment.

It is beneficial in such complex deployments to be able to atomically execute actions across more than one blockchain.

Payment versus Payment (PvP) crosschain interoperability involves payments taking place across two Ethereum blockchains.

Delivery versus Payment (DvP) crosschain interoperability involves transferring securities in a ledger on a Corda blockchain while the corresponding payment for the securities takes place on an Ethereum blockchain.

For example, in DvP, a securities earmark will be placed between two accounts in a function on a Corda ledger.
The transaction data emitted from the transaction will then be sent to a DvP contract deployed on Ethereum.
The aim is for the DvP contract to verify that the securities were earmarked on Corda, without direct interaction, but rather in a decentralised and trustless manner.
The DvP contract essentially trusts the parties (among which is a Corda notary and custodian) that signed the Corda transaction.
Upon verification of the earmark, the DvP contract will execute a transfer of the agreed exchange amount for the securities on a token contract also deployed on Ethereum.
An event and proof of the transfer on Ethereum will be sent to the Corda chain, which will then also be atomically verified before completing the transfer of securities.
An exchange of securities for funds should ideally occur in such a manner that either both legs of the exchange happens, or both legs are rolled back.

This document describes the deserialisation and verification schemes required by each blockchain in order to create, transmit and verify a transaction proof.

The first two chapters introduce the interop service functionality and defines the Crosschain interop service API.

The Foreign System Integration chapter explains the information needed for two separate system to interoperate.

The Crosschain Protocol Stack chapter outlines the Crosschain Interoperability Specification defined by the Enterprise Ethereum Alliance (EEA) which was incorporated into the implementation of the system.

The System Components chapter provides class diagrams that illustrate the on-chain and off-chain system components and the PvP Component Communication and DvP Component Communication chapters describe the communication diagrams, showing the specific interactions between each component.

The PvP happy and unhappy path flows are defined based on a leader-follower design pattern, discussed in the PvP Leader-Follower Happy Path and PvP Leader-Follower Unhappy Paths chapters, respectively.

The DvP happy and unhappy path flows are discussed in the DvP Leader-Follower Happy Path and DvP Leader-Follower Unhappy Paths chapters, respectively.

The Ethereum Proof Creation and Ethereum Proof Verification chapters provide an overview of the steps required to create and verify an Ethereum receipt proof, respectively.
Specific details and examples the proof verification process are covered in the Block Header Verification and Merkle Patricia Proof Verification chapters.

The Corda Proof Verification chapter lists the steps required to verify a Corda signature-based proof, details of which are covered in the Corda Signature-Based Proof Verification chapter

Chapter 2. Application API

2.1. Overview

The interop service allows Alice and Bob to trade one currency, say USD, on a Ethereum network A, for another currency, say GBP, on another Ethereum network B, without the need for a trusted intermediary.
Once a PvP trade between Alice and Bob has been agreed upon, details cannot be changed and must be settled via a Settlement Instruction.

Alice and Bob are said to be participants on the network, meaning that they both hold accounts on network A and network B. The PvP trade agreed on by Alice and Bob has a unique TradeId common for both legs (lead leg on network A and follow leg on network B) which cannot be used in any other settlement instruction.

The networks have the ability to allow some assets to be earmarked and released depending on certain conditions being met.
Earmarks are created via a settlement obligation.

A cryptographic proof of the assets being earmarked must be provided for a specific TradeId.
Each network understands the public keys and signature scheme of the validator set, i.e. validators that could sign foreign IBFT2 or QBFT block headers, on the network.

When Alice offers to trade USD on network A with Bob at a given rate and Bob accepts the trade, then Alice will earmark the USD on the network using an ERC-2020 type contract that holds the funds for Bob.
Similarly, Bob earmarks the settlement of the GBP asset on network B using an ERC-2020 type contract that holds the funds for Alice.
Once the earmark transaction is validated on network A, a proof of earmarking the USD is provided to a smart contract on the network B, which validates the proof.
If the transaction is valid, the smart contract moves the GBP funds held for Alice from Bob to her account on network B. Alice or Bob then provides proof to network A that the settlement asset was released to Alice.
Network A validates the proof and if the transaction is valid, the USD asset is released to Bob and both legs have been successful.

In the case of a cancellation, Alice earmarks the asset, but Bob does not.
Alice, therefore, decides to trigger a cancellation to get the USD back.
Alice triggers the cancellation on network B. The ERC2020 contract creates a proof if no evidence of Bob earmarking is present.
Alice retrieves the proof and provides the proof to network A. If the proof is valid, the funds are released back to Alice and both legs are cancelled.

Crosschain interop service application API defines endpoints for SettlementObligations, SettlementInstructions, and RepoObligations.

2.1.1. Version information

Version : 0.0.1

2.1.2. URI scheme

Host : localhost:3030
BasePath : /
Schemes : HTTP, HTTPS

2.1.3. Tags

- RepoObligations : Repo Obligations Resource
- SettlementInstructions : Settlement Instructions Resource
- SettlementObligations : Settlement Obligations Resource

2.1.4. Consumes

- `application/json`

2.1.5. Produces

- `application/json`

2.2. Resources

The following chapter defines the Application API paths for each resource.

2.2.1. RepoObligations

Repo Obligations Resource

Submit a repo obligation.

```
POST /{systemId}/repoObligations
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**repoObligation** *optional*	RepoObligation

Responses

HTTP Code	Description	Schema
201	Created repo obligation.	SettlementObligationResponse
500	An error occurred.	No Content

2.2.2. SettlementInstructions

Settlement Instructions Resource

Add a settlement instruction.

```
POST /{systemId}/settlementInstructions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**settlementInstruction** *optional*	SettlementInstruction

Responses

HTTP Code	Description	Schema
201	Created settlement instruction.	SettlementProof
400	Bad request.	No Content
500	An error occurred.	No Content

Consumes

- `application/json`

Fetch a settlement instruction by operationId, or by tradeId, fromAccount and toAccount.

```
GET /{systemId}/settlementInstructions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**fromAccount** *optional*	string
Query	**operationId** *optional*	string
Query	**toAccount** *optional*	string
Query	**tradeId** *optional*	string

Responses

HTTP Code	Description	Schema
200	Settlement instruction response.	SettlementInstructionResponse
400	Bad request.	No Content
500	An error occurred.	No Content

Delete a settlement instruction by operationId, or by tradeId, fromAccount and toAccount.

```
DELETE /{systemId}/settlementInstructions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**fromAccount** *optional*	string

Type	Name	Schema
Query	**operationId** *optional*	string
Query	**toAccount** *optional*	string
Query	**tradeId** *optional*	string

Responses

HTTP Code	Description	Schema
200	Settlement instruction response.	SettlementInstructionResponse
400	Bad request.	No Content
500	An error occurred.	No Content

Update a settlement instruction by tradeId, fromAccount and toAccount.

```
PATCH /{systemId}/settlementInstructions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**fromAccount** *optional*	string
Query	**toAccount** *optional*	string
Query	**tradeId** *optional*	string
Body	**updateSettlementInstruction** *optional*	UpdateSettlementInstruction

Responses

HTTP Code	Description	Schema
200	Settlement instruction response.	SettlementInstructionResponse
400	Bad request.	No Content
500	An error occurred.	No Content

2.2.3. SettlementObligations

Settlement Obligations Resource

Submit a settlement obligation.

```
POST /{systemId}/settlementObligations
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**settlementObligation** *optional*	SettlementObligation

Responses

HTTP Code	Description	Schema
201	Created settlement instruction.	SettlementObligationResponse
500	An error occurred.	No Content

Consumes

- `application/json`

2.3. Definitions

The following sections describe the definitions defined for the Application API endpoints.

2.3.1. RepoObligation

The Settlement Obligations API consists of the POST endpoint defined below.

> This API will only be used for POC's. In the end state, the bank will get the trade and generate an obligation in their treasury system through their normal processes. The treasury, or back office system, will then generate a request, either MT202 or API, to place a hold on the funds.

Name	Description	Schema
closingLeg *optional*		closingLeg
notional *optional*	**Example** : `0.0`	number
openingLeg *optional*		openingLeg
tradeId *required*	**Example** : `"string"`	string

closingLeg

Name	Description	Schema
amount *optional*	**Example** : `0.0`	number
fromAccount *optional*	**Example** : `"string"`	string
timestamp *optional*	**Example** : `0.0`	number
toAccount *optional*	**Example** : `"string"`	string

openingLeg

Name	Description	Schema
amount *optional*	**Example** : `0.0`	number

Name	Description	Schema
fromAccount *optional*	**Example** : `"string"`	string
toAccount *optional*	**Example** : `"string"`	string

This API will only be used for POC's. In the end state, the bank will get the trade and generate an obligation in their treasury system through their normal processes. The treasury, or back office system, will then generate a request, either MT202 or API, to place a hold on the funds.

2.3.2. SettlementInstruction

A settlement instruction is submitted via the Settlement Instructions Interop API and then transitions into various states.
The states that a settlement instruction can be in are: `confirmed`, `waitingForHold`, `waitingForCrossBlockchainCallExecuted`, `processed`, `failed`, and `timedOut`.
The following diagram shows the state transitions for the lead and follow ledger.

If the instruction is cancelled the states that it transitions through are then: `confirmed`, `waitingForHold`, `waitingForForeignSystemCancellation`, `cancelled`, `failed`, and `timedOut`.
The following diagram shows the state transitions for the lead and follow ledger.

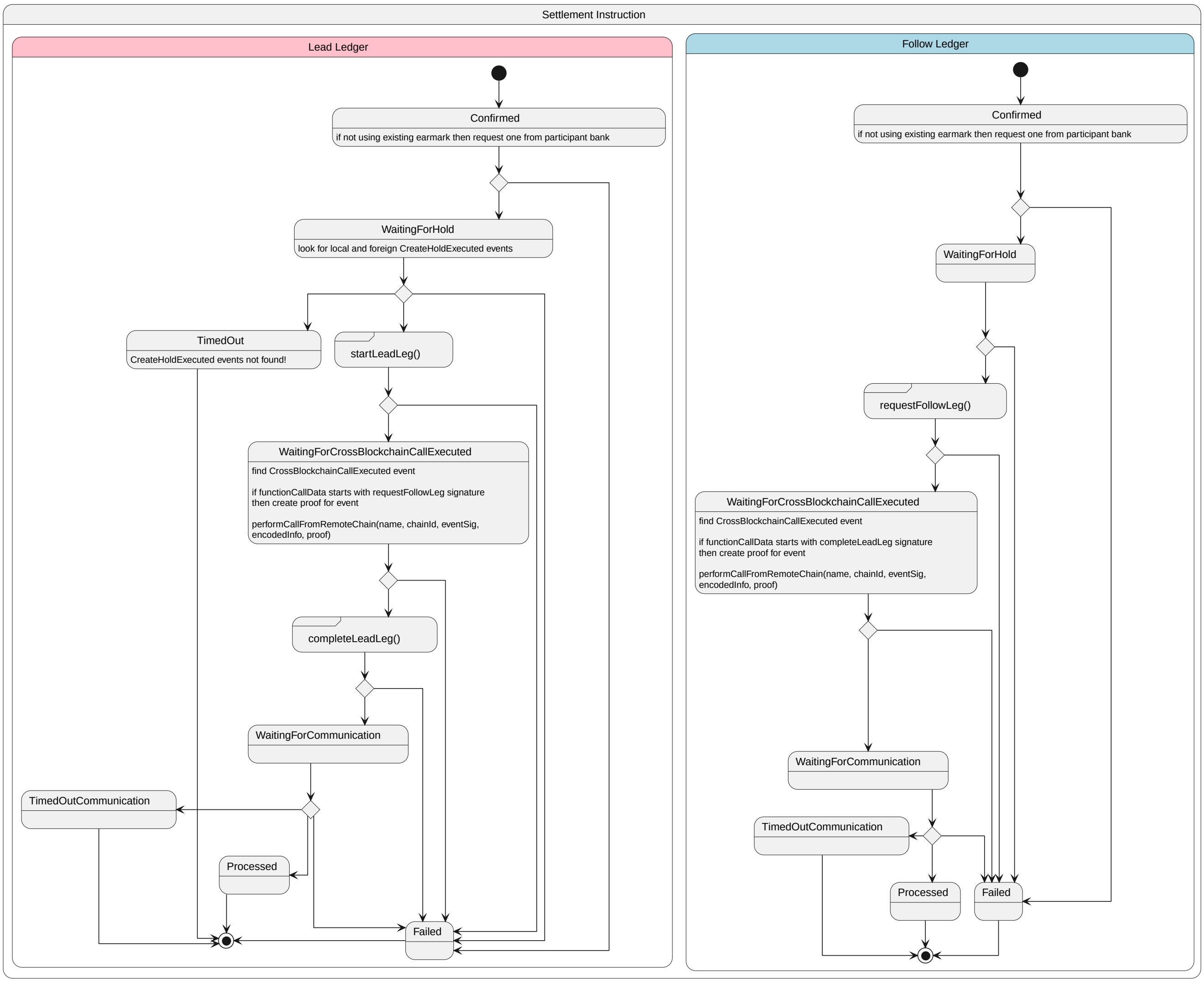

Figure 1. Settlement Instruction States

If the instruction is cancelled the states that it transitions through are then: `confirmed`, `waitingForHold`, `waitingForForeignSystemCancellation`, `cancelled`, `failed`, and `timedOut`.
The following diagram shows the state transitions for the lead and follow ledger.

> **Note:** A cancellation can be started from either the lead or follow ledger, based on where the hold is that needs to be cancelled.

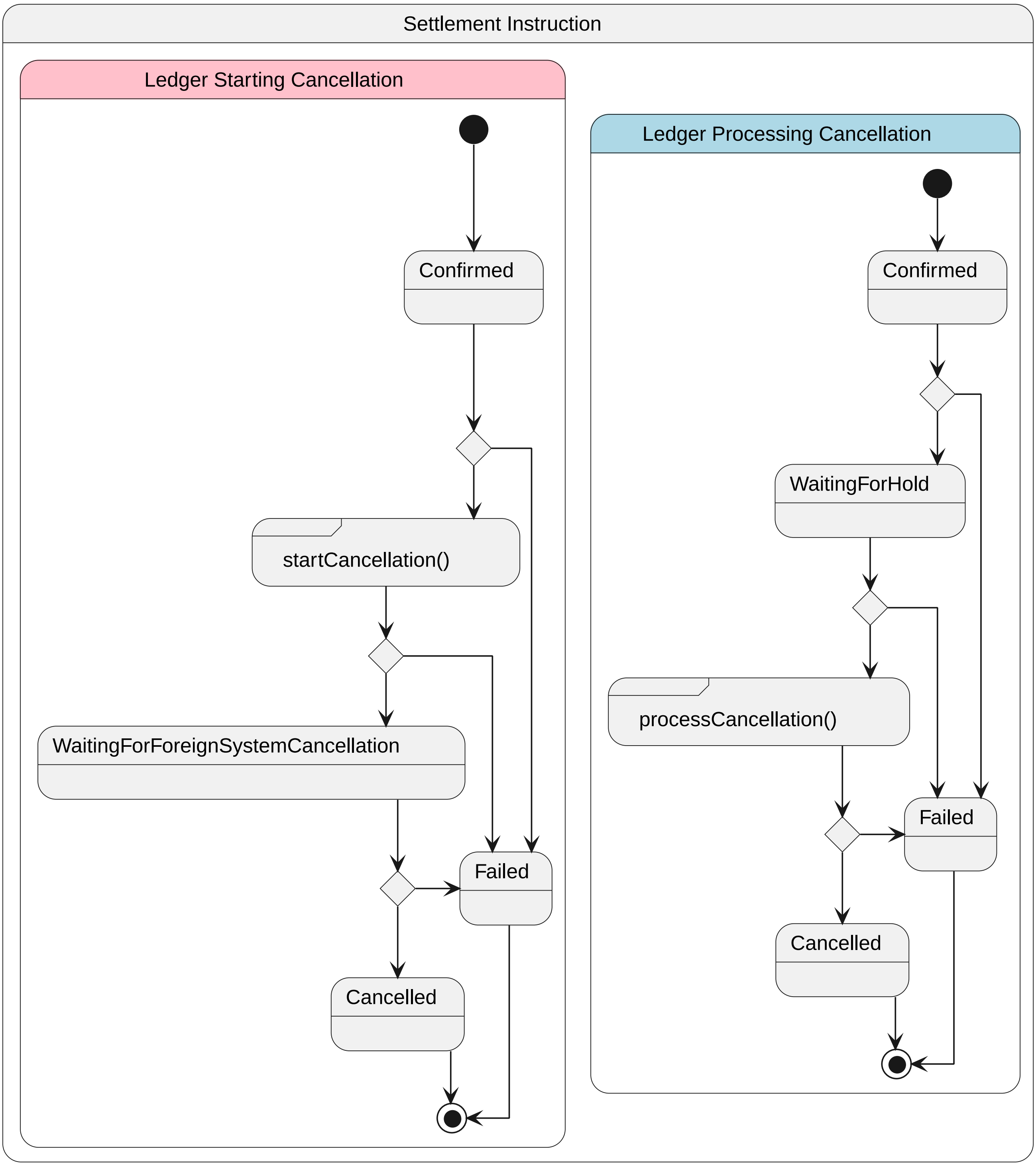

Figure 2. Settlement Instruction Cancellation States

Name	Description	Schema
amount *required*	**Example** : `0.0`	number
callbackURL *optional*	**Example** : `"string"`	string
currency *optional*	**Example** : `"string"`	string

Name	Description	Schema
foreignSystemId *optional*	**Example** : `1.0`	number
fromAccount *required*	**Example** : `"string"`	string
signatureOrProof *optional*		signatureOrProof
toAccount *required*	**Example** : `"string"`	string
tradeId *required*	**Example** : `"string"`	string
triggerLeadLeg *required*	**Example** : `true`	boolean
useExistingEarmark *required*	**Example** : `true`	boolean
useForCancellation *optional*	**Example** : `false`	boolean

signatureOrProof

Name	Description	Schema
encodedEventData *optional*	**Example** : `"string"`	string
encodedKey *optional*	**Example** : `"string"`	string
encodedSignature *optional*	**Example** : `"string"`	string

Name	Description	Schema
partialMerkleRoot *optional*	**Example** : "string"	string
platformVersion *optional*	**Example** : 0.0	number
schemaNumber *optional*	**Example** : 0.0	number
sourceSystemId *optional*	**Example** : 0.0	number

This API will only be used for POC's. In the end state, the bank will get the trade and generate an obligation in their treasury system through their normal processes. The treasury, or back office system, will then generate a request, either MT202 or API, to place a hold on the funds.

2.3.3. SettlementInstructionResponse

Name	Description	Schema
amount *optional*	**Example** : 0.0	number
callbackURL *optional*	**Example** : "string"	string
creationDate *optional*	**Example** : 0.0	number
currency *optional*	**Example** : "string"	string
foreignSystemId *optional*	**Example** : 1.0	number
fromAccount *optional*	**Example** : "string"	string

Name	Description	Schema
humanReadableTimestamp *optional*	**Example** : `"string"`	string
lastUpdate *optional*	**Example** : `0.0`	number
operationId *optional*	**Example** : `"string"`	string
signatureOrProof *optional*		signatureOrProof
state *optional*	**Example** : `"string"`	string
systemId *optional*	**Example** : `0.0`	number
toAccount *optional*	**Example** : `"string"`	string
tradeId *optional*	**Example** : `"string"`	string
triggerLeadLeg *optional*	**Example** : `true`	boolean
useExistingEarmark *optional*	**Example** : `true`	boolean
useForCancellation *optional*	**Example** : `false`	boolean

signatureOrProof

Name	Description	Schema
encodedEventData *optional*	**Example** : "string"	string
sourceSystemId *optional*	**Example** : 0.0	number

2.3.4. SettlementObligation

A list of the blockchain's current block validator addresses is stored as a mapping to each system chain Id.

The validators that could sign foreign IBFT2 or QBFT block headers are required in Verifying the Validator Signatures, which is used to verify the block headers.

Name	Description	Schema
amount *required*	**Example** : 0.0	number
currency *optional*	**Example** : "string"	string
fromAccount *required*	**Example** : "string"	string
toAccount *required*	**Example** : "string"	string
tradeId *required*	**Example** : "string"	string

This API will only be used for POC's. In the end state, the bank will get the trade and generate an obligation in their treasury system through their normal processes. The treasury, or back office system, will then generate a request, either MT202 or API, to place a hold on the funds.

2.3.5. SettlementObligationResponse

Name	Description	Schema
operationId *optional*	**Example** : "string"	string

2.3.6. SettlementProof

Name	Description	Schema
encodedInfo *optional*	**Example** : `"string"`	string
signatureOrProof *optional*	**Example** : `"string"`	string
sourceSystemId *optional*	**Example** : `0.0`	number
systemId *optional*	**Example** : `0.0`	number
tradeId *optional*	**Example** : `"string"`	string

2.3.7. UpdateSettlementInstruction

Name	Description	Schema
state *optional*	**Example** : `"string"`	string

Chapter 3. Administration API

3.1. Overview

Crosschain interop service Administration API defines endpoints for maintaining the Corda Notaries, Corda Participants, Corda Registered Functions, Interop Authentication Parameters, Interop Participants, and Validators resources.

3.1.1. Version information

Version : 0.0.1

3.1.2. URI scheme

Host : localhost:3031
BasePath : /
Schemes : HTTP, HTTPS

3.1.3. Tags

- CordaNotaries : Corda Notaries Resource
- CordaParticipants : Corda Participants Resource
- CordaRegisteredFunctions : Corda Registered Functions Resource
- InteropAuthParams : Interop Authentication Parameters Resource
- InteropParticipants : Interop Participants Resource
- ValidatorUpdateInstructions : Validator Update Instruction Resource
- Validators : Validators Resource

3.1.4. Consumes

- `application/json`

3.1.5. Produces

- `application/json`

3.2. Resources

The following chapter defines the Admin API paths for each resource.

3.2.1. CordaNotaries

Corda Notaries Resource

Create a corda notary.

```
POST /{systemId}/cordaNotaries
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**cordaNotary** *optional*	CordaNotary

Responses

HTTP Code	Description	Schema
200	Created corda notary.	CordaNotary
500	An error occurred.	No Content

Fetch if is a Corda Notary.

```
GET /{systemId}/cordaNotaries
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignSystemId** *required*	number
Query	**publicKey** *required*	string

Responses

HTTP Code	Description	Schema
200	Fetched if is a Corda Notary.	Response 200
500	An error occurred.	No Content

Response 200

Name	Description	Schema
isNotary *optional*	**Example** : `true`	boolean

Remove a corda notary.

```
DELETE /{systemId}/cordaNotaries
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignSystemId** *required*	number
Query	**publicKey** *required*	string

Responses

HTTP Code	Description	Schema
200	Removed corda notary.	CordaNotary
500	An error occurred.	No Content

3.2.2. CordaParticipants

Corda Participants Resource

Create a corda participant.

```
POST /{systemId}/cordaParticipants
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**cordaParticipant** *optional*	CordaParticipant

Responses

HTTP Code	Description	Schema
200	Created corda participant.	CordaParticipant
500	An error occurred.	No Content

Fetched if is a Corda participant.

```
GET /{systemId}/cordaParticipants
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignSystemId** *required*	number
Query	**publicKey** *required*	string

Responses

HTTP Code	Description	Schema
200	Fetched if is a Corda participant.	Response 200
500	An error occurred.	No Content

Response 200

Name	Description	Schema
isParticipant *optional*	**Example** : `true`	boolean

Remove a corda participant.

```
DELETE /{systemId}/cordaParticipants
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignSystemId** *required*	number
Query	**publicKey** *required*	string

Responses

HTTP Code	Description	Schema
200	Removed corda participant.	CordaParticipant
500	An error occurred.	No Content

3.2.3. CordaRegisteredFunctions

Corda Registered Functions Resource

Create a Corda registered function.

```
POST /{systemId}/cordaRegisteredFunctions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignSystemId** *required*	number
Query	**functionSignature** *required*	number
Body	**parameterHandlers** *optional*	CordaParameterHandlers

Responses

HTTP Code	Description	Schema
200	Created a Corda registered function.	CordaRegisteredFunction
500	An error occurred.	No Content

Consumes

- `application/json`

Fetch the parameter handler for a registered function.

```
GET /{systemId}/cordaRegisteredFunctions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string

Type	Name	Schema
Query	**foreignSystemId** *required*	number
Query	**functionSignature** *required*	number
Query	**index** *required*	number

Responses

HTTP Code	Description	Schema
200	Fetched the parameter handler for a registered function.	CordaParameterHandler
500	An error occurred.	No Content

Remove a Corda registered function.

```
DELETE /{systemId}/cordaRegisteredFunctions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignSystemId** *required*	number
Query	**functionSignature** *required*	number

Responses

HTTP Code	Description	Schema
200	Removed a Corda registered function.	CordaRegisteredFunction

HTTP Code	Description	Schema
500	An error occurred.	No Content

Consumes

- `application/json`

3.2.4. InteropAuthParams

Interop Authentication Parameters Resource

Create interop authentication parameters.

```
POST /{systemId}/interopAuthParams
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**interopAuthParam** *optional*	InteropAuthParam

Responses

HTTP Code	Description	Schema
200	Created interop authentication parameters.	InteropAuthParam
500	An error occurred.	No Content

Consumes

- `application/json`

Fetch if is interop authentication parameters.

```
GET /{systemId}/interopAuthParams
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignContractAddress** *required*	string
Query	**foreignSystemId** *required*	number

Responses

HTTP Code	Description	Schema
200	Fetched if is interop authentication parameters.	Response 200
500	An error occurred.	No Content

Response 200

Name	Description	Schema
isAuthParam *optional*	**Example** : `true`	boolean

Remove interop authentication parameters.

```
DELETE /{systemId}/interopAuthParams
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**interopAuthParam** *optional*	InteropAuthParam

Responses

HTTP Code	Description	Schema
200	Removed interop authentication parameters.	InteropAuthParam
500	An error occurred.	No Content

Consumes

- `application/json`

3.2.5. InteropParticipants

Interop Participants Resource

Create interop participant.

```
POST /{systemId}/interopParticipants
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**interopParticipant** *optional*	InteropParticipant

Responses

HTTP Code	Description	Schema
200	Created interop participant.	InteropParticipant
500	An error occurred.	No Content

Consumes

- `application/json`

Fetch the interop participant.

```
GET /{systemId}/interopParticipants
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignAccountId** *required*	number

Responses

HTTP Code	Description	Schema
200	Fetched the interop participant.	InteropParticipant
500	An error occurred.	No Content

Remove interop participant.

```
DELETE /{systemId}/interopParticipants
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**foreignAccountId** *optional*	string
Body	**interopParticipant** *optional*	InteropParticipant

Responses

HTTP Code	Description	Schema
200	Removed interop participant.	InteropParticipant
500	An error occurred.	No Content

3.2.6. ValidatorUpdateInstructions

Validator Update Instruction Resource

Create validator update instruction.

```
POST /{systemId}/validatorUpdateInstructions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Body	**validatorUpdateInstruction** *optional*	ValidatorUpdateInstructionRequest

Responses

HTTP Code	Description	Schema
200	Created validator update instruction.	ValidatorUpdateInstructionResponse
500	An error occurred.	No Content

Consumes

- `application/json`

Fetch the validator update instruction.

```
GET /{systemId}/validatorUpdateInstructions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**operationId** *required*	number

Responses

HTTP Code	Description	Schema
200	Fetched the validator update instruction.	ValidatorUpdateInstructionResponse
500	An error occurred.	No Content

Remove validator update instruction.

```
DELETE /{systemId}/validatorUpdateInstructions
```

Parameters

Type	Name	Schema
Path	**systemId** *required*	string
Query	**operationId** *required*	number

Responses

HTTP Code	Description	Schema
200	Removed validator update instruction.	ValidatorUpdateInstructionResponse
500	An error occurred.	No Content

3.2.7. Validators

Validators Resource

Fetch list of validators.

```
GET /validators
```

Parameters

Type	Name	Schema
Query	**blockHash** *optional*	string

Responses

HTTP Code	Description	Schema
200	Fetched list of validators.	Validators
500	An error occurred.	No Content

3.3. Definitions

The following sections describe the definitions defined for the Admin API endpoints.

3.3.1. CordaNotary

Verifying the signatures for a Corda trade or transaction, requires a list of the active Corda notaries.

The notaries are stored in the CrosschainMessaging contract as a mapping of their public key to the chain Id.

Name	Description	Schema
foreignSystemId *required*	**Example** : `0.0`	number
publicKey *required*	**Example** : `"string"`	string

3.3.2. CordaParameterHandler

Name	Description	Schema
componentIndex *optional*	**Example** : `0.0`	number
describedPath *optional*	**Example** : `"string"`	string

Name	Description	Schema
describedSize *optional*	**Example** : `0.0`	number
describedType *optional*	**Example** : `"string"`	string
fingerprint *optional*	**Example** : `"string"`	string
parser *optional*	**Example** : `"string"`	string
solidityType *optional*	**Example** : `"string"`	string

3.3.3. CordaParameterHandlers

Type : < CordaParameterHandler > array

3.3.4. CordaParticipant

Verifying the signatures for a Corda trade or transaction, requires a list of the active Corda participants.

The mapping for the participants are stored in the CrosschainMessaging contract for each chain Id, by their public key.

Name	Description	Schema
foreignSystemId *required*	**Example** : `0.0`	number
publicKey *required*	**Example** : `"string"`	string

3.3.5. CordaRegisteredFunction

Registers the parameter handler at given index for a Corda registered function with given function signature.

Name	Description	Schema
functionsSignature *optional*	**Example** : `0.0`	number
parameterHandlers *optional*		< CordaParameterHandler > array
systemId *optional*	**Example** : `0.0`	number

3.3.6. InteropAuthParam

Crosschain function call components need to determine the source blockchain and contract address on the source blockchain that initiated the crosschain function call. Functions in contracts on destination blockchains use this information to determine if the caller is authorised to execute the functionality in the function.

To prevent possible attacks, these authentication parameters are provided as hidden parameters that exist outside the scope of a functions declared parameters. The parameters are appended to the call data of a function call by the function call component and extracted by the application.

Name	Description	Schema
foreignContractAddress *required*	**Example** : `"string"`	string
foreignSystemId *required*	**Example** : `"string"`	string

3.3.7. InteropParticipant

The Interop service API will translate the fromAccount and toAccount fields to account Ids used by the local system by using the mapping maintained by the Interop Participants.

Name	Description	Schema
foreignAccountId *required*	**Example** : `"string"`	string

Name	Description	Schema
localAccountId *required*	**Example** : "string"	string

3.3.8. ValidatorUpdateInstructionRequest

Name	Description	Schema
blockHeader *optional*	**Example** : "string"	string
callbackURL *optional*	**Example** : "string"	string
contractAddress *optional*	**Example** : "string"	string
foreignSystemId *required*	**Example** : 1.0	number
operationId *required*	**Example** : "string"	string

3.3.9. ValidatorUpdateInstructionResponse

Name	Description	Schema
creationDate *optional*	**Example** : 0.0	number
foreignSystemId *optional*	**Example** : 1.0	number
humanReadableTimestamp *optional*	**Example** : "string"	string
lastUpdate *optional*	**Example** : 0.0	number

Name	Description	Schema
operationId *optional*	**Example** : `"string"`	string
state *optional*	**Example** : `"string"`	string
systemId *optional*	**Example** : `0.0`	number

3.3.10. Validators

A list of the blockchain's current block validator addresses is stored as a mapping to each system chain Id.

The validators that could sign foreign IBFT2 or QBFT block headers are required in Verifying the Validator Signatures, which is used to verify the block headers.

Name	Description	Schema
chainId *optional*	**Example** : `0.0`	number
ethereumAddresses *optional*	**Example** : `[ "string" ]`	< string > array

Chapter 4. Foreign System Integration

Integrating with a foreign system involves configuring information from the local system needed to verify transaction details, block headers, and Merkle Patricia tree proofs for the transaction receipt.

4.1. Foreign Accounts

A mapping of foreign to local account Ids is maintained by the local system.
The Interop service API will translate the `fromAccount` and `toAccount` fields to account Ids used by the local system by using the mapping maintained by the Interop Participants.

> A recommendation is made that the foreign system maintains a similar mapping of foreign to local account Ids so that it can convert any Interop service responses containing account Ids.

The mapping of foreign to local account Id is stored by the `XvP` contract as follows:

Mapping Foregign to Local Account

```
mapping(string => string) public foreignToLocalAccountId;
```

4.2. Ethereum Validators

The validators that could sign foreign IBFT2 or QBFT block headers are required in Verifying the Validator Signatures, which is used to verify the block headers.

Therefore, a list of the blockchain's current block validator addresses is stored as a mapping to each system chain Id.

The validators are stored in the `CrosschainMessaging` contract as follows:

Mapping of Validators

```
mapping(uint256 => address[]) public chainHeadValidators;
```

4.3. Receipts Root

Proofs generated by the interop service for Ethereum IBFT2 or QBFT based chains rely on proving that an event is contained in a receipt, and that the receipt is a leaf in a tree. The root of that tree, called the `receiptsRoot`, is required to equal the root of the most recent block's receipts tree, and when multiple Merkle Patricia proofs are calculated in the same block, they would all contain the same transaction `receiptsRoot`. Therefore, storing the transaction `receiptsRoot` during the block header verification process optimises the Merkle Patricia Proof Verification process. However, it is not strictly necessary to store the `receiptsRoot`, since the block header containing the receipt root will be submitted with each proof.

The mapping of the current block's hash to `receiptsRoot` is stored in the `CrosschainMessaging` contract as follows:

Mapping of blockHeaders to receiptsRoots

```
mapping(bytes32 => bytes32) receiptsRoots;
```

4.4. Corda Notaries and Participants

Verifying the signatures for a Corda trade or transaction, requires a list of the active notaries and participants, respectively.

The mapping for the notaries and participants are stored in the `CrosschainMessaging` contract for each chain Id, by their public key, as follows:

Mapping of Active Notaries

```
mapping(uint256 => mapping(uint256 => bool)) activeNotaries;
```

Mapping of Active Participants

```
mapping(uint256 => mapping(uint256 => bool)) activeParticipants;
```

The API endpoints used to maintain these mappings are described in the Corda Participants and Corda Notaries sections of the Interop Service API chapter.

4.5. Proving Schemes

There are currently three types of proving schemes that can be handled by the system, namely, Corda Trades, Corda Transactions, and Ethereum block headers. Proving schemes are onboarded for each system Id and are used to decode and verify the events for each type of proof received. The following table lists the scheme Ids assigned to each proving scheme.

Table 1. Proving Scheme Ids

Proving Scheme	Id
Corda Trade	0
Corda Transaction	1
Ethereum BlockHeader	2

The Corda signature schemes are not onboarded, but are required to be included in the signature meta-data. The system extracts the scheme from the meta-data and then verifies them based on their type. The following table lists the Ids assigned to each signature scheme.

Table 2. Signature Scheme Ids

Signature Scheme	Id
SECP256K1	2
SECP256R1	3
ED25519	4

The Corda signature schemes are required to be included in the signature meta-data.

The Crosschain Messaging contract stores the mapping of system Id to proving scheme Id as follows:

Mapping of System Id to Proving Scheme

```
mapping(uint256 => ProvingScheme) provingSchemes;
```

The mapping of systemId to provingScheme is part of initial interop service setup and not exposed by API endpoint.

4.6. Event Decoding Schemes

The event decoding schemes are required for handling different event types for, Corda Trades, Corda Transactions, and Ethereum block headers are onboarded. The following table lists the Ids assigned to each event decoding scheme.

Table 3. Event Decoding Scheme

Event Decoding Scheme	Id
Corda Trade	0
Corda Transaction	1
Ethereum Event Log	2

A mapping of system Id to event decoding scheme Id is stored in the `CrosschainFunctionCall` contract as follows:

Mapping of system Id to Event Decoding Scheme

```
mapping(uint256 => EventDecodingScheme) eventDecodingSchemes;
```

The mapping of systemId to eventDecodingScheme is part of initial interop service setup and not exposed by API endpoint.

Chapter 5. Crosschain Protocol Stack

The Crosschain Interoperability Specification is a formal definition of the implementation requirements for Enterprise Ethereum clients to achieve scalable crosschain communications.

The definition of the protocol stack, as indicated in the Crosschain Interoperability Protocol Stack diagram, consists of the following three layers:

1. Crosschain Applications
2. Crosschain Function Calls
3. Crosschain Messaging

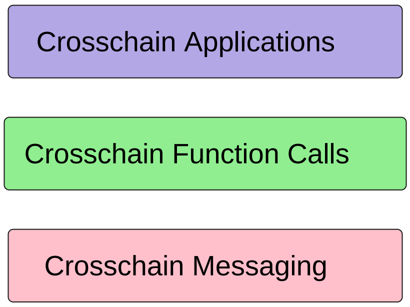

Figure 3. Crosschain Interoperability Protocol Stack

The Crosschain Applications layer is designed to contain all business logic, while the Crosschain Function Calls layer is required for executing functions across blockchains and the Crosschain Messaging layer enables a trustless environment concerning events that are generated by one blockchain to be trusted on another blockchain.

The design of the architecture, protocols, and interfaces is in alignment with the standards set by the Enterprise Ethereum Alliance (EEA), with the specifications defined in the three layers of the protocol stack.

Chapter 6. System Components

The table below provides a summary of the on-chain and off-chain system components grouped by the layers as defined in the EEA crosschain interoperability specification.

Table 4. Summary of XvP System Components

Specification Layer	**On-Chain**	**Off-Chain**
Application	XvP Contract	CrosschainApplication SDK
Function Call	CrosschainFunctionCall Contract	CrosschainFunctionCall SDK
Messaging	CrosschainMessaging Contract	CrosschainMessaging SDK

The following sections provide detailed class diagrams of these components.

6.1. On-Chain Components

The On-Chain Components diagram shows the on-chain components, which includes the XvP contract, the `CrosschainFunctionCall` contract and the `CrosschainMessaging` contract.

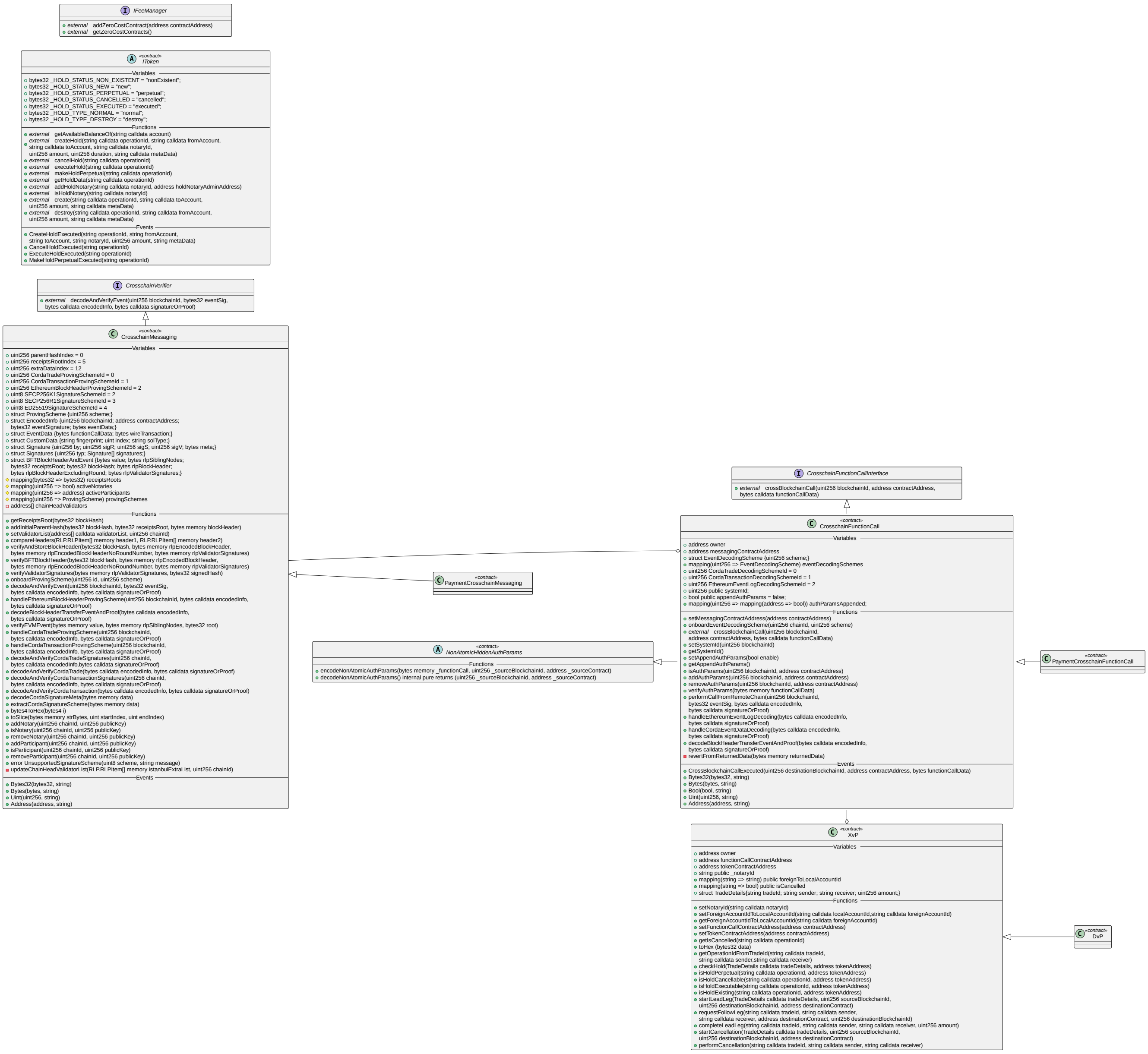

Figure 4. On-Chain Components

6.2. Off-Chain Components

The Off-Chain Components diagram shows the off-chain components, which includes the `CrosschainApplication` SDK, the `CrosschainFunctionCall` SDK and the `CrosschainMessaging` SDK.

Figure 5. Off-Chain Components

Chapter 7. PvP Leader-Follower Happy Path

The following section provides an overview of the happy path payment versus payment process. Details of the communication between the system components involved is given in the Component Communication chapter.

The Leader-Follower Payment versus Payment (PvP) diagram shows a transfer between two accounts in a PvP contract on a blockchain, called chain A. The event emitted by that transaction will then be used by a second PvP contract, which is deployed on another blockchain, chain B. The aim is for the PvP contract on chain B to verify the transfer of tokens between the two accounts on chain A, without direct interaction.
The processing is spread across three layers called the Application layer, Crosschain Function Call layer and Crosschain Messaging layer. Each layer is represented by off-chain and on-chain components.

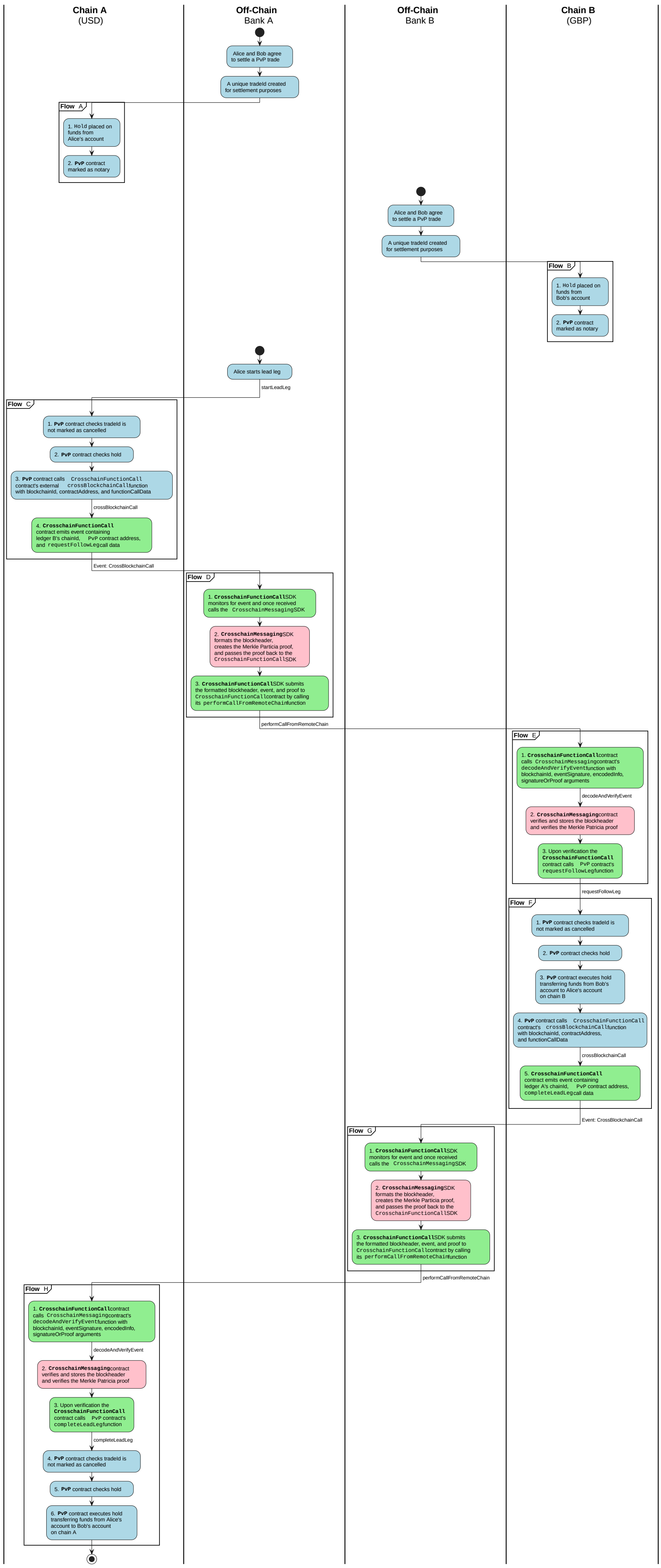

Figure 6. Leader-Follower Payment versus Payment (PvP)

Chapter 8. PvP Leader-Follower Unhappy Paths

The following sections describe the support provided when dealing with unhappy paths encountered during the settlement of a PvP trade.
For each unhappy scenario it is assumed that a unique trade id was agreed upon for settlement purposes.

8.1. States Automatically Recoverable

An unhappy path is automatically recoverable when the error produced is addressed by the SDK that encountered the error and the PvP trade can then be retried.

8.1.1. Feature 1: Transaction Failure Support

The transaction failure support for unhappy paths which are automatically recoverable are defined according to the following assumptions:

- There are no service interruptions when interacting with the two ledgers, for example, client connection loss.
- There are no cancellations by either party.

The following sections define the possible scenarios of transaction failure support within this unhappy paths feature.

Scenario 1: Insufficient Gas when Placing Hold

The unhappy path in the table describes the scenario when a transaction is submitted, to either chain A or chain B, with insufficient gas to place the hold.

Table 5. Scenario for Insufficient Gas when Placing Hold

Step	Description
Given	the hold is not placed on chain A
Or	the hold is not placed on chain B
And	insufficient gas provided on chain A
Or	insufficient gas provided on chain B
When	placing the hold
And	the hold transaction reverts
Then	the gas amount is increased
And	placing the hold is retried

Scenario 2: Insufficient Gas when Checking the Lead Leg Hold

The unhappy path in the table describes the scenario when the hold on chain A cannot be checked due to insufficient gas.

Table 6. Scenario for Insufficient Gas when Checking the Lead Hold

Step	Description
Given	a successfully placed hold on chain A
And	insufficient gas provided on chain A
When	the PvP contract checks the hold
And	the check transaction reverts
Then	the gas amount is increased
And	checking the hold is retried

Scenario 3: Lead Leg SDK restart

The unhappy path in the table describes the scenario when the lead leg SDK restarts and tries to reprocess the `CrossBlockchainCallExecuted` event from chain A.

Table 7. Scenario for Lead Leg SDK restart

Step	Description
Given	a successfully checked hold on chain A
And	a `CrossBlockchainCallExecuted` event is emitted from chain A
And	the `CrosschainFunctionCall` SDK is restarted
And	the `CrossBlockchainCallExecuted` event is processed twice by the `CrosschainFunctionCall` SDK
When	checking the hold on chain B fails
And	the transaction reverts
Then	the flow is aborted

Scenario 4: Insufficient Gas when Verifying the Lead Leg

The unhappy path in the table describes the scenario when the lead leg's payment details cannot be verified due to insufficient gas.

Table 8. Scenario for Insufficient Gas when Verifying the Lead Leg

Step	Description
Given	a successfully placed hold on chain B
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	insufficient gas provided to decode and verify the event on chain B
Or	insufficient gas provided to check the tradeId and hold

Step	Description
When	decoding and verifying the event
Or	checking the tradeId and hold
And	the transaction reverts
Then	the gas amount is increased
And	verifying the lead payment is retried

Scenario 5: Insufficient Gas when Executing the Follow Leg

The unhappy path in the table describes the scenario when the follow leg's hold cannot be executed due to insufficient gas.

Table 9. Scenario for Insufficient Gas when Executing the Follow Leg

Step	Description
Given	a successfully placed hold on chain B
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	the event has been decoded and verified
And	the tradeId and hold has been checked
And	there is insufficient gas
When	executing the hold
And	the transaction reverts
Then	the gas amount is increased
And	the instruction to `performCallFromRemoteChain` is resubmitted on chain B

Scenario 6: Follow Leg SDK restart

The unhappy path in the table describes the scenario when the follow leg SDK restarts and tries to reprocess the
`CrossBlockchainCallExecuted` event from chain B.

Table 10. Scenario for Follow Leg SDK restart

Step	Description
Given	a successfully executed hold chain B
And	a `CrossBlockchainCallExecuted` event is emitted from chain B
And	the `CrosschainFunctionCall` SDK is restarted
And	the `CrossBlockchainCallExecuted` event is processed twice by the `CrosschainFunctionCall` SDK
When	checking the hold on chain A fails

Step	Description
And	the transaction reverts
Then	the flow is aborted

Scenario 7: Insufficient Gas when Verifying the Follow Leg

The unhappy path in the table describes the scenario when the follow leg's payment details cannot be verified due to insufficient gas.

Table 11. Scenario for Insufficient Gas when Verifying the Follow Leg

Step	Description
Given	a successfully placed hold on chain A
And	an instruction to `performCallFromRemoteChain` submitted on chain A
And	insufficient gas provided to decode and verify the event on chain A
When	decoding and verifying the event
And	the transaction reverts
Then	the gas amount is increased
And	the instruction to `performCallFromRemoteChain` is resubmitted on chain A

Scenario 8: Insufficient Gas when Executing the Lead Leg

The unhappy path in the table describes the scenario when the lead leg's hold cannot be executed due to insufficient gas.

Table 12. Scenario for Insufficient Gas when Executing the Lead Leg

Step	Description
Given	a successfully placed hold on chain A
And	an instruction to `performCallFromRemoteChain` submitted on chain A
And	the event has been decoded and verified
And	insufficient gas provided to execute the hold on chain A
When	executing the hold
And	the transaction reverts
Then	the gas amount is increased
And	the instruction to `performCallFromRemoteChain` is resubmitted on chain A

8.1.2. Feature 2: Service Interruption Support

The transaction failure support for unhappy paths which are automatically recoverable are defined according to the following assumptions:

- The services will always be restored, when interruptions occur, while interacting with the two ledgers.
- There are no cancellations by either party.

The following section defines the possible scenario of service interruption support within this unhappy paths feature.

Scenario 1: Inability to Submit Transactions

The unhappy path in the table describes the scenario when transactions cannot be submitted to a ledger due to service interruptions.

Table 13. Scenario for Inability to Submit Transactions

Step	Description
Given	a settlement trade in-progress
And	the ledger services are unavailable
When	submitting a transaction on chain A
Or	submitting a transaction on chain B
And	the transaction reverts
Then	the system will wait
And	submit the transaction again

8.2. States Requiring Intervention

When an unhappy path produces an error that cannot be automatically addressed by the SDK, the error will either be addressed and the PvP trade retried, or alternatively, the trade will be cancelled and any changes to state are rolled back.

8.2.1. Feature 1: Transaction Failure Support

The transaction failure support unhappy paths requiring intervention are defined according to the following assumptions:

- There are no service interruptions when interacting with the two ledgers, for example, client connection loss.
- There are no cancellations by either party.

The following sections define the possible scenarios of transaction failure support within this unhappy paths feature.

Scenario 1: Insufficient Funds on Chain B

The unhappy path in the table describes the scenario when chain B has an insufficient account balance or insufficient tokens in order to place the hold.

Table 14. Scenario for Insufficient Funds on Chain B

Step	Description
Given	insufficient account balance on chain B
Or	insufficient tokens on chain B
When	placing a hold on chain B
Then	the hold transaction reverts
And	the flow is terminated

Scenario 2: Insufficient Funds on Chain A

The unhappy path in the table describes the scenario when chain A has an insufficient account balance or insufficient tokens in order to place the hold.

Table 15. Scenario for Insufficient Funds on Chain A

Step	Description
Given	insufficient account balance on chain A
Or	insufficient tokens on chain A
When	placing a hold
Then	the hold transaction reverts
And	the flow is terminated

Scenario 3: Incorrect Hold Amount on Chain A

The unhappy path in the table describes the scenario where the hold amount on chain A does not match the trade details.

Table 16. Scenario for Incorrect Hold Amount on Chain A

Step	Description
Given	a successfully placed hold on chain A
And	a successfully placed hold on chain B
And	the hold amount on chain A is incorrect
When	the hold amount is checked on chain A
Then	the transaction reverts
And	the flow is terminated

Scenario 4: Incorrect Notary Contract on Chain A

The unhappy path in the table describes the scenario where the notary on chain A does not match the trade details.

Table 17. Scenario for Incorrect Notary Contract on Chain A

Step	Description
Given	a successfully placed hold on chain A
And	a successfully placed hold on chain B
And	the notary contract is incorrect
When	the notary contract is checked on chain A
Then	the transaction reverts
And	the flow is terminated

Scenario 5: Failing to Decode the Event on Chain B

The unhappy path in the table describes the scenario where the event received by chain B cannot be decoded.

Table 18. Scenario for Failing to Decode the Event on Chain B

Step	Description
Given	a successfully placed hold on chain A
And	a successfully placed hold on chain B
And	a `CrossBlockchainCallExecuted` event is received by the `CrosschainFunctionCall` SDK
And	an instruction to `performCallFromRemoteChain` submitted on chain B
When	decoding of the event fails
And	the transaction reverts
Then	the flow is terminated

Scenario 6: Failing to Verify the Merkle Patricia Proof on Chain B

The unhappy path in the table describes the scenario where the proof received by chain B cannot be verified.

Table 19. Scenario for Failing to Verify the Merkle Patricia Proof on Chain B

Step	Description
Given	a successfully placed hold on chain A
And	a successfully placed hold on chain B
And	a `CrossBlockchainCallExecuted` event is received by the `CrosschainFunctionCall` SDK
And	an instruction to `performCallFromRemoteChain` submitted on chain B
When	verification of the Merkle Patricia proof fails
And	the transaction reverts
Then	the flow is terminated

Scenario 7: Failing TradeId or Hold Check on Chain B

The unhappy path in the table describes the scenario where the event received by chain B contains the incorrect trade id or amount.

Table 20. Scenario for Failing TradeId or Hold Check on Chain B

Step	Description
Given	a successfully placed hold on chain B
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	the event has been decoded and verified
And	the trade id is incorrect
Or	the hold amount is incorrect
When	checking the hold fails
Then	the transaction reverts
And	the flow is terminated

Scenario 8: Failing to Decode the Event on Chain A

The unhappy path in the table describes the scenario where the event received by chain A cannot be decoded.

Table 21. Scenario for Failing to Decode the Event on Chain A

Step	Description
Given	a successfully executed hold on chain B
And	a `CrossBlockchainCallExecuted` event is received by the `CrosschainFunctionCall` SDK
And	an instruction to `performCallFromRemoteChain` submitted on chain A
When	decoding of the event fails
And	the transaction reverts
Then	the flow is terminated

Scenario 9: Failing to Verify the Merkle Patricia Proof on Chain A

The unhappy path in the table describes the scenario where the proof received by chain A cannot be verified.

Table 22. Scenario for Failing to Verify the Merkle Patricia Proof on Chain A

Step	Description
Given	a successfully executed hold on chain B
And	a `CrossBlockchainCallExecuted` event is received by the `CrosschainFunctionCall` SDK

Step	Description
And	an instruction to `performCallFromRemoteChain` submitted on chain A
When	verification of the Merkle Patricia proof fails
And	the transaction reverts
Then	the flow is terminated

8.3. Cancellations

Cancelling a PvP trade can be used as a mechanism to resolve unhappy paths requiring intervention which will result on the error state being rolled back.
For example, if Alice and Bob have agreed to settle a trade, but after some time has passed, either Alice or Bob do not response then the other party can decide to cancel the trade.

The cancellation support unhappy paths are defined according to the following assumptions:

- There are no service interruptions on either chain A or chain B
- There are no gas-related transaction failures
- Trades can only be cancelled using the unique trade id
- Trades can only be cancelled on the chain where the hold does not already exist
- Trades can only be cancelled once by either party

Feature 1 describes the cancellation scenarios for cancellations started on the follow ledger and Feature 2 describes the scenarios started on the lead ledger.

8.3.1. Feature 1: Cancellation started on the follow ledger

The following diagram shows the Leader-Follower PvP Unhappy Paths for Cancellations Started on the Follow Leg.

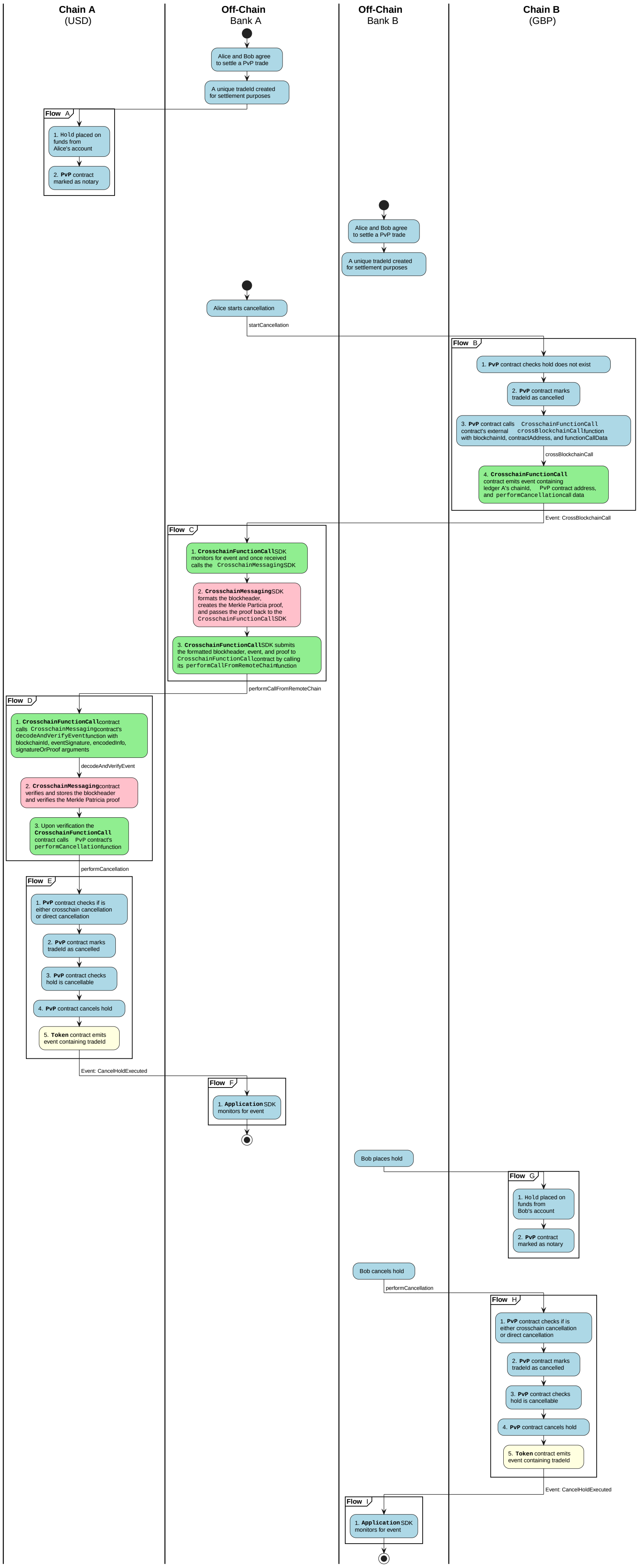

Figure 7. Leader-Follower PvP Unhappy Paths for Cancellations Started on the Follow Leg

Scenario 1: Alice starts a crosschain cancellation before the hold is placed on the follow ledger

The table describes the scenario shown in flow A to F in the diagram, which is the case where a trade cancellation is initiated on chain B before the hold is placed.

Table 23. Scenario when Alice starts a crosschain cancellation before the hold is placed on the follow ledger

Step	Description
Given	the hold placed on chain A
And	the hold not placed on chain B
When	Alice executes `pvp.startCancellation()` on chain B
And	the PvP contract marks the tradeId as cancelled on chain B
And	the PvP contract on chain B calls `crosschainFunctionCall.crossBlockchainCall()` with the `pvp.performCancellation()` function call data
And	the `CrosschainFunctionCall` SDK for bank A picks up the `CrossBlockchainCallExecuted` event
And	the `CrosschainMessaging` SDK formats the blockheader and creates the Merkle Patricia proof
Then	`crosschainFunctionCall.performCallFromRemoteChain()` is called on chain A
And	the Merkle Patricia proof is verified on chain A by the `crosschainMessaging.decodeAndVerifyEvent()` function
And	the `CrosschainFunctionCall` contract calls `pvp.performCancellation()` on chain A
And	the PvP contract cancels the hold on chain A
And	the `Application` SDK picks up `CancelHoldExecuted` event
And	the flow is terminated

Scenario 2: Bob performs a direct cancellation after the trade has been cancelled

The table describes the scenario shown in flow G to I in the diagram, which is the case where a hold is cancelled after it was placed on the follow ledger following a crosschain cancellation.

Table 24. Scenario when Bob performs a direct cancellation after the trade has been cancelled

Step	Description
Given	Alice cancelled the trade on chain B
And	Bob places the hold on chain B

Step	Description
When	Bob calls `pvp.performCancellation()` on chain B
And	the PvP checks the tradeId was marked as cancelled
Then	the PvP contract cancels the hold
And	the `Application` SDK picks up `CancelHoldExecuted` event
And	the flow is terminated

Scenario 3: Alice starts a crosschain cancellation after the hold is placed on the follow ledger

The table describes the scenario shown in flow B in the diagram, which is the case where a trade cancellation attempt is initiated on the follow ledger after the hold has already been placed on the follow ledger.

Table 25. Scenario when Alice starts a crosschain cancellation after the hold is placed on the follow ledger

Step	Description
Given	the hold placed on chain A
And	the hold placed on chain B
When	Alice executes `pvp.startCancellation()` on chain B
And	the PvP contract checks that the hold does not exist on chain B
Then	the PvP contract reverts
And	the flow is terminated

Scenario 4: Bob performs a direct cancellation before the trade has been cancelled

The table describes the scenario shown in flow H in the diagram, which is the case where a direct cancellation is attempted on the follow ledger while the trade still exists on the lead ledger.

Table 26. Scenario when Bob performs a direct cancellation before the trade has been cancelled

Step	Description
Given	Bob places the hold on chain B
And	the trade has not been cancelled
When	Bob calls `pvp.performCancellation()` on chain B
And	the PvP checks the function call came from the `crosschainFunctionCall` contract or the tradeId was marked as cancelled
Then	the PvP contract reverts
And	the flow is terminated

8.3.2. Feature 2: Cancellation started on the lead ledger

The following section describes the scenarios for cancellations started on the lead ledger.

The Leader-Follower PvP Unhappy Paths for Cancellations Started on the Lead Leg diagram shows the leader-follower PvP flows for cancelling a PvP trade on the lead ledger.

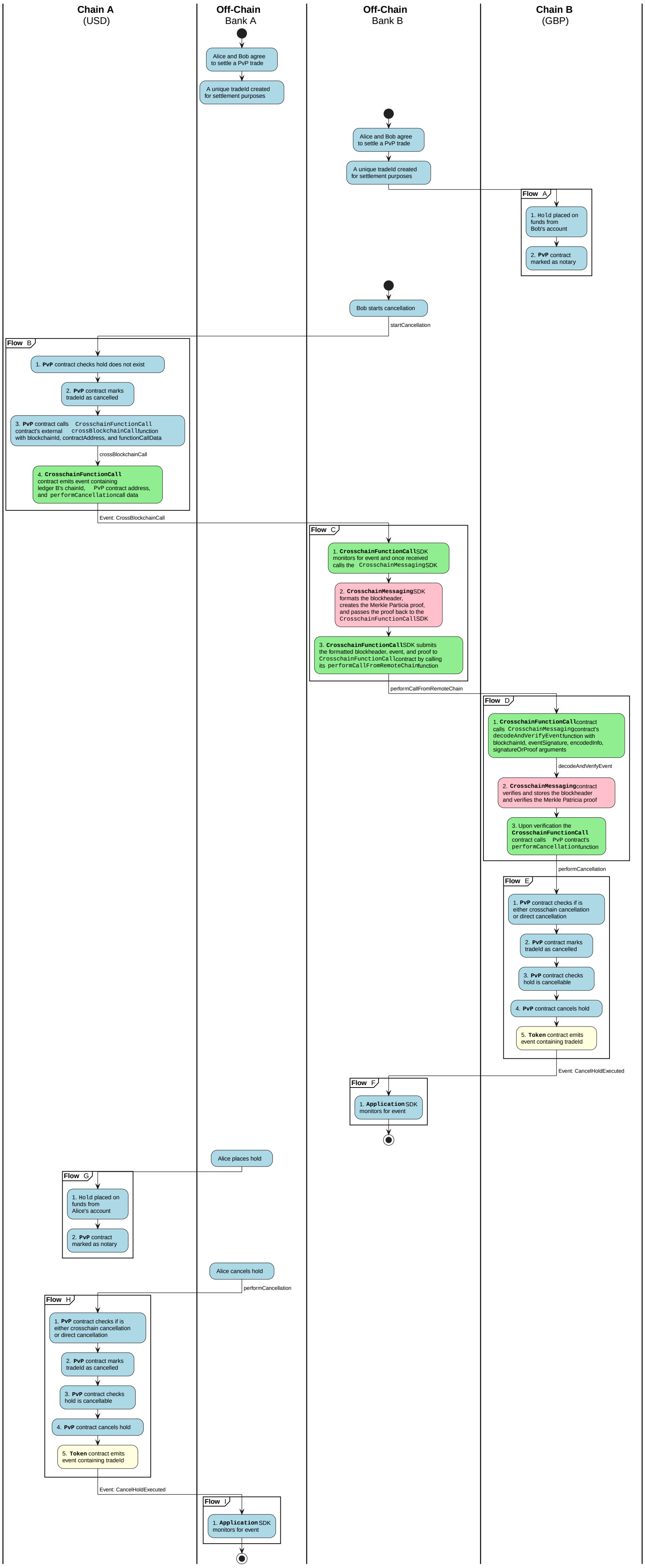

Figure 8. Leader-Follower PvP Unhappy Paths for Cancellations Started on the Lead Leg

Scenario 1: Bob starts a crosschain cancellation before the hold is placed on the lead ledger

The table describes the scenario shown in flow A to F in the diagram, which is the case where a trade cancellation is initiated on chain A before the hold is placed.

Table 27. Scenario when Bob starts a crosschain cancellation before the hold is placed on the lead ledger

Step	Description
Given	the hold placed on chain B
And	the hold not placed on chain A
When	Bob executes `pvp.startCancellation()` on chain A
And	the PvP contract marks the tradeId as cancelled on chain A
And	the PvP contract on chain A calls `crosschainFunctionCall.crossBlockchainCall()` with the `pvp.performCancellation()` function call data
And	the `CrosschainFunctionCall` SDK for bank B picks up the `CrossBlockchainCallExecuted` event
And	the `CrosschainMessaging` SDK formats the blockheader and creates the Merkle Patricia proof
Then	`crosschainFunctionCall.performCallFromRemoteChain()` is called on chain B
And	the Merkle Patricia proof is verified on chain B by the `crosschainMessaging.decodeAndVerifyEvent()` function
And	the `CrosschainFunctionCall` contract calls `pvp.performCancellation()` on chain B
And	the PvP contract cancels the hold on chain B
And	the `Application` SDK picks up `CancelHoldExecuted` event
And	the flow is terminated

Scenario 2: Alice performs a direct cancellation after the trade has been cancelled

The table describes the scenario shown in flow G to I in the diagram, which is the case where a hold is cancelled after it was placed on the lead ledger following a crosschain cancellation.

Table 28. Scenario when Alice performs a direct cancellation after the trade has been cancelled

Step	Description
Given	Bob cancelled the trade on chain A
And	Alice places the hold on chain A
When	Alice calls `pvp.performCancellation()` on chain A

Step	Description
And	the PvP checks the tradeId was marked as cancelled
Then	the PvP contract cancels the hold
And	the `Application` SDK picks up `CancelHoldExecuted` event
And	the flow is terminated

Scenario 3: Bob starts a crosschain cancellation after the hold is placed on the lead ledger

The table describes the scenario shown in flow A in the diagram, which is the case where a crosschain cancellation is attempted after the hold has already been placed on the lead ledger.

Table 29. Scenario when Bob starts a crosschain cancellation after the hold is placed on the lead ledger

Step	Description
Given	the hold placed on chain B
And	the hold placed on chain A
When	Bob executes `pvp.startCancellation()` on chain A
And	the PvP contract checks that the hold does not exist on chain A
Then	the PvP contract reverts
And	the flow is terminated

Scenario 4: Alice performs a direct cancellation before the trade has been cancelled

The table describes the scenario shown in flow H in the diagram, which is the case where a direct cancellation is attempted on the lead ledger while the trade still exists on the follow ledger.

Table 30. Scenario when Alice performs a direct cancellation before the trade has been cancelled

Step	Description
Given	Alice places the hold on chain A
And	the trade has not been cancelled
When	Alice calls `pvp.performCancellation()` on chain A
And	the PvP checks the function call came from the `crosschainFunctionCall` contract or the tradeId was marked as cancelled
Then	the PvP contract reverts
And	the flow is terminated

Chapter 9. PvP Component Communication

The following sections describe the happy and unhappy path communication diagrams, detailing the specific interactions between components in the leader-follower PvP process.

9.1. Happy Path

The communication between the components involved in the leader-follower PvP happy path can be identified as follows:

1. Chain A start lead leg payment
2. Off-chain lead leg orchestration
3. Chain B request follow leg payment upon verification
4. Chain B follow leg payment
5. Off-chain follow leg orchestration
6. Chain A complete lead leg payment upon verification

The lead leg payment and complete lead leg payment communications take place on chain A, while the request follow leg payment and follow leg payment communications take place on chain B. The off-chain orchestration communications involve the `CrosschainFunctionCall` and `CrosschainMessaging` SDKs, which prepares the event and proof before it is submitted to each blockchain.

9.1.1. Chain A Start Lead Leg

Once a trade has been agreed on and a unique tradeId is created for settlement purposes, the Corda ledger places the hold on Alice's securities.
The Start Lead Leg diagram shows the interactions between the Application Layer and Crosschain Function Call Layer on chain A. The off-chain `Application` SDK calls the `startLeadLeg` function on the `PvP0` contract deployed on chain A. It is followed by a call to the `crossBlockchainCall` function on the `CrosschainFunctionCall` contract, which is the protocol provided for applications to call functions on other blockchains.
The `crossBlockchainCall` will emit an event containing the id for chain B, the `PvP1` contract address on chain B, and the `requestFollowLeg` call data.
The event will then be monitored for and consumed by the `CrosschainFunctionCall` SDK off-chain.

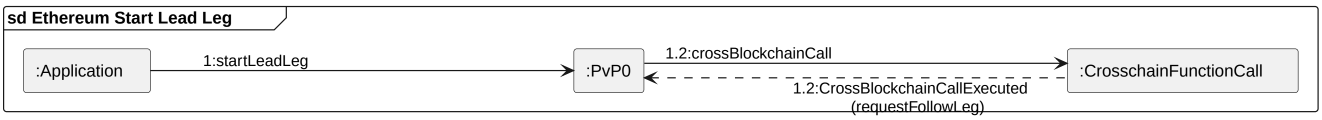

Figure 9. Start Lead Leg

9.1.2. Off-Chain Lead Leg Orchestration

The off-chain orchestration communications in the Lead Leg Orchestration diagram makes use of the functionality provided by the off-chain `CrosschainFunctionCall` and `CrosschainMessaging` SDK components.

The CrosschainFunctionCall SDK picks up the event emitted by the CrosschainFunctionCall contract and calls the CrosschainMessaging SDK.
The CrosschainMessaging SDK then creates the Merkle Patricia tree proof, which is returned to the CrosschainFunctionCall SDK.

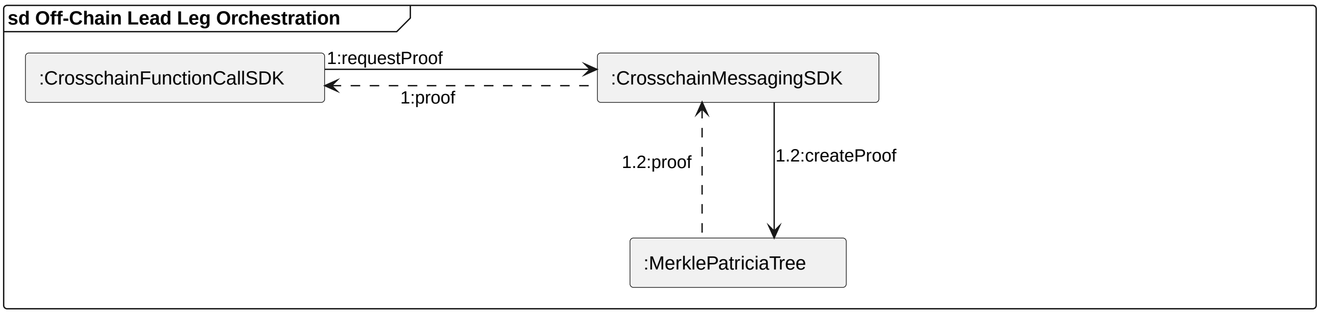

Figure 10. Lead Leg Orchestration

9.1.3. Chain B Request Follow Leg

The request follow leg process that occurs on chain B is shown in the Request Follow Leg diagram.
The CrosschainFunctionCall SDK reads the block header that contains the event and submits the block header, event, and proof using the performCallFromRemoteChain function.
The CrosschainFunctionCall contract's decodeBlockHeaderTransferEventAndProof function will verify the validator signatures on the block header and persist the receipts root contained in the block header.
The CrosschainMessaging contract's decodeAndVerifyEvent function verifies the Merkle Patricia tree proof by checking that its root is the same as the receipts root in the block header.
Upon verification of the event the requestFollowLeg function is called, which was obtained from the function call data emitted in the data from the lead payment process.

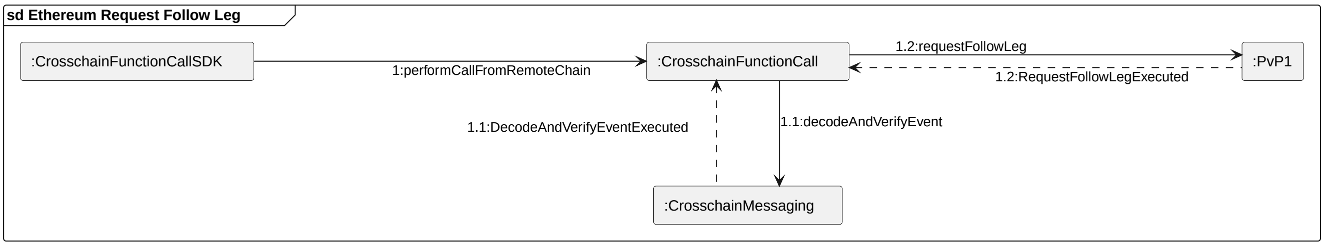

Figure 11. Request Follow Leg

9.1.4. Chain B Follow Leg

The Follow Leg Communication diagram illustrates the communication taking place during the follow leg payment process.
Once the tradeId and hold is confirmed, the PvP1 contract executes the hold and then calls the crossBlockchainCall function on the CrosschainFunctionCall contract.
The crossBlockchainCall will emit an event containing the id for chain B, the PvP0 contract address on chain A, and the completeLeadLeg call data.
The event will then be monitored for and consumed by the CrosschainFunctionCall SDK off-chain.

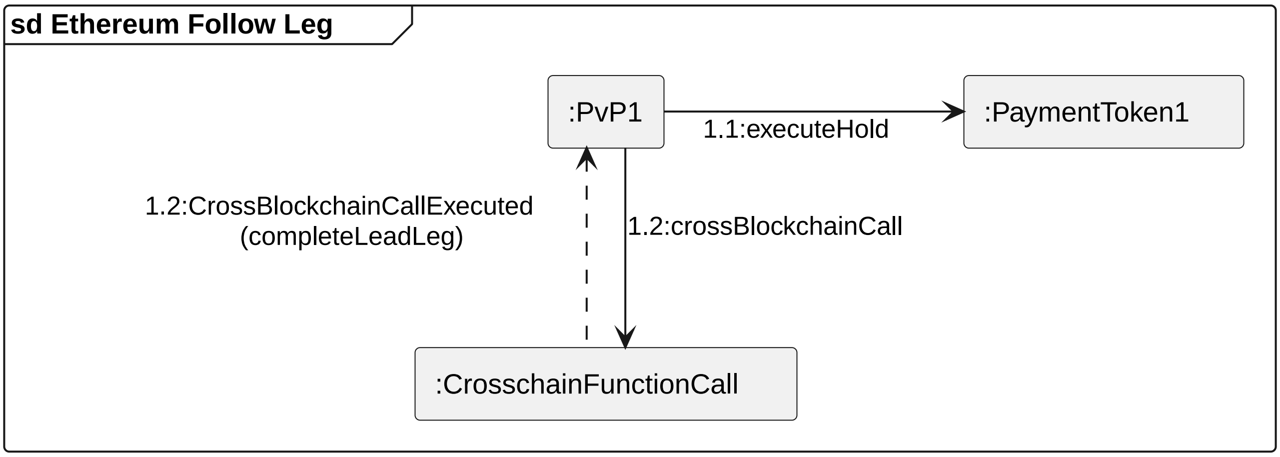

Figure 12. Follow Leg Communication

9.1.5. Off-Chain Follow Leg Orchestration

The orchestration communication in the Follow Leg Orchestration diagram describes the functionality provided by the off-chain `CrosschainFunctionCall` and `CrosschainMessaging` SDK components.
The `CrosschainFunctionCall` SDK picks up the event emitted by the `CrosschainFunctionCall` contract and then calls the `CrosschainMessaging` SDK.
The `CrosschainMessaging` SDK then creates the Merkle Patricia tree proof, which is returned to the `CrosschainFunctionCall` SDK.

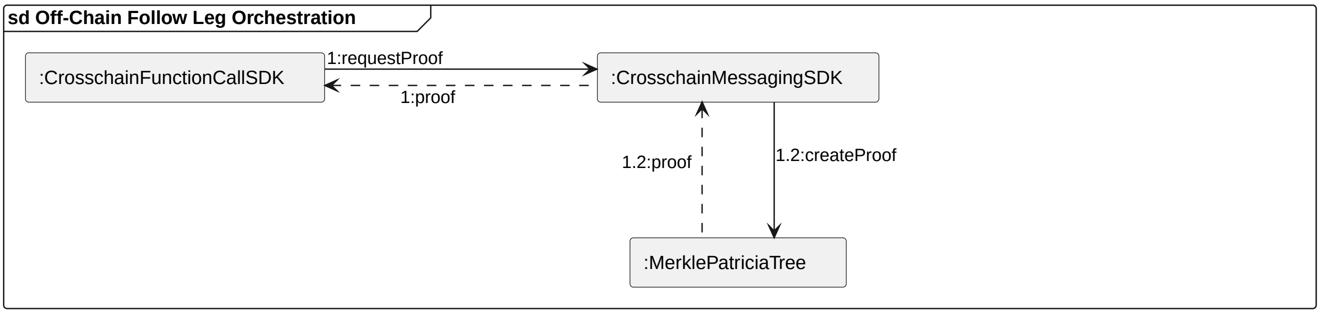

Figure 13. Follow Leg Orchestration

9.1.6. Chain A Complete Lead Leg

The final process completes the lead leg payment on chain A, which is shown in the Complete Lead Leg diagram.
The `CrosschainFunctionCall` SDK reads the block header that contains the event and submits the block header, event, and proof using the `performCallFromRemoteChain` function.
The `CrosschainFunctionCall` contract's `decodeBlockHeaderTransferEventAndProof` function will verify the validator signatures on the block header and persist the receipts root contained in the block header.
The `CrosschainMessaging` contract's `decodeAndVerifyEvent` function then verifies the Merkle Patricia proof by checking that its root is the same as the receipts root in the block header.
The `completeLeadLeg` function is called, which was obtained from the data emitted in the event from the `CrosschainFunctionCall` contract in the follow payment process.
It will execute the `PvP0` hold that Alice placed, transferring the tokens.

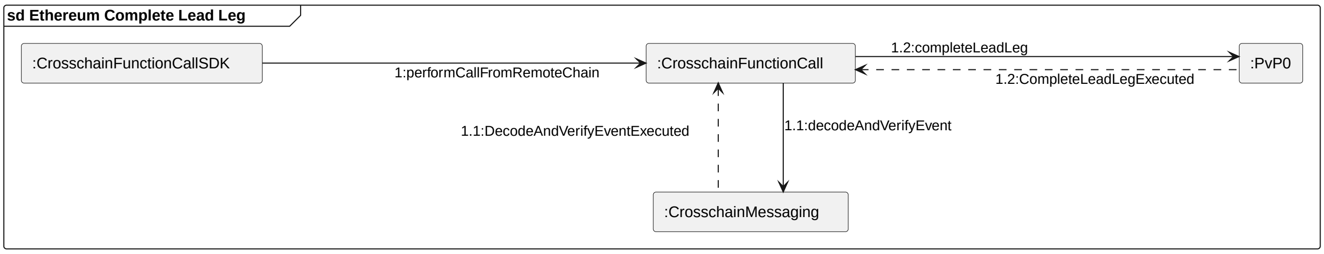

Figure 14. Complete Lead Leg

9.2. Unhappy Paths

The communication between the components involved in the leader-follower PvP unhappy paths can be identified as follows:

1. Start cancellation
2. Perform cancellation

9.2.1. Start Cancellation

The Start Cancellation diagram shows the interactions between the Application Layer and Crosschain Function Call Layer on Ethereum. The off-chain `Application` SDK calls the `startCancellation` function on the `PvP` contract. The `PaymentDvP` contract cancels the hold by calling the `cancelHold` function on the `PaymentToken` contract. Cancelling the hold is then followed by a call to the `crossBlockchainCall` function on the `CrosschainFunctionCall` contract. The `crossBlockchainCall` will emit an event containing the id for the remote chain, the remote `PvP` contract address, and the `performCancellation` call data. The event will then be monitored for and consumed by the off-chain `CrosschainFunctionCall` SDK.

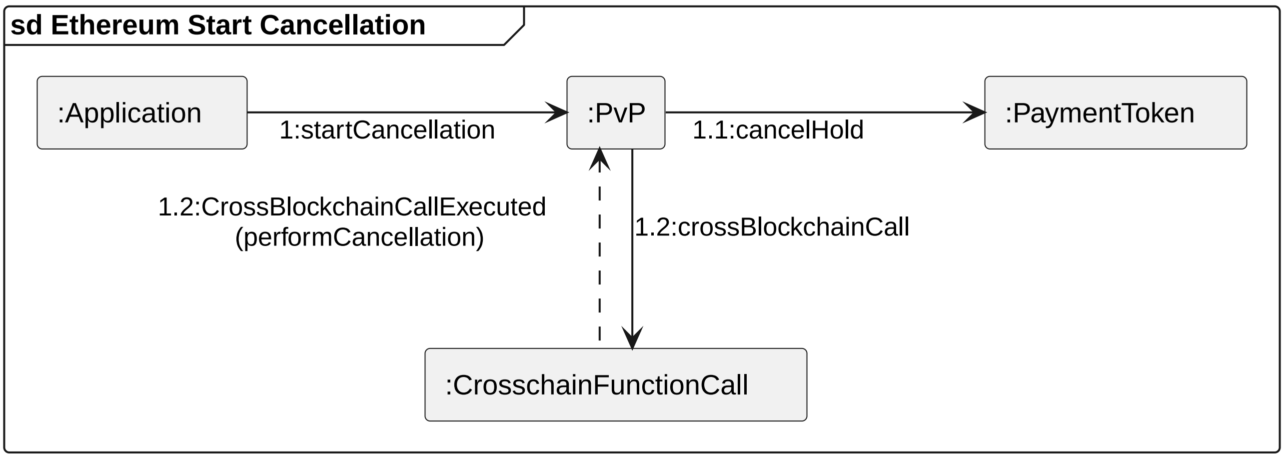

Figure 15. Start Cancellation

9.2.2. Perform Cancellation

The Follow Leg Perform Cancellation diagram shows the interactions between the Application Layer and the PvP contract on Ethereum. The off-chain `Application` SDK calls the `performCancellation` function on the `PvP` contract. The `DvP` contract cancels the hold by calling the `cancelHold` function on the `PaymentToken` contract. The token contract will emit a `CancelHoldExecuted` event which is monitored for and consumed by the off-chain `Application` SDK.

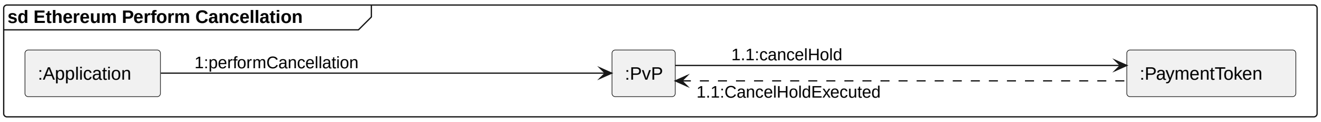

Figure 16. Perform Cancellation

Chapter 10. DvP Leader-Follower Happy Path

The Leader-Follower Delivery versus Payment (DvP) provides a complete view of the happy path delivery versus payment process.
A detailed description of the communication between the components is given in the Component Communication chapter.

The activity diagram shows a transfer of securities in a ledger on a Corda blockchain.
The serialized transaction data emitted by that transaction will then be used by a DvP contract, which is deployed on an Ethereum blockchain in order to handle the payment leg of the transaction.
The aim is for the DvP contract to verify the delivery of securities between two participants on the Corda chain, without direct interaction.
The processing is spread across three layers called the Application layer, Crosschain Function Call layer and Crosschain Messaging layer.
Each layer is represented in off-chain and on-chain components.

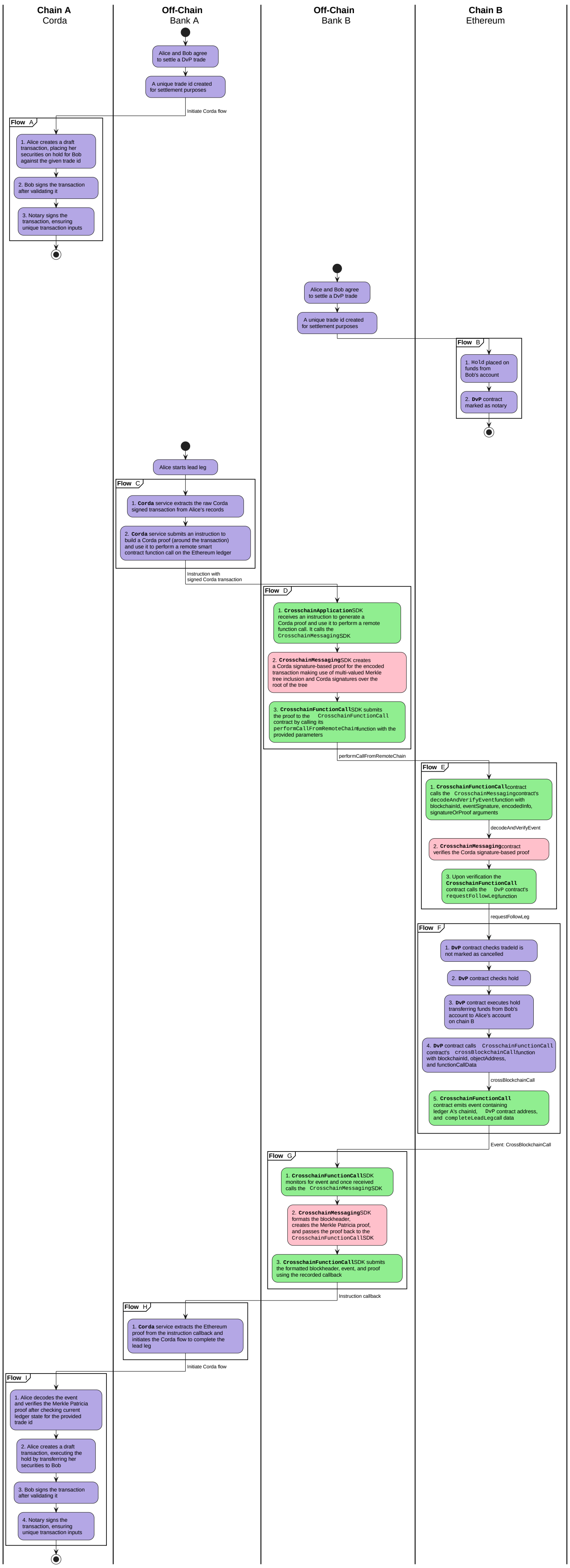

Figure 17. Leader-Follower Delivery versus Payment (DvP)

Chapter 11. DvP Leader-Follower Unhappy Paths

The following section describes the support provided when dealing with unhappy paths encountered during the settlement of a DvP trade.
For each unhappy scenario it is given that a unique trade id was agreed upon for settlement purposes.

11.1. States Automatically Recoverable

An unhappy path is automatically recoverable when the error produced is addressed by the SDK that encountered the error and the DvP trade can then be retried.

11.1.1. Feature 1: Transaction Failure Support

The transaction failure support recoverable unhappy paths are defined according to the following assumptions:

- There are no service interruptions when interacting with the two ledgers, for example, client connection loss.
- There are no cancellations by either party.

The following sections define the possible scenarios of transaction failure support within this feature.

Scenario 1: Insufficient Gas when Placing Hold

The unhappy path in the table describes the scenario when chain B has insufficient gas in order to place the hold.

Table 31. Scenario for Insufficient Gas when Placing Hold

Step	Description
Given	the hold is not placed on chain B
And	insufficient gas provided on chain B
When	placing the hold
And	the hold transaction reverts
Then	the gas amount is increased
And	placing the hold is retried

Scenario 2: Lead Leg SDK restart

The unhappy path in the table describes the scenario when the lead leg SDK restarts and tries to reprocess the AMQP\1.0 encoded signed transaction event from chain A.

Table 32. Scenario for Lead Leg SDK restart

Step	Description
Given	a successfully checked hold on chain A
And	a AMQP\1.0 encoded signed transaction event is emitted from chain A
And	the `CrosschainFunctionCall` SDK is restarted
And	the AMQP\1.0 encoded signed transaction event is processed twice by the `CrosschainFunctionCall` SDK
When	checking the hold on chain B fails
And	the transaction reverts
Then	the flow is aborted

Scenario 3: Insufficient Gas when Verifying the Lead Leg

The unhappy path in the table describes the scenario when the lead leg details cannot be verified due to insufficient gas.

Table 33. Scenario for Insufficient Gas when Verifying the Lead Leg

Step	Description
Given	a successfully placed hold on chain B
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	insufficient gas provided to decode and verify the event on chain B
Or	insufficient gas provided to check the tradeId and hold
When	decoding and verifying the event
Or	checking the tradeId and hold
And	the transaction reverts
Then	the gas amount is increased
And	verifying the lead leg is retried

Scenario 4: Insufficient Gas when Executing the Follow Leg

The unhappy path in the table describes the scenario when the follow leg hold cannot be executed due to insufficient gas.

Table 34. Scenario for Insufficient Gas when Executing the Follow Leg

Step	Description
Given	a successfully placed hold on chain B
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	the event has been decoded and verified

Step	Description
And	the tradeId and hold has been checked
And	there is insufficient gas
When	executing the hold
And	the transaction reverts
Then	the gas amount is increased
And	the instruction to `performCallFromRemoteChain` is resubmitted on chain B

Scenario 5: Follow Leg SDK restart

The unhappy path in the table describes the scenario when the follow leg SDK restarts and tries to reprocess the
`CrossBlockchainCallExecuted` event from chain B.

Table 35. Scenario for Follow Leg SDK restart

Step	Description
Given	a successfully executed hold chain B
And	a `CrossBlockchainCallExecuted` event is emitted from chain B
And	the `CrosschainFunctionCall` SDK is restarted
And	the `CrossBlockchainCallExecuted` event is processed twice by the `CrosschainFunctionCall` SDK
When	checking the hold on chain A fails
And	the transaction reverts
Then	the flow is aborted

Scenario 6: Insufficient Gas when Verifying the Follow Leg

The unhappy path in the table describes the scenario when the follow leg details cannot be verified due to the notary rejecting the transaction.

Table 36. Scenario for Insufficient Gas when Verifying the Follow Leg

Step	Description
Given	a successfully placed hold on chain A
And	an instruction to `performCallFromRemoteChain` submitted on chain A
And	the notary rejects the transaction
When	decoding and verifying the event
And	the transaction reverts
Then	the reason for rejection is addressed

Step	Description
And	the instruction to `performCallFromRemoteChain` is resubmitted on chain A

11.1.2. Feature 2: Service Interruption Support

The transaction failure support for unhappy paths which are automatically recoverable are defined according to the following assumptions:

- The services can be restarted when service interruptions occur while interacting with the two ledgers.
- There are no cancellations by either party.

The following section defines the possible service interruption support scenario within this feature.

Scenario 1: Inability to Submit Transactions

The unhappy path in the table describes the scenario when transactions cannot be submitted to a ledger due to service interruptions.

Table 37. Scenario for Inability to Submit Transactions

Step	Description
Given	a settlement trade in-progress
And	the ledger services are unavailable
When	submitting a transaction on chain A
Or	submitting a transaction on chain B
And	the transaction reverts
Then	the system will wait
And	executing the transaction is retried

11.2. States Requiring Intervention

When an unhappy path produces and error that cannot be addressed by the SDK, the error will either be addressed and the DvP trade retried, or alternatively, the trade will be cancelled and any changes to state are rolled back.

11.2.1. Feature 1: Transaction Failure Support

The transaction failure support unhappy paths requiring intervention are defined according to the following assumptions:

- There are no service interruptions when interacting with the two ledgers, for example, client connection loss.
- There are no cancellations by either party.

The following sections define the possible scenarios of transaction failure support within this feature.

Scenario 1: Insufficient Funds on Chain B

The unhappy path in the table describes the scenario when chain B has an insufficient account balance or insufficient tokens in order to place the hold.

Table 38. Scenario for Insufficient Funds on Chain B

Step	**Description**
Given	insufficient account balance on chain B
Or	insufficient tokens on chain B
And	a successfully placed hold on chain A
When	placing a hold on chain B
Then	the hold transaction reverts
And	the flow is terminated

Scenario 2: Insufficient Funds on Chain A

The unhappy path in the table describes the scenario when chain A has an insufficient account balance or insufficient securities in order to place the hold.

Table 39. Scenario for Insufficient Funds on Chain A

Step	Description
Given	insufficient account balance on chain A
Or	insufficient securities on chain A
And	a successfully placed hold on chain B
When	placing a hold on chain A
Then	the DCR transaction is not notarised
And	Bob does not sign the DCR transaction
And	the hold transaction reverts
And	the flow is terminated

Scenario 3: Failing to Decode the Event on Chain B

The unhappy path in the table describes the scenario where the event received by chain B cannot be decoded.

Table 40. Scenario for Failing to Decode the Event on Chain B

Step	**Description**
Given	a successfully placed hold on chain A
And	a successfully placed hold on chain B

Step	Description
And	a AMQP\1.0 encoded signed transaction event is received by the `CrosschainFunctionCall` SDK
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	the `performCallFromRemoteChain` event cannot be decoded
When	decoding and verifying the event
And	the transaction reverts
Then	the flow is terminated

Scenario 4: Failing to Verify the Signature Based Proof on Chain B

The unhappy path in the table describes the scenario where the proof received by chain B cannot be verified.

Table 41. Scenario for Failing to Verify the Signature Based Proof on Chain B

Step	Description
Given	a successfully placed hold on chain A
And	a successfully placed hold on chain B
And	Bob does not sign the DCR transaction on chain A
And	a AMQP\1.0 encoded signed transaction event is received by the `CrosschainFunctionCall` SDK
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	the `performCallFromRemoteChain` proof cannot be verified
When	verifying the multi-signature based proof
And	the transaction reverts
Then	the flow is terminated

Scenario 5: Failing TradeId or Hold Check on Chain B

The unhappy path in the table describes the scenario where the event received by chain B contains the incorrect tradeId or amount.

Table 42. Scenario for Failing TradeId or Hold Check on Chain B

Step	Description
Given	a successfully placed hold on chain A
And	a successfully placed hold on chain B
And	Bob signs the DCR transaction on chain A
And	the notary signs the DCR transaction on chain A

Step	Description
And	an instruction to `performCallFromRemoteChain` submitted on chain B
And	the event has been decoded and verified
And	the tradeId is incorrect
Or	the hold amount is incorrect
When	checking the tradeId and hold
Then	the transaction reverts
And	the flow is terminated

Scenario 6: Failing to Decode the Event on Chain A

The unhappy path in the table describes the scenario where the event received by chain A cannot be decoded.

Table 43. Scenario for Failing to Decode the Event on Chain A

Step	Description
Given	a successfully executed hold on chain B
And	a `CrossBlockchainCallExecuted` event is received by the `CrosschainFunctionCall` SDK
And	an instruction to `performCallFromRemoteChain` submitted on chain A
And	the `performCallFromRemoteChain` event cannot be decoded
When	decoding and verifying the event
And	the transaction reverts
Then	the flow is terminated

Scenario 7: Failing to Verify the Merkle Patricia Proof on Chain A

The unhappy path in the table describes the scenario where the proof received by chain A cannot be verified.

Table 44. Scenario for Failing to Verify the Merkle Patricia Proof on Chain A

Step	Description
Given	a successfully executed hold on chain B
And	a `CrossBlockchainCallExecuted` event is received by the `CrosschainFunctionCall` SDK
And	an instruction to `performCallFromRemoteChain` submitted on chain A
And	the `performCallFromRemoteChain` proof cannot be verified
When	verifying the Merkle Patricia proof

Step	Description
And	the transaction reverts
Then	the flow is terminated

11.3. Cancellations

Cancelling a DvP trade can be used as a mechanism to resolve unhappy paths requiring intervention which will result on the error state being rolled back.
For example, if Alice and Bob have agreed to settle a trade, but after some time has passed, either Alice or Bob do not respond then the other party can decide to cancel the trade.

The cancellation unhappy paths are defined according to the following assumptions:

- There are no service interruptions on either chain A or chain B
- There are no gas-related transaction failures
- Trades can only be cancelled using the unique trade id
- Cancelling a trade is initiated on the system or chain where the hold does not exist.
  Once holds exists on both sides, the trade can no longer be cancelled.

Feature 1 describes the cancellation scenarios for cancellations started on the follow ledger and Feature 2 describes the scenarios started on the lead ledger.

The Leader-Follower DvP Unhappy Paths for Cancellations Started on the Follow Leg diagram shows the leader-follower DvP flows for cancelling a DvP trade on the follow ledger.

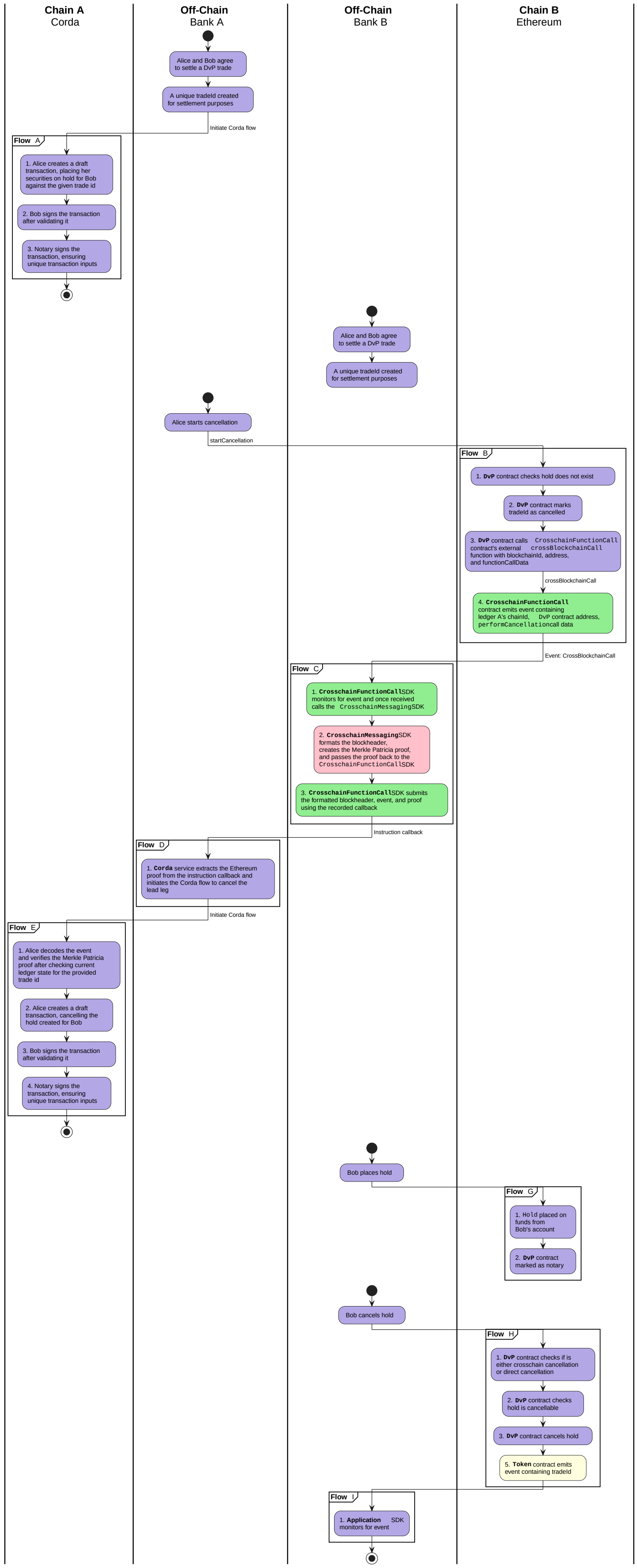

Figure 18. Leader-Follower DvP Unhappy Paths for Cancellations Started on the Follow Leg

11.3.1. Feature 1: Cancellation started on the follow ledger

Scenario 1: Alice starts a crosschain cancellation before the hold is placed on the follow ledger

The table describes the scenario shown in flow A to F in the diagram, which is the case where a trade cancellation is initiated on chain B before the hold is placed.

Table 45. Scenario when Alice starts a crosschain cancellation before the hold is placed on the follow ledger

Step	Description
Given	the hold placed on chain A
And	the hold not placed on chain B
When	Alice executes `dvp.startCancellation()` on chain B
And	the DvP contract marks the tradeId as cancelled on chain B
And	the DvP contract on chain B calls `crosschainFunctionCall.crossBlockchainCall()` with the `dvp.performCancellation()` function call data
And	the `CrosschainFunctionCall` SDK for bank A picks up the `CrossBlockchainCallExecuted` event
And	the `CrosschainMessaging` SDK formats the blockheader and creates the Merkle Patricia proof
Then	chain A receives the crosschain function call data as an attachment to a DCR transaction
And	the Merkle Patricia proof is verified on chain A
And	the hold is cancelled on chain A
And	the `Application` SDK extracts serialized data containing cancellation
And	the flow is terminated

Scenario 2: Bob performs a direct cancellation after the trade has been cancelled

The table describes the scenario shown in flow G to I in the diagram, which is the case where a hold is cancelled after it was placed on the follow ledger following a crosschain cancellation.

Table 46. Scenario when Bob performs a direct cancellation after the trade has been cancelled

Step	Description
Given	Alice cancelled the trade on chain B
And	Bob places the hold on chain B
When	Bob calls `dvp.performCancellation()` on chain B

Step	Description
And	the DvP checks the tradeId was marked as cancelled
Then	the DvP contract cancels the hold
And	the `Application` SDK picks up `CancelHoldExecuted` event
And	the flow is terminated

Scenario 3: Alice starts a crosschain cancellation after the hold is placed on the follow ledger

The table describes the scenario shown in flow B in the diagram, which is the case where a trade cancellation attempt is initiated on the follow ledger after the hold has already been placed on the follow ledger.

Table 47. Scenario when Alice starts a crosschain cancellation after the hold is placed on the follow ledger

Step	Description
Given	the hold placed on chain A
And	the hold placed on chain B
When	Alice executes `dvp.startCancellation()` on chain B
And	the DvP contract checks that the hold does not exist on chain B
Then	the DvP contract reverts
And	the flow is terminated

Scenario 4: Bob performs a direct cancellation before the trade has been cancelled

The table describes the scenario shown in flow I in the diagram, which is the case where a direct cancellation is attempted on the follow ledger while the trade still exists on the lead ledger.

Table 48. Scenario when Bob performs a direct cancellation before the trade has been cancelled

Step	Description
Given	Bob places the hold on chain B
And	the trade has not been cancelled
When	Bob calls `dvp.performCancellation()` on chain B
And	the DvP checks the function call came from the `crosschainFunctionCall` contract or the tradeId was marked as cancelled
Then	the DvP contract reverts
And	the flow is terminated

11.3.2. Feature 2: Cancellation started on the lead ledger

The following section describes the scenarios for cancellations started on the lead ledger.

The Leader-Follower DvP Unhappy Paths for Cancellations Started on the Lead Leg diagram shows the cancellation unhappy paths for Leader-Follower DvP flows for cancelling a DvP trade on the lead ledger.

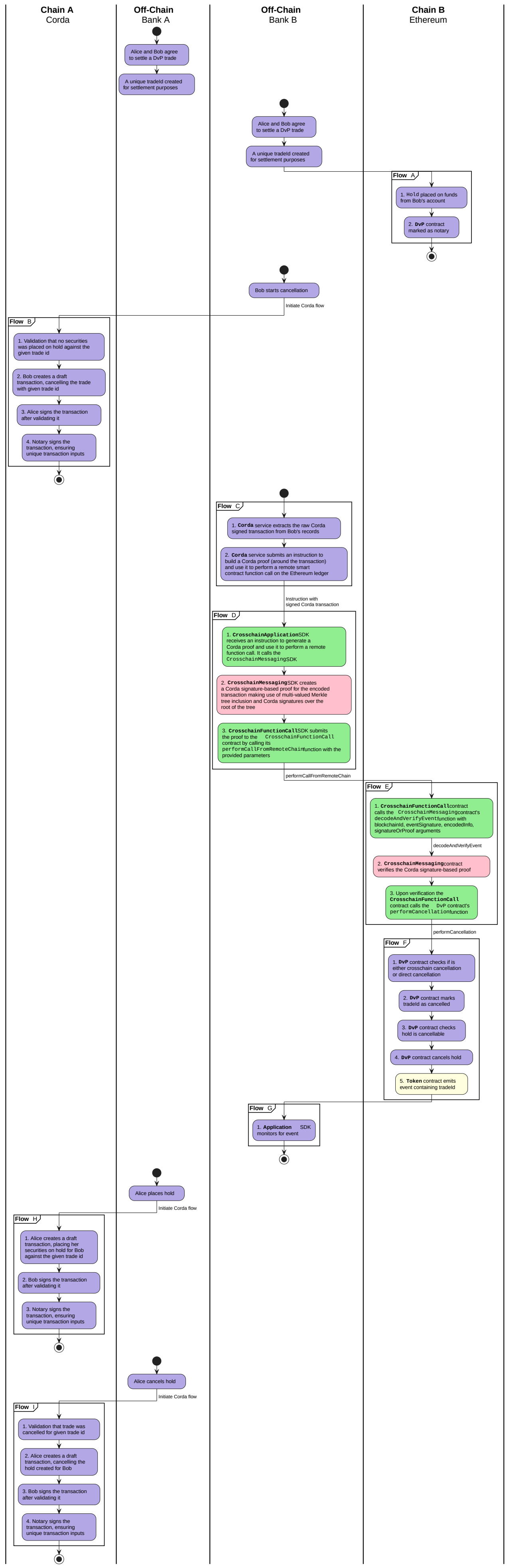

Figure 19. Leader-Follower DvP Unhappy Paths for Cancellations Started on the Lead Leg

Scenario 1: Bob starts a crosschain cancellation before the hold is placed on the lead ledger

The table describes the scenario shown in flow A to G in the diagram, which is the case where a trade cancellation is initiated on chain A before the hold is placed.

Table 49. Scenario when Bob starts a crosschain cancellation before the hold is placed on the lead ledger

Step	Description
Given	the hold placed on chain B
And	the hold not placed on chain A
When	Bob starts a cancellation on chain A
And	the tradeId is marked as cancelled on chain A
And	the serailized data containing the performCancellation call data is extracted from chain A
And	the `CrosschainFunctionCall` SDK for bank B picks up the encoded signed transaction as an event
And	the `CrosschainMessaging` SDK creates a multiple DSA signature based proof
Then	`crosschainFunctionCall.performCallFromRemoteChain()` is called on chain B
And	the proof is verified on chain B by the `crosschainMessaging.decodeAndVerifyEvent()` function
And	the `CrosschainFunctionCall` contract calls `dvp.performCancellation()` on chain B
And	the DvP contract cancels the hold on chain B
And	the `Application` SDK picks up `CancelHoldExecuted` event
And	the flow is terminated

Scenario 2: Alice performs a direct cancellation after the trade has been cancelled

The table describes the scenario shown in flow H to J in the diagram, which is the case where a hold is cancelled after it was placed on the lead ledger following a crosschain cancellation.

Table 50. Scenario when Alice performs a direct cancellation after the trade has been cancelled

Step	Description
Given	Bob cancelled the trade on chain A
And	Alice places the hold on chain A
When	Alice performs a cancellation on chain A
And	tradeId was marked as cancelled
Then	the hold is cancelled

Step	Description
And	the `Application` SDK extracts the serialized data containing the cancellation
And	the `Application` SDK picks up the encoded event
And	the flow is terminated

Scenario 3: Bob starts a crosschain cancellation after the hold is placed on the lead ledger

The table describes the scenario shown in flow A in the diagram, which is the case where a crosschain cancellation is attempted after the hold has already been placed on the lead ledger.

Table 51. Scenario when Bob starts a crosschain cancellation after the hold is placed on the lead ledger

Step	Description
Given	the hold placed on chain B
And	the hold placed on chain A
When	Bob starts a cancellation on chain A
And	the check that the hold does not exist on chain A fails
Then	the cancellation fails
And	the flow is terminated

Scenario 4: Alice performs a direct cancellation before the trade has been cancelled

The table describes the scenario shown in flow H in the diagram, which is the case where a direct cancellation is attempted on the lead ledger while the trade still exists on the follow ledger.

Table 52. Scenario when Alice performs a direct cancellation before the trade has been cancelled

Step	Description
Given	Alice places the hold on chain A
And	the trade has not been cancelled
When	Alice performs a cancellation on chain A
And	the checks the cancellation came from a remote chain or that the tradeId is marked as cancelled fails
Then	the cancellation fails
And	the flow is terminated

Chapter 12. DvP Component Communication

The following sections describe the happy and unhappy path communication diagrams, detailing the specific interactions between components in the leader-follower DvP process.

12.1. Happy Path

The communication between the components involved in the leader-follower DvP happy path can be identified as follows:

1. Corda start lead leg delivery earmark
2. Off-chain lead leg orchestration
3. Ethereum request follow leg payment upon verification
4. Ethereum follow leg payment
5. Off-chain follow leg orchestration
6. Corda complete lead leg delivery upon verification

The lead leg delivery and complete lead leg delivery communications take place on the Corda chain, while the request follow leg payment and follow leg payment communications take place on the Ethereum chain. The orchestration communications involve the `CrosschainFunctionCall` and `CrosschainMessaging` SDKs, which prepares the data or event and proof before it is submitted to each chain.

12.1.1. Corda Start Lead Leg

Once a trade has been agreed on and a unique tradeId is created
for settlement purposes, the Corda ledger places the hold on Alice’s securities.
The Start Lead Leg diagram shows the interaction between the Application Layer and Crosschain Function Call Layer on Corda. The off-chain `Application` SDK is used to call the `startLeadLeg` function deployed on a `SecurityDvP` Corda ledger. It is followed by a call to a `crossBlockchainCall` function, which will emit the transaction data. Corda must also provide the parameters required by the Ethereum protocol to call functions on the Ethereum blockchain, which is the Ethereum chain id, its `SecurityDvP` contract address, and the `requestFollowLeg` call data. The transaction data will then be monitored for and consumed by the Crosschain Function Call SDK off-chain.

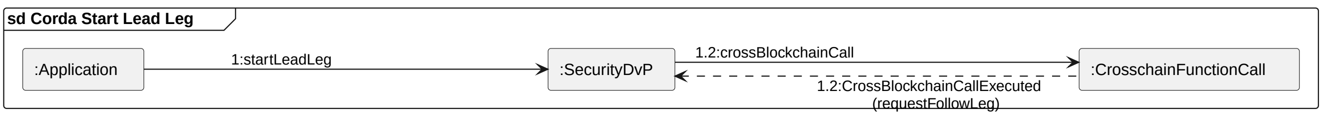

Figure 20. Start Lead Leg

12.1.2. Off-Chain Lead Leg Orchestration

The Corda off-chain orchestration shown in the Lead Leg Orchestration diagram makes use of functionality provided by the off-chain `CrosschainFunctionCall` and `CrosschainMessaging` SDK components. The `CrosschainFunctionCall` SDK picks up the transaction data emitted by the `crossBlockchainCall` function and then calls the `CrosschainMessaging` SDK. The `CrosschainMessaging`

SDK deserializes the data and submits the extracted proof components to the `CrosschainMessaging` contract on Ethereum by calling the `verifyAndStoreSignatures` function. This function will verify the validator signatures. The `CrosschainMessaging` SDK then creates the Corda Merkle tree proof, which is returned to the `CrosschainFunctionCall` SDK.

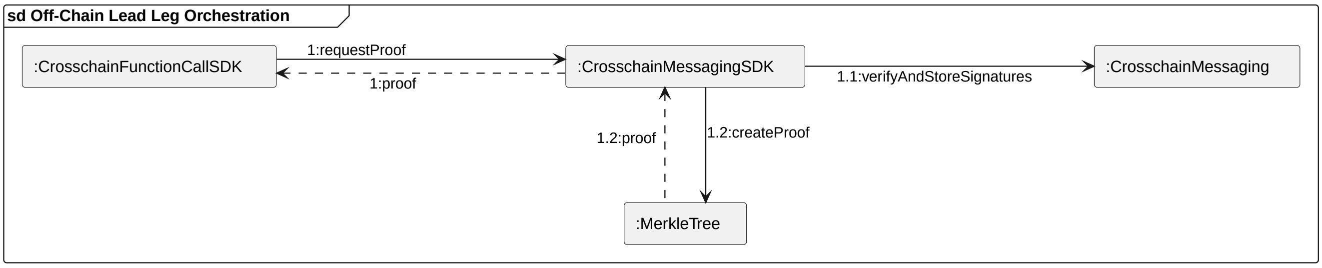

Figure 21. Lead Leg Orchestration

12.1.3. Ethereum Request Follow Leg

The delivery verification process shown in the Request Follow Leg diagram occurs once the `CrosschainFunctionCall` SDK has received the proof from the `CrosschainMessaging` SDK. The `CrosschainFunctionCall` SDK submits the event and the proof to the Ethereum chain using the `performCallFromRemoteChain` function. The `CrosschainMessaging` contract's `decodeAndVerifyEvent` function verifies the Merkle proof and then checks that the stored signatures are the same as those that signed the calculated Merkle tree root.

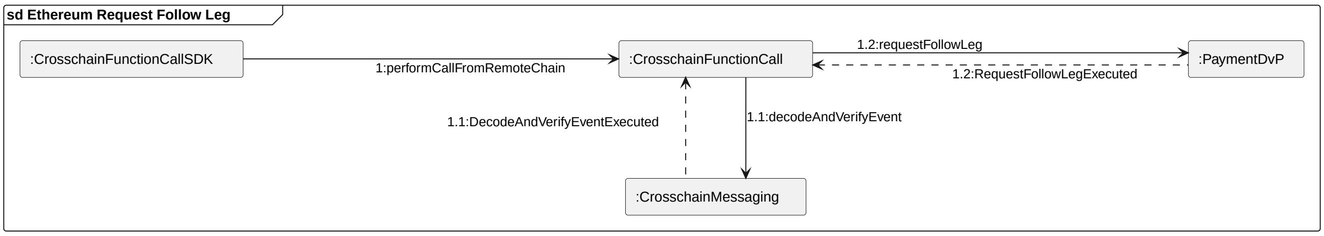

Figure 22. Request Follow Leg

12.1.4. Ethereum Follow Leg

The follow leg payment communications involves a call to the `requestFollowLeg` function, which was obtained from the function call data emitted in the data from the lead leg payment process. Once Hold amount is confirmed, the `PaymentDvP` contract executes the hold and then calls the `crossBlockchainCall` function on the `CrosschainFunctionCall` contract. The `crossBlockchainCall` will emit an event containing the id for the Corda chain, the `PaymentDvP` function reference on the Corda ledger, and the `completeLeadLeg` call data. The event will then be monitored for and consumed by the `CrosschainFunctionCall` SDK off-chain. The Follow Leg Communication diagram illustrates the follow leg payment process.

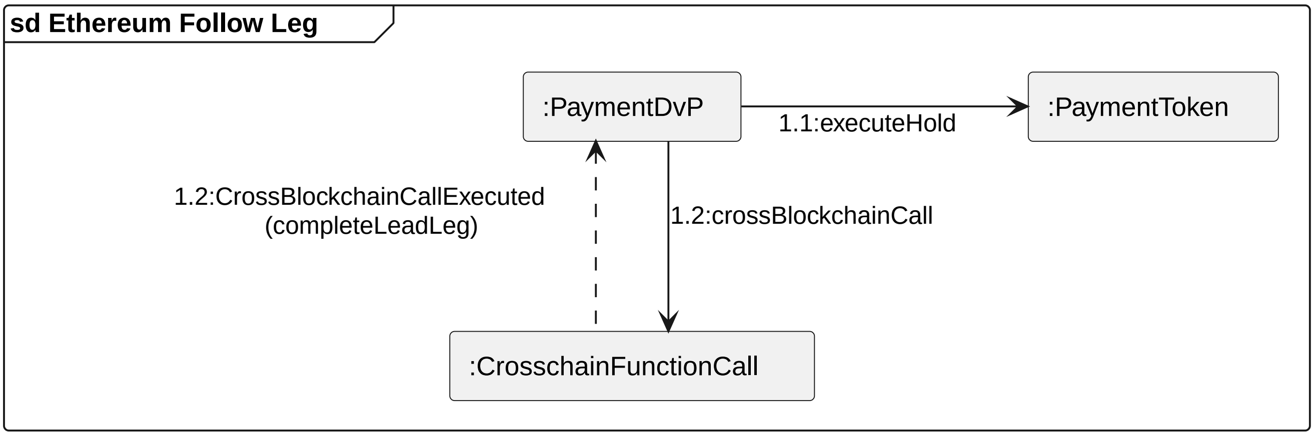

Figure 23. Follow Leg Communication

12.1.5. Off-Chain Follow Leg Orchestration

The Follow Leg Orchestration diagram describes the functionality provided by the off-chain `CrosschainFunctionCall` and `CrosschainMessaging` SDK components. The `CrosschainFunctionCall` SDK picks up the event emitted by the `CrosschainFunctionCall` contract and then calls the `CrosschainMessaging` SDK. The `CrosschainMessaging` SDK then creates the Merkle Patricia tree proof, which is returned to the `CrosschainFunctionCall` SDK.

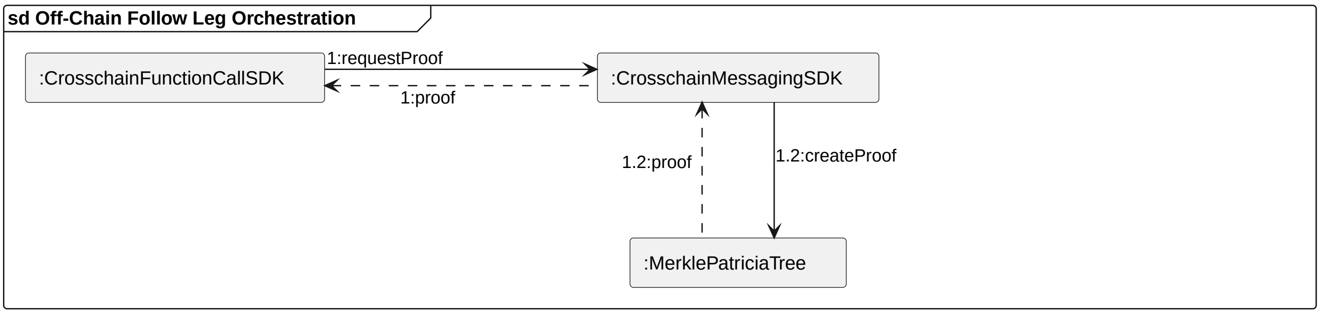

Figure 24. Follow Leg Orchestration

12.1.6. Corda Complete Lead Leg

The final process completes the lead leg delivery on Corda, which is shown in the Complete Lead Leg diagram. The `CrosschainFunctionCall` SDK reads the block header that contains the event and submits the block header, event, and proof to Corda using the `performCallFromRemoteChain` function. The `CrosschainFunctionCall` contract's `decodeBlockHeaderTransferEventAndProof` function will verify the validator signatures on the block header and persist the receipts root contained in the block header. The `CrosschainMessaging` contract's `decodeAndVerifyEvent` function verifies the Merkle Patricia proof by checking that its root is the same as the receipts root in the block header. The `completeLeadLeg` function is called, which was obtained from the data emitted in the event from the `CrosschainFunctionCall` contract in the follow payment process. It will execute the `SecurityDvP` hold that Alice placed, transferring the securities.

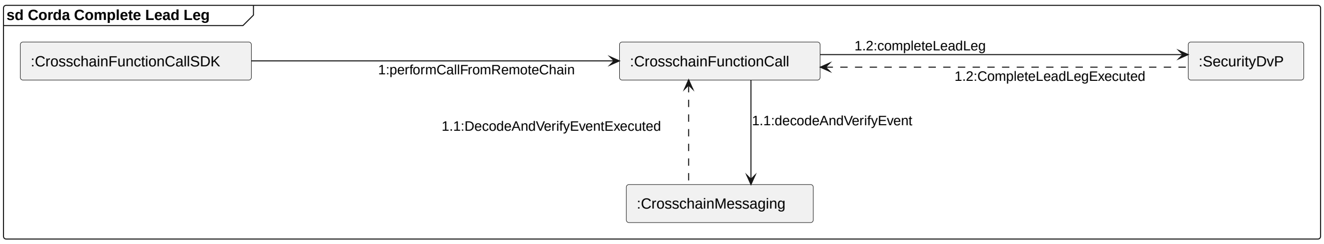

Figure 25. Complete Lead Leg

12.2. Unhappy Paths

The communication between the components involved in the leader-follower DvP unhappy paths can be identified as follows:

1. Follow leg start cancellation
2. Lead leg start cancellation
3. Follow Leg perform cancellation
4. Lead Leg perform cancellation

The follow leg start and perform cancellations take place on Ethereum and the lead leg start and perform cancellations take place on Corda.

12.2.1. Ethereum Start Cancellation

The Follow Leg Start Cancellation diagram shows the interactions between the Application Layer and Crosschain Function Call Layer on Ethereum. The off-chain `Application` SDK calls the `startCancellation` function on the `PaymentDvP` contract deployed on Ethereum. The `PaymentDvP` contract cancels the hold by calling the `cancelHold` function on the `PaymentToken` contract. Cancelling the hold is then followed by a call to the `crossBlockchainCall` function on the `CrosschainFunctionCall` contract. The `crossBlockchainCall` will emit an event containing the id for Corda, the security `DvP` contract address on Corda, and the `performCancellation` call data. The event will then be monitored for and consumed by the off-chain `CrosschainFunctionCall` SDK.

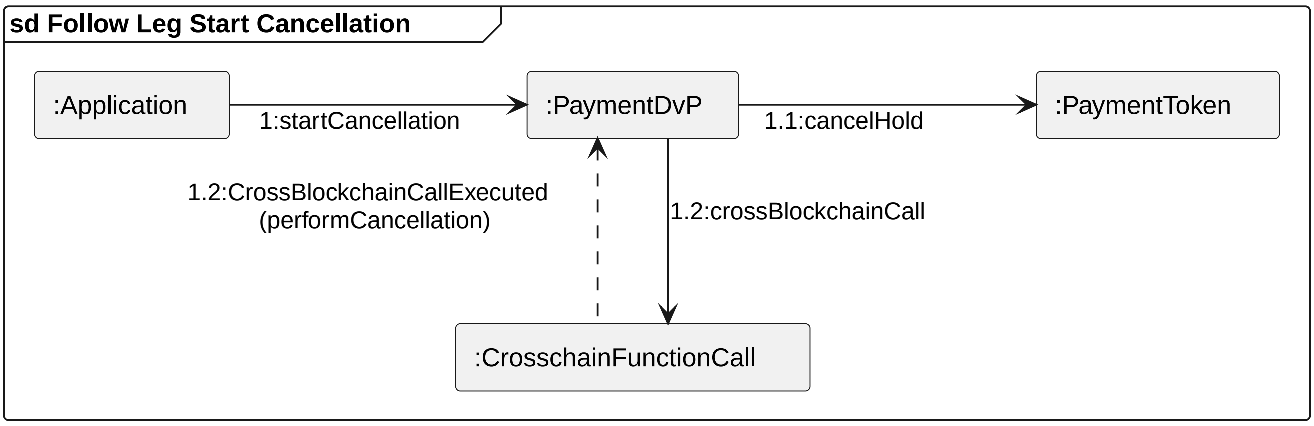

Figure 26. Follow Leg Start Cancellation

12.2.2. Corda Start Cancellation

The Lead Leg Start Cancellation diagram shows the interactions between the Application Layer and Crosschain Function Call Layer on Ethereum. The off-chain `Application` SDK calls the `startCancellation` function on the `SecurityDvP` contract on Corda. The `SecurityDvP` contract cancels the hold. The `Application` SDK will emit an event containing the id for the Ethereum chain, the payment `DvP` contract address on Ethereum, and the `performCancellation` call data. The event will then be monitored for and consumed by the off-chain `CrosschainFunctionCall` SDK.

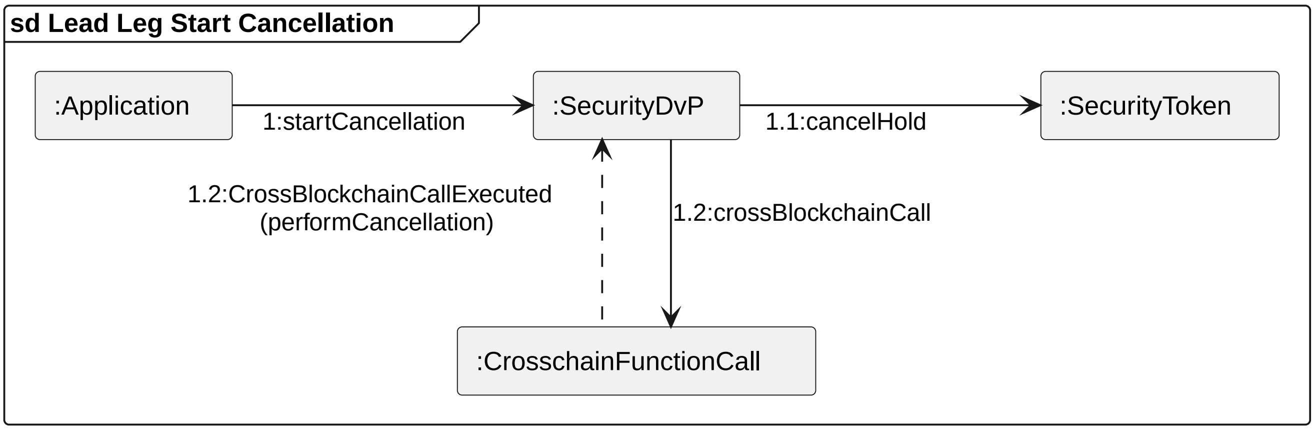

Figure 27. Lead Leg Start Cancellation

12.2.3. Ethereum Perform Cancellation

The Follow Leg Perform Cancellation diagram shows the interactions between the Application Layer and the DvP contract on Ethereum. The off-chain `Application` SDK calls the `performCancellation` function on the `PaymentDvP` contract deployed on Ethereum. The `PaymentDvP` contract cancels the hold by calling the `cancelHold` function on the `PaymentToken` contract. The token contract will emit a `CancelHoldExecuted` event which is monitored for and consumed by the off-chain `Application` SDK.

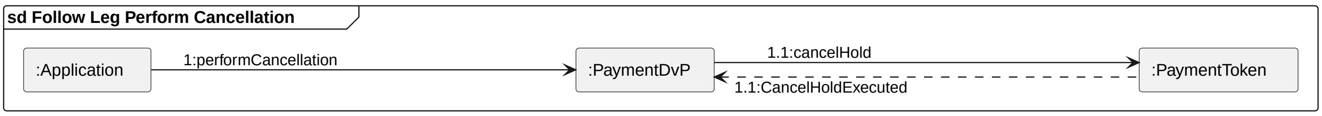

Figure 28. Follow Leg Perform Cancellation

12.2.4. Corda Perform Cancellation

The Lead Leg Perform Cancellation diagram shows the interactions between the Application Layer and the DvP contract on Corda. The off-chain `Application` SDK calls the `performCancellation` function on the `SecurityDvP` contract deployed on Corda. The `SecurityDvP` contract cancels the hold and the `Application` SDK will extract the `CancelHoldExecuted` data.

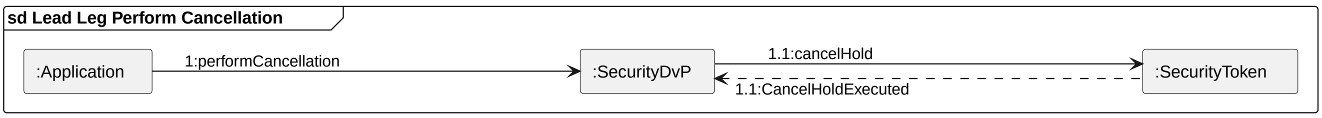

Figure 29. Lead Leg Perform Cancellation

Chapter 13. Ethereum Proof Creation

The process of creating the ethereum proof of the `CrossBlockchainCallExecuted` event involves the steps illustrated in the Create Ethereum Proof Activity UML Diagram, which is described in detail in the following sections.

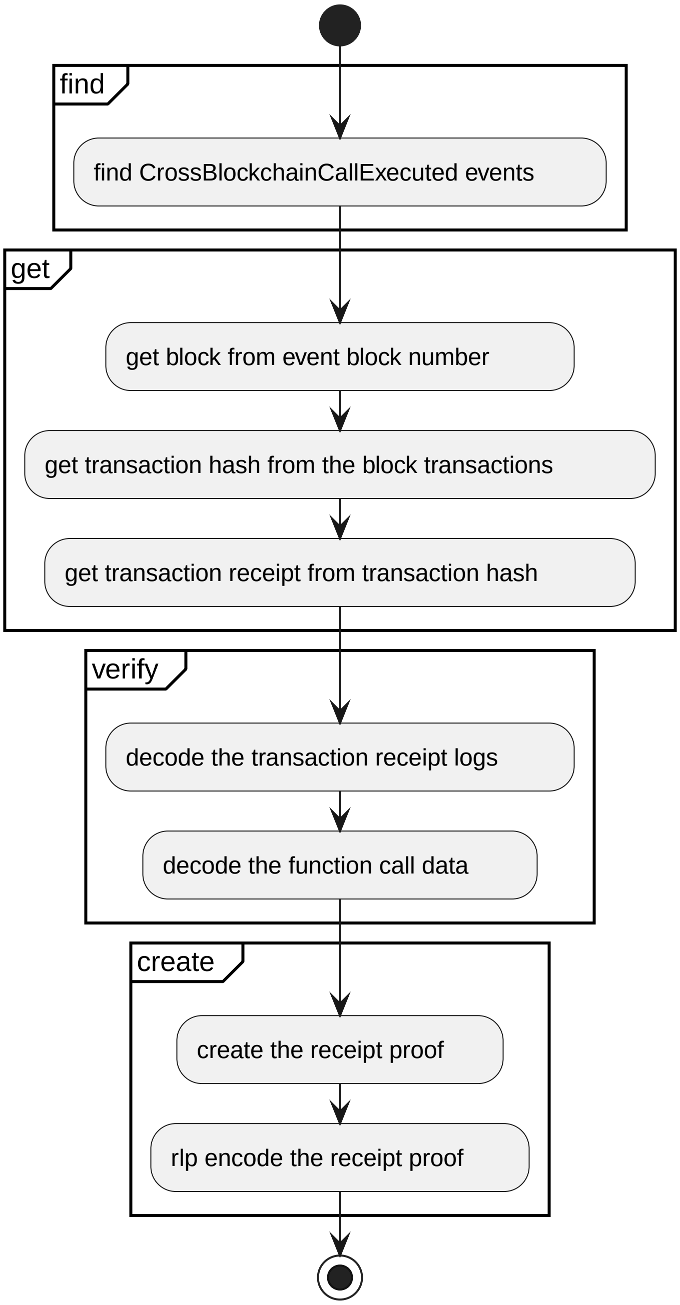

Figure 30. Create Ethereum Proof Activity UML Diagram

13.1. CrossBlockchainCallExecuted Event

When the `CrosschainFunctionCall` contract's external `crossBlockchainCall` function is called, it emits a `CrossBlockchainCallExecuted` event containing the `destinationBlockchainId`, XvP `contractAddress`, and the `functionCallData`.

CrossBlockchainCallExecuted Event

```
event CrossBlockchainCallExecuted(
  uint256 destinationBlockchainId,
  address contractAddress,
  bytes functionCallData
);
```

where:

1. `destinationBlockchainId` is the uint256 blockchain ID of the destination blockchain.
2. `contractAddress` is the 160-bit Ethereum address of the contract on the remote blockchain to be called.
3. `functionCallData` is the ABI encoded function signature and parameter data.

> The `functionCallData` encodes the function signature and parameter data of either the requestFollowLeg or completeLeadLeg function, depending on whether the event is being emitted from the lead or follow ledger, respectively.

13.2. Request Follow Leg Function

When the CrossBlockchainCallExecuted event is emitted from the lead ledger the `functionCallData` contained in its parameters encodes the function signature and parameter data of the `requestFollowLeg` function.

The `requestFollowLeg` function starts the follow leg of a trade, with the given trade details, as instructed from a remote chain. It returns true if the follow leg was successfully started.

requestFollowLeg function signature

```
function requestFollowLeg(
  string calldata tradeId,
  string calldata sender,
  string calldata receiver,
  address destinationContract,
  uint256 destinationBlockchainId,
  uint256 foreignNotional
)
```

where:

1. `tradeId` The trade identifier.
2. `sender` The sending party's foreign account identifier
3. `receiver` The receiving party's foreign account identifier.
4. `destinationContract` The destination contract address.
5. `destinationBlockchainId` The destination chain identifier.

6. `foreignNotional` The nominal value of the trade on the remote chain.

13.3. Complete Lead Leg Function

When the CrossBlockchainCallExecuted event is emitted from the follow ledger the `functionCallData` contained in its parameters encodes the function signature and parameter data of the `completeLeadLeg` function.

The `completeLeadLeg` function completes the lead leg of a trade, with the given trade details, as instructed from a remote chain. It returns true if the lead leg was successfully completed.

completeLeadLeg function signature

```
function completeLeadLeg(
  string calldata tradeId,
  string calldata sender,
  string calldata receiver,
  uint256 foreignNotional
)
```

where:

1. `tradeId` The trade identifier.
2. `sender` The sending party's foreign account identifier
3. `receiver` The receiving party's foreign account identifier.
4. `foreignNotional` The nominal value of the trade on the remote chain.

13.4. Find CrossBlockchainCallExecuted Events

Web3 can be used to find events emitted by a specific contract and filtered by event signature.

Find CrossBlockchainCallExecuted events

```
    latestBlock = await web3.eth.getBlock('latest')
    startingBlock = Math.max(latestBlock.number - 3000, 0)

    const eventsFound = await findCrossBlockchainCallExecutedEvent(startingBlock,
web3, contractAddress)
```

An array of events is returned by the function.

Example event

```
    {
      eventsFound: [
        {
        "decodedLog": {
```

```
          "0": "1",
          "1": "0x37bCb3CAc66F4d859a4eF77dcD97EEc146BBC425",
          "2":
"0x8903901f000000000000000000000000000000000000000000000000000000000000008000000000000
000000000000000000000000000000000000000000000000000c0000000000000000000000000000000000
0000000000000000000000000000100000000000000000000000000000000000000000000000000000000
000002710000000000000000000000000000000000000000000000000000000000000083765386231313
46500000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000b4854474247423030555344000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000b485455535553303
05553440000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000002ac3a5e96e4a09b6613273aca6649f453f9425712",
          "__length__": 3,
          "destinationBlockchainId": "1",
          "contractAddress": "0x37bCb3CAc66F4d859a4eF77dcD97EEc146BBC425",
          "functionCallData":
"0x8903901f000000000000000000000000000000000000000000000000000000000000008000000000000
000000000000000000000000000000000000000000000000000c0000000000000000000000000000000000
0000000000000000000000000000100000000000000000000000000000000000000000000000000000000
000002710000000000000000000000000000000000000000000000000000000000000083765386231313
46500000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000b4854474247423030555344000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000b485455535553303
05553440000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000002ac3a5e96e4a09b6613273aca6649f453f9425712"
        },
        "blockNumber": 632,
        "txHash":
"0x88742501c31825e7fd2a1fb650d34b6b64d2dc7e15cb08b9a16058daace64c19",
        "data":
"0x00000000000000000000000000000000000000000000000000000000000000010000000000000000000
0000037bcb3cac66f4d859a4ef77dcd97eec146bbc42500000000000000000000000000000000000000000
000000000000000000000600000000000000000000000000000000000000000000000000000000000000017
88903901f000000000000000000000000000000000000000000000000000000000000008000000000000000
0000000000000000000000000000000000000000000000000c000000000000000000000000000000000000
00000000000000000000000000100000000000000000000000000000000000000000000000000000000000
0002710000000000000000000000000000000000000000000000000000000000000008376538623131346
50000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000b48544742474230305553440000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000b48545553555330305
55344000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000002ac3a5e96e4a09b6613273aca6649f453f94257120000000000000000",
        "logIndex": 1
      }
      ]
    }
```

The `txHash` field will be used to retreive the transaction receipt and generate the Merkle Patricia tree proof using the eth-proof npm module.

The `findEvents` function implementation is shown below:

findEvents function

```
    async function findCrossBlockchainCallExecutedEvent(startingBlock, web3,
contractAddress){

      const CrosschainFunctionCallJson =
require('../../../../../../build/contracts/CrosschainFunctionCall.json')
      const eventName = 'CrossBlockchainCallExecuted'

      let eventABI = {}
      let eventSignature = eventName+'('
      for(let item of CrosschainFunctionCallJson.abi){
        if(item.name === eventName){
          eventABI = item
          for(let i in item.inputs){
            const input = item.inputs[i]
            eventSignature += input.type
            if(i < item.inputs.length -1){
              eventSignature += ','
            } else {
              eventSignature += ')'
            }
          }
        }
      }

      const filterTopics = [web3.utils.keccak256(eventSignature)]
      const eventLogs = await web3.eth.getPastLogs({fromBlock: startingBlock, toBlock:
'latest', address: contractAddress, topics: filterTopics})

      const decodedEventLogs = []
      for(let log of eventLogs){
        const decodedLog = web3.eth.abi.decodeLog(eventABI.inputs, log.data)
        decodedEventLogs.push({
          decodedLog, //TODO: clean up the decoded log to only contain the named
parameters?
          blockNumber: log.blockNumber,
          txHash: log.transactionHash,
          data: log.data,
          logIndex: log.logIndex
        })
      }
      return decodedEventLogs
    }
```

13.4.1. Get the Transaction Receipt from the Transaction Hash

The `web3.eth.getTransactionReceipt` function is used to return the receipt of a transaction by the transaction hash.

Get txReceipt

```
let txReceipt = await web3.eth.getTransactionReceipt(txHash)
```

Example txReceipt


```
{
  "txReceipt": {
    "blockHash":
"0xde80a72f94baa725464b746ea3bb8186cc30627536348a107f6ea0c12181e8b4",
    "blockNumber": 632,
    "contractAddress": null,
    "cumulativeGasUsed": 413939,
    "from": "0x1657e3999d263706af7349d1af7c6cb88d7e7ffa",
    "gasUsed": 413939,
    "effectiveGasPrice": 0,
    "logs": [
      {
        "address": "0x57F1DEfACaAfaa664BBB0f4F3347b786d874A25c",
        "topics": [
          "0xf0e25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067"
        ],
        "data":
"0x00000000000000000000000000000000000000000000000000000000000000200000000000000000000
0000000000000000000000000000000000000000000406133376531623037656637653063656262616335 3
13838306238633136633130393335326566313230623133643230393163313762656361613039303234366
6",
        "blockNumber": 632,
        "transactionHash":
"0x88742501c31825e7fd2a1fb650d34b6b64d2dc7e15cb08b9a16058daace64c19",
        "transactionIndex": 0,
        "blockHash":
"0xde80a72f94baa725464b746ea3bb8186cc30627536348a107f6ea0c12181e8b4",
        "logIndex": 0,
        "removed": false,
        "id": "log_e752b49f"
      },
      {
        "address": "0xc13141Df25aC03EB6E249aB4319F61d2Be3d4254",
        "topics": [
          "0x7a752c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92"
        ],
        "data":
"0x00000000000000000000000000000000000000000000000000000000000000010000000000000000000
0000037bcb3cac66f4d859a4ef77dcd97eec146bbc42500000000000000000000000000000000000000000
0000000000000000000006000000000000000000000000000000000000000000000000000000000000017
88903901f000000000000000000000000000000000000000000000000000000000000008000000000000000
0000000000000000000000000000000000000000000000000c000000000000000000000000000000000000
00000000000000000000000000100000000000000000000000000000000000000000000000000000000000
00027100000000000000000000000000000000000000000000000000000000000000008376538623131346
```

```
500000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000b4854474247423030555344000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000b48545553555330305
5534400000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000002ac3a5e96e4a09b6613273aca6649f453f94257120000000000000000",
"blockNumber": 632,
            "transactionHash":
"0x88742501c31825e7fd2a1fb650d34b6b64d2dc7e15cb08b9a16058daace64c19",
            "transactionIndex": 0,
            "blockHash":
"0xde80a72f94baa725464b746ea3bb8186cc30627536348a107f6ea0c12181e8b4",
            "logIndex": 1,
            "removed": false,
            "id": "log_e87bbddf"
          }
        ],
        "logsBloom":
"0x0000000000000000000400000000000000080040000200000040000000000000000000000000000000000000
00000000000020000000000000000000000000000000000000000000800000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000400000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000400000010000000000000000000000000000000000
0000000000000000000008000000000000000000000000000000000000001000000000000000000000000"
,
        "status": true,
        "to": "0xc13141df25ac03eb6e249ab4319f61d2be3d4254",
        "transactionHash":
"0x88742501c31825e7fd2a1fb650d34b6b64d2dc7e15cb08b9a16058daace64c19",
        "transactionIndex": 0,
        "type": "0x0"
      }
    }
```

13.5. Verify the Event from the Trade Details

The trade details can be used to verify the event before creating the receipt proof.

13.5.1. Decode the Transaction Receipt Logs

The `CrosschainFunctionCall` contract and `CrosschainXvP` contract ABI’s are manually added to the abi-decoder library in order to be able to decode parameters from the transaction receipt logs using the `decodeLogs` function.

Decode logs

```
    let decodedLogs = abiDecoder.decodeLogs(txReceipt.logs)
```

The decoded transaction receipt logs defines three fields, namely `name`, `events`, and `address`. The `events` field stores an array containing the CrossBlockchainCallExecuted Event parameters, including the functionCallData, which will also be decoded as shown in Section Decode the Function Call Data.

Example decodedLog


```
{
  "decodedLog": {
    "name": "CrossBlockchainCallExecuted",
    "events": [
      {
        "name": "destinationBlockchainId",
        "type": "uint256",
        "value": "1"
      },
      {
        "name": "contractAddress",
        "type": "address",
        "value": "0x37bcb3cac66f4d859a4ef77dcd97eec146bbc425"
      },
      {
        "name": "functionCallData",
        "type": "bytes",
        "value":
"0x8903901f00000000000000000000000000000000000000000000000000000000000000800000000000000
000000000000000000000000000000000000000000000000000c0000000000000000000000000000000000
0000000000000000000000000000100000000000000000000000000000000000000000000000000000000
0000027100000000000000000000000000000000000000000000000000000000000000083765386231313
4650000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000b48544742474230305553440000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000b485455535553303
0555344000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000002ac3a5e96e4a09b6613273aca6649f453f9425712"
      }
    ],
    "address": "0xc13141Df25aC03EB6E249aB4319F61d2Be3d4254"
  }
}
```


13.5.2. Decode the Function Call Data

By looping over the `events` field in the transaction receipt logs, it's possible to find the `functionCallData` parameter, so that its `value` field can be decoded.

Decode functionCallData

```
for (let param of decodedLog.events) {
  if (!!param && param.name === 'functionCallData') {
    const decodedFunctionCallData = abiDecoder.decodeMethod(param.value)
    console.log(JSON.stringify({decodedFunctionCallData}, null, 2))
  }
}
```

The decoded function call data contains the function `name` and `params` relating to the function and parameters of either the requestFollowLeg or completeLeadLeg function that was encoded in the `CrossBlockchainCallExecuted` event. These parameters can be used to match the event to specific trade details.

Example decodedFunctionCallData

```
{
  "decodedFunctionCallData": {
    "name": "completeLeadLeg",
    "params": [
      {
        "name": "tradeId",
        "value": "7e8b114e",
        "type": "string"
      },
      {
        "name": "sender",
        "value": "HTGBGB00USD",
        "type": "string"
      },
      {
        "name": "receiver",
        "value": "HTUSUS00USD",
        "type": "string"
      },
      {
        "name": "foreignNotional",
        "value": "10000",
        "type": "uint256"
      }
    ]
  }
}
```

13.6. Create

13.6.1. Create the Receipt Proof

Once it has been verified that the txHash returns the txReceipt containing the correct `CrossBlockchainCallExecuted` event in its logs, which matches the required trade details, then the `txHash` can be used together with the eth-proof module to create a receipt proof.

Get Receipt Proof

```
const ethProof = new GetProof(config[chainName].httpProvider)
let receiptProof = await ethProof.receiptProof(txHash)
```

13.6.2. RLP-Encode the Receipt Proof

Finally, the receipt proof is RLP encoded using the `encodedReceiptProof` function.

Encode Receipt Proof

```
encodedReceiptProof = encodeReceiptProof(receiptProof)
```

Example encodedReceiptProof


```
{
  encodedReceiptProof: {
    path: '0x1',
    rlpEncodedReceipt:
'0xf9040301830650f3b901000000000000000000000400000000000000000800400002000000400000000
00000000000000000000000000000000002000000000000000000000000000000000000000000080000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000004000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000400000010000000000000
00000000000000000000000000000000000000000080000000000000000000000000000000000000001000
000000000000000000000f902f8f8b99457f1defacaafaa664bbb0f4f3347b786d874a25ce1a0f0e25c639
81e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b880000000000000000000000000000
00000000000000000000000000000000000200000000000000000000000000000000000000000000000000
00000000000004061333765316230376566376530636562626163353138383062386331366331303933353
265663132306231336432303931633137626563616130393032343666f9023a94c13141df25ac03eb6e249
ab4319f61d2be3d4254e1a07a752c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a9
2b902000000000000000000000000000000000000000000000000000000000000000000010000000000000
00000000037bcb3cac66f4d859a4ef77dcd97eec146bbc4250000000000000000000000000000000000000
00000000000000000000000006000000000000000000000000000000000000000000000000000000000000
001788903901f000000000000000000000000000000000000000000000000000000000000080000000000
00000000000000000000000000000000000000000000000000000c00000000000000000000000000000000
00000000000000000000000000000010000000000000000000000000000000000000000000000000000000
0000000271000000000000000000000000000000000000000000000000000000000000000000837653862313
13465000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000b48544742474230305553440000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000b4854555355533
03055534400000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000002ac3a5e96e4a09b6613273aca6649f453f94257120000000000000000'
,
```

```
        witness:
'0xf9040ff9040c822080b90406f9040301830650f3b901000000000000000000000400000000000000080
04000020000004000000000000000000000000000000000000000000002000000000000000000000000000
00000000000000008000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000400000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00004000000100000000000000000000000000000000000000000000000000000080000000000000000000
000000000000000000001000000000000000000000000f902f8f8b99457f1defacaafaa664bbb0f4f3347b
786d874a25ce1a0f0e25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b880000
00000000000000000000000000000000000000000000000000000000000200000000000000000000000000
0000000000000000000000000000000000000406133376531623037656637653063656262616335313838 3
062386331366331303933353265663132306231336432303931633137626563616130393032343666f9023
a94c13141df25ac03eb6e249ab4319f61d2be3d4254e1a07a752c4d100a96be23f60f072fed1f33da2890f
be45eb15af52750faa2772a92b90200000000000000000000000000000000000000000000000000000000
0000000010000000000000000000000000037bcb3cac66f4d859a4ef77dcd97eec146bbc4250000000000000
00000000000000000000000000000000000000000000000006000000000000000000000000000000000000
000000000000000000000000001788903901f0000000000000000000000000000000000000000000000000
00000000000008000000000000000000000000000000000000000000000000000000000000000c00000000
00000000000000000000000000000000000000000000000000000010000000000000000000000000000000
00000000000000000000000000000002710000000000000000000000000000000000000000000000000000
00000000000083765386231313465000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000b4854474247423030555344000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000b4854555355533030555344000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000002ac3a5e96e4a09b6613273aca6649f453f
94257120000000000000000'
      }
    }
```

encodeReceiptProof function

```
    function encodeReceiptProof(proof){
      // the path is HP encoded
      const indexBuffer = proof.txIndex.slice(2);
      const hpIndex = '0x' + (indexBuffer.startsWith('0') ? '1' + indexBuffer.slice(1)
: '00' + indexBuffer);
      //const hpIndex = '0x30'
      // the value is the second buffer in the leaf (last node)
      const value = '0x' + Buffer.from(proof.receiptProof[proof.receiptProof.length -
1][1]).toString('hex');
      // the parent nodes must be rlp encoded
      const parentNodes = rlp.encode(proof.receiptProof);
      return {
        path: hpIndex,
        rlpEncodedReceipt: value,
        witness: '0x'+parentNodes.toString('hex')
      };
    }
```

13.7. Complete Code Example

Example

```
abiDecoder.addABI(CrosschainFunctionCallJson.abi)
abiDecoder.addABI(CrosschainXvPJson.abi)
let block
let encodedReceiptProof
let eventDetails = []
for (const event of eventsFound) {
  block = await web3.eth.getBlock(event.blockNumber)
  console.log(JSON.stringify({block}, null, 2))
  if (block != null) {
    let txHash = event.txHash
    console.log({txHash})
    let txReceipt = await web3.eth.getTransactionReceipt(txHash)
    console.log(JSON.stringify({txReceipt}, null, 2))
    if (txReceipt !== null) {
      let decodedLogs = abiDecoder.decodeLogs(txReceipt.logs)
      for (let decodedLog of decodedLogs) {
        if (decodedLog.name === 'CrossBlockchainCallExecuted') {
          console.log(JSON.stringify({decodedLog}, null, 2))
          for (let param of decodedLog.events) {
            if (!!param && param.name === 'functionCallData') {
              const decodedFunctionCallData = abiDecoder.decodeMethod(param.value)
              console.log(JSON.stringify({decodedFunctionCallData}, null, 2))
            }
          }
          const ethProof = new GetProof(config[chainName].httpProvider)
          let receiptProof = await ethProof.receiptProof(txHash)
          encodedReceiptProof = encodeReceiptProof(receiptProof)
          console.log({encodedReceiptProof})
        }
      }
    }
  }
}
```

13.8. Required NPM Packages

13.8.1. ABI-Decoder

The abi-decoder library is used for decoding data params and events from ethereum transactions.

13.8.2. RLP

The rlp library is used for Recursive Length Prefix encoding.

13.8.3. EthProof

The eth-proof library is a generalized merkle-patricia-proof module. If you have a single hash that you trust (i.e. blockHash), you can use this module to succinctly prove exactly what data was contained in the Ethereum blockchain at that snapshot in history.

The Get Receipt Proof Activity UML Diagram shows how the Patricia trie proof is generated from the sibling transactions hashes inside the block transactions.

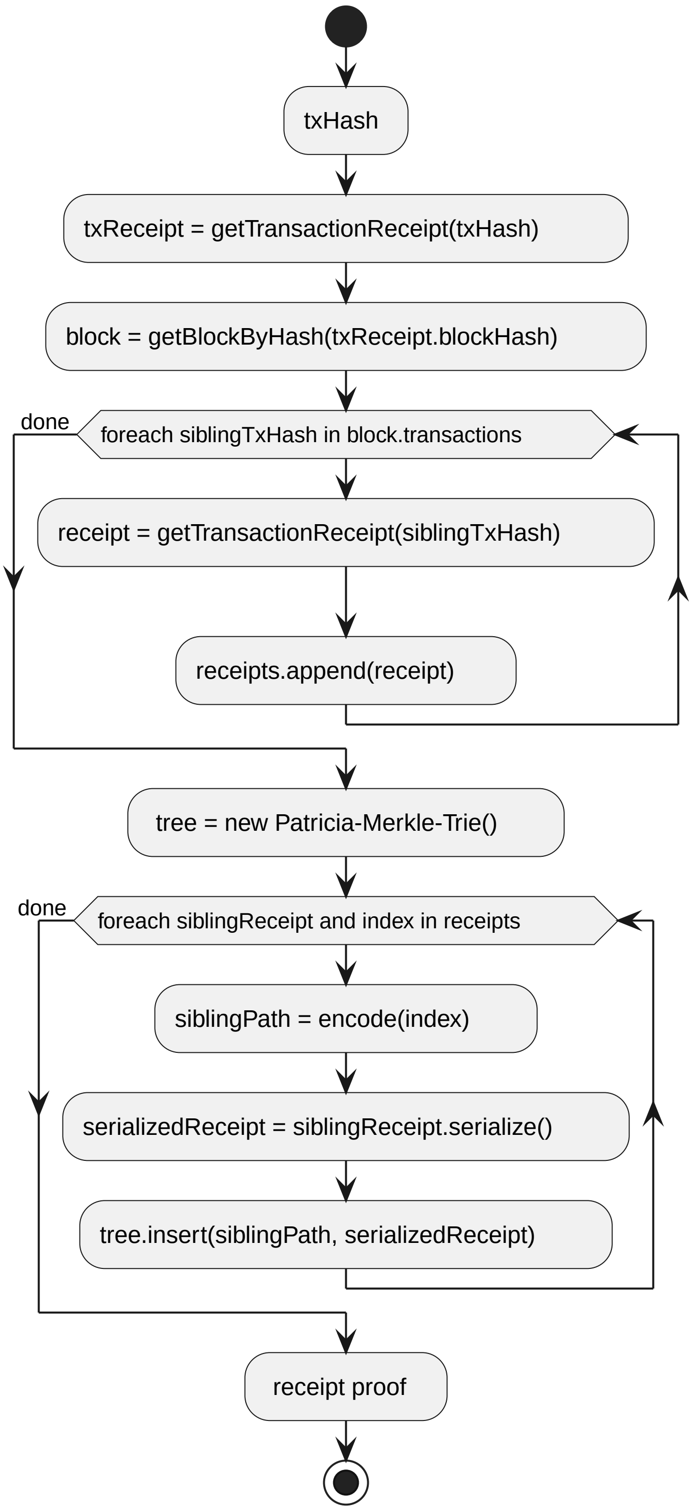

Figure 31. Get Receipt Proof Activity UML Diagram

Chapter 14. Ethereum Proof Submission

Once the Ethereum proof has been created, as explained in the Ethereum Proof Creation chapter, it is submitted to a remote blockchain, which involves the steps illustrated in the Submit Ethereum Proof Activity UML Diagram.

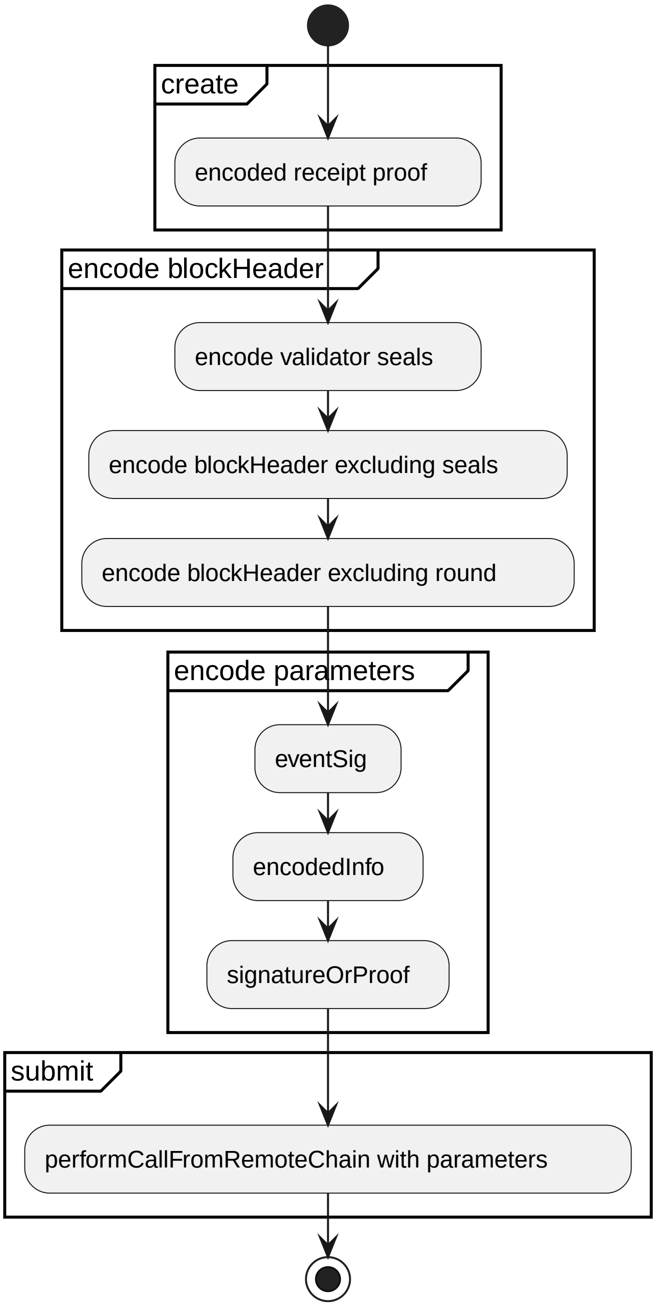

Figure 32. Submit Ethereum Proof Activity UML Diagram

14.1. Ethereum Block Header

An Ethereum block header consists of the following fields:

1. `parentHash`: The Keccak 256-bit hash of the parent block's header
2. `sha3Uncles`: The Keccak 256-bit hash of the ommers list portion of this block

3. `miner`: Miner who mined the block
4. `stateRoot`: The Keccak 256-bit hash of the root node of the state trie
5. `transactionsRoot`: The Keccak 256-bit hash of the root node of the transaction trie
6. `receiptsRoot`: The Keccak 256-bit hash of the root node of the receipts trie
7. `logsBloom`: The Bloom filter composed out of information contained in each log from the receipt of each transaction
8. `difficulty`: A scalar value corresponding to the effort required to mine the block
9. `number`: A scalar value equal to the number of ancestor blocks
10. `gasLimit`: A scalar value equal to the current limit of gas expenditure per block
11. `gasUsed`: A scalar value equal to the total gas used in transactions in this block
12. `timestamp`: A scalar value equal to the time at which the block was mined
13. `extraData`: An arbitrary byte array containing data relevant to this block
14. `mixHash`: A 256-bit hash.
    When combined with the nonce proves that a sufficient amount of work has been carried out on this block.
15. `nonce`: A 256-bit hash.
    When combined with the mixHash proves that a sufficient amount of work has been carried out on this block.

> Blocks could originate from either a QBFT or a IBFT network. It is therefore recommended that the chain identifiers are included during the onboarding process, together with the specific verification and decoding schemes.

14.1.1. QBFT

The extraData field in the block header, defined in QBFT Extra Data, is an RLP encoded structure containing the following information:

1. 32 bytes of vanity data.
2. If using Block header validator selection, a list of validator addresses. Otherwise, if using Contract validator selection, no validators.
3. Any validator votes. No vote is included in the genesis block.
4. The round the block was created on. The round in the genesis block is 0.
5. A list of seals of the validators (signed block hashes). No seals are included in the genesis block.

14.1.2. IBFT

The extraData field in the block header, defined in IBFT 2.0 Extra Data, is an RLP encoded structure containing the following information:

1. 32 bytes of vanity data.
2. A list of validator addresses.

3. Any validator votes. No vote is included in the genesis block.
4. The round the block was created on. The round in the genesis block is 0.
5. A list of seals of the validators (signed block hashes). No seals are included in the genesis block.

14.2. Perform Call From Remote Chain Function

Perform function call from a remote chain. Returns true if remote function call was successfully performed.

requestFollowLeg function signature

```
function performCallFromRemoteChain(
  uint256 blockchainId,
  bytes32 eventSig,
  bytes calldata encodedInfo,
  bytes calldata signatureOrProof
)
```

where:

1. `blockchainId` The source chain identification.
2. `eventSig` The event function signature.
3. `encodedInfo` The combined encoding of the blockchain identifier, the cross-chain control contract's address, the event function signature, and the event data.
4. `signatureOrProof` The information that a validating implementation can use to determine if the event data, given as part of encodedInfo, is valid.

14.3. Event Signature

The `eventSig` is the event function-signature hash, which identifies the event, i.e, the CrossBlockchainCallExecuted event, that was emitted.

eventSig

```
    const eventSig =
web3.utils.soliditySha3('CrossBlockchainCallExecuted(uint256,address,bytes)')
```

14.4. Encoded Information

The `encodedInfo` field is defined as a struct with the following fields:

Struture of the EncodedInfo

```
struct EncodedInfo {
  uint256 blockchainId;
  address contractAddress;
  bytes32 eventSignature;
  bytes   eventData;
}
```

where:

1. `blockchainId` is the blockchain identifier
2. `contractAddress` is the 160-bit Ethereum address of the crosschain control contract
3. `eventSignature` is the event function signature
4. `eventData` is the event data needed to re-construct the Merkle Patricia tree

encodedInfo

```
    let blockchainId = 2
    let crosschainControlContract = '0x0000000000000000000000000000000000000000'
    const encodedInfo = web3.eth.abi.encodeParameters(
      ['uint256', 'address', 'bytes32', 'bytes'],
      [blockchainId, crosschainControlContract, eventSig,
encodedReceiptProof.rlpEncodedReceipt]
    )
```

14.5. Signature or Proof

When an Ethereum proof is submitted to the remote chain, the `signatureOrProof` field is specified by the following struct:

Structure of the SignatureOrProof

```
struct SignatureOrProof {
  bytes rlpSiblingNodes;
  bytes32 receiptsRoot;
  bytes32 blockHash;
  bytes rlpBlockHeader;
  bytes rlpBlockHeaderExcludingRound;
  bytes rlpValidatorSignatures;
}
```

where:

1. `rlpSiblingNodes` is the RLP encoded stack of nodes of the Merkle Patricia tree.
2. `receiptsRoot` is the root of the receipts tree.

3. `blockHash` is the hash of the block.
4. `rlpBlockHeader` is the RLP encoded block header that excludes the validator signatures.
5. `rlpBlockHeaderExcludingRound` is the RLP encoded block header that excludes the round the block was created on (the round in the genesis block is 0) and validator signatures.
6. `rlpValidatorSignatures` is the RLP encoded validator signatures.

signatureOrProof

```
    const rlpSiblingNodes = encodedReceiptProof.witness
    const blockHash = block.hash
    const receiptsRoot = block.receiptsRoot
    const signatureOrProof = web3.eth.abi.encodeParameters(
      ['bytes', 'bytes32', 'bytes32', 'bytes', 'bytes', 'bytes'],
      [rlpSiblingNodes, receiptsRoot, blockHash, rlpBlockHeaderExcludingSeals,
rlpBlockHeaderExcludingRound, rlpValidatorSignatures]
    )
```

The `rlpSiblingNodes` are obtained from the encoded receipt proof’s witness field.

14.5.1. Block Header

The block header indexes are defined according to the ethereum block header structure.

Block Header Field Indexes

```
    const extraDataVanityIndex = 0;
    const extraDataValidatorsIndex = 1;
    const extraDataVoteIndex = 2;
    const extraDataRoundIndex = 3;
    const extraDataSealsIndex = 4;
    const headerParentHashIndex = 0;
    const headerSha3UnclesIndex = 1;
    const headerMinerIndex = 2;
    const headerStateRootIndex = 3;
    const headerTransactionsRootIndex = 4;
    const headerReceiptsRoot = 5;
    const headerLogsBloom = 6;
    const headerDifficulty = 7;
    const headerNumber = 8;
    const headerGasLimit = 9;
    const headerGasUsed = 10;
    const headerTime = 11;
    const headerExtraData = 12;
    const headerMixedHash = 13;
    const headerNonce = 14;
```

The block header is RLP encoded as follows:

Example rlpBlockHeader

```
    let blockHeaderArray = [];
    blockHeaderArray[headerParentHashIndex] = block.parentHash;
    blockHeaderArray[headerSha3UnclesIndex] = block.sha3Uncles;
    blockHeaderArray[headerMinerIndex] = block.miner;
    blockHeaderArray[headerStateRootIndex] = block.stateRoot;
    blockHeaderArray[headerTransactionsRootIndex] = block.transactionsRoot;
    blockHeaderArray[headerReceiptsRoot] = block.receiptsRoot;
    blockHeaderArray[headerLogsBloom] = block.logsBloom;
    blockHeaderArray[headerDifficulty] = block.difficulty === 0 ? '0x' :
web3.utils.toHex(block.difficulty);
    blockHeaderArray[headerNumber] = block.number === 0 ? '0x' :
web3.utils.toHex(block.number);
    blockHeaderArray[headerGasLimit] = block.gasLimit === 0 ? '0x' :
web3.utils.toHex(block.gasLimit);
    blockHeaderArray[headerGasUsed] = block.gasUsed === 0 ? '0x' :
web3.utils.toHex(block.gasUsed);
    blockHeaderArray[headerTime] = block.timestamp === 0 ? '0x' :
web3.utils.toHex(block.timestamp);
    blockHeaderArray[headerExtraData] = block.extraData;
    blockHeaderArray[headerMixedHash] = block.mixHash;
    blockHeaderArray[headerNonce] = block.nonce;
    const rlpBlockHeader = rlp.encode(blockHeaderArray)
```

14.5.2. Block Header with Extra Data Excluding Seals

The `rlpBlockHeader` is an rlp encoded form of the block header where the `extraData` field is replaced by `extraDataExcludingValidatorSeals`, which is a rlp encoding of the block extra data excluding the list of validator seals.

Extra Data Excluding Validator Seals

```
    extraDataExcludingValidatorSeals = '0x' +
rlp.encode([decodedExtraData[extraDataVanityIndex],
decodedExtraData[extraDataValidatorsIndex], decodedExtraData[extraDataVoteIndex],
decodedExtraData[extraDataRoundIndex]]).toString('hex')
```

RLP Encoded Block Header Excluding Validator Seals

```
    let blockHeaderArrayExcludingSeals = [];
    blockHeaderArrayExcludingSeals[headerParentHashIndex] = block.parentHash;
    blockHeaderArrayExcludingSeals[headerSha3UnclesIndex] = block.sha3Uncles;
    blockHeaderArrayExcludingSeals[headerMinerIndex] = block.miner;
    blockHeaderArrayExcludingSeals[headerStateRootIndex] = block.stateRoot;
    blockHeaderArrayExcludingSeals[headerTransactionsRootIndex] =
block.transactionsRoot;
    blockHeaderArrayExcludingSeals[headerReceiptsRoot] = block.receiptsRoot;
    blockHeaderArrayExcludingSeals[headerLogsBloom] = block.logsBloom;
    blockHeaderArrayExcludingSeals[headerDifficulty] = block.difficulty === 0 ? '0x' :
web3.utils.toHex(block.difficulty);
    blockHeaderArrayExcludingSeals[headerNumber] = block.number === 0 ? '0x' :
web3.utils.toHex(block.number);
    blockHeaderArrayExcludingSeals[headerGasLimit] = block.gasLimit === 0 ? '0x' :
web3.utils.toHex(block.gasLimit);
    blockHeaderArrayExcludingSeals[headerGasUsed] = block.gasUsed === 0 ? '0x' :
web3.utils.toHex(block.gasUsed);
    blockHeaderArrayExcludingSeals[headerTime] = block.timestamp === 0 ? '0x' :
web3.utils.toHex(block.timestamp);
    blockHeaderArrayExcludingSeals[headerExtraData] =
extraDataExcludingValidatorSeals;
    blockHeaderArrayExcludingSeals[headerMixedHash] = block.mixHash;
    blockHeaderArrayExcludingSeals[headerNonce] = block.nonce;
    const rlpBlockHeaderExcludingSeals = rlp.encode(blockHeaderArrayExcludingSeals)
```

14.5.3. Block Header with Extra Data Excluding Seals and Round

The `rlpBlockHeaderExcludingRound` is another rlp encoded form of the block header where the `extraData` field is replaced by `extraDataExcludingRoundAndValidatorSeals`, which is a rlp encoding of the block extra data excluding the list of validator seals and the round the block was created on.

Extra Data Excluding Round and Validator Seals

```
    let consensus = 'qbft'
    let extraDataExcludingRoundAndValidatorSeals
    if (consensus === 'ibft') {
      extraDataExcludingRoundAndValidatorSeals = '0x' +
rlp.encode([decodedExtraData[extraDataVanityIndex],
decodedExtraData[extraDataValidatorsIndex],
decodedExtraData[extraDataVoteIndex]]).toString('hex')
    } else if (consensus === 'qbft') {
      extraDataExcludingRoundAndValidatorSeals = '0x' +
rlp.encode([decodedExtraData[extraDataVanityIndex],
decodedExtraData[extraDataValidatorsIndex], decodedExtraData[extraDataVoteIndex],
decodedExtraData[extraDataRoundIndex], []]).toString('hex')
    }
```

RLP Encoded Block Header Excluding Round and Validator Seals

```
    let blockHeaderArrayExcludingRoundAndValidatorSeals = [];
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerParentHashIndex] =
block.parentHash;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerSha3UnclesIndex] =
block.sha3Uncles;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerMinerIndex] = block.miner;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerStateRootIndex] =
block.stateRoot;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerTransactionsRootIndex] =
block.transactionsRoot;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerReceiptsRoot] =
block.receiptsRoot;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerLogsBloom] =
block.logsBloom;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerDifficulty] =
block.difficulty === 0 ? '0x' : web3.utils.toHex(block.difficulty);
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerNumber] = block.number === 0
? '0x' : web3.utils.toHex(block.number);
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerGasLimit] = block.gasLimit
=== 0 ? '0x' : web3.utils.toHex(block.gasLimit);
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerGasUsed] = block.gasUsed ===
0 ? '0x' : web3.utils.toHex(block.gasUsed);
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerTime] = block.timestamp ===
0 ? '0x' : web3.utils.toHex(block.timestamp);
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerExtraData] =
extraDataExcludingRoundAndValidatorSeals;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerMixedHash] = block.mixHash;
    blockHeaderArrayExcludingRoundAndValidatorSeals[headerNonce] = block.nonce;
    const rlpBlockHeaderExcludingRound =
rlp.encode(blockHeaderArrayExcludingRoundAndValidatorSeals)
```

14.5.4. Extra Data Validator Seals

The validator seals are contained in the block's extra data, defined in QBFT Extra Data and IBFT Extra Data, which are extracted and then rlp encoded in order to obtain the `rlpValidatorSignatures`.

Encoded Extra Data Validator Seals

```
    let decodedExtraData = rlp.decode(block.extraData)
    const decodedValidatorSeals = decodedExtraData[extraDataSealsIndex]
    const decodedValidatorSealArray = []
    for (let vs of decodedValidatorSeals) {
      decodedValidatorSealArray.push(vs)
    }
    rlpValidatorSignatures = '0x' +
rlp.encode(decodedValidatorSealArray).toString('hex')
```

Chapter 15. Ethereum Proof Verification

A proof will be received from the Ethereum chain that must verify the correctness of at least the following information:

- The transaction details included in the event
- Payer bank id
- Beneficiary bank id
- Amount
- Timestamp
- Trade id
- Chain id
- The Ethereum transaction receipt
- Block number
- Transaction hash
- The transaction event log details (as stated in the transaction details)
- The block was signed by the required validators

This required proof can be provided as a tree data structure, in which parent nodes are constructed by hashing the concatenation of their children's node data.
The tree data structure used by Ethereum is called the Merkle Patricia tree.
Since the joining of nodes depends on the values of connected nodes, any changes in a node's data would result in a change of the tree's root.
Furthermore, a comparison of the roots from an existing tree to that of a reconstructed tree, will only match if the exact same nodes were used in the reconstructed tree.
Using the above mechanism, it is therefore possible to verify a node's data membership in the structure by reconstructing the merkle patricia tree.
Such a reconstruction is possible without needing to use all the leaf nodes.
Rather, it only requires:

1. The leaf node needed to be proven, and it's sibling
2. The siblings of the parent nodes, up until the root node is reached

An ethereum block header contains 3 roots from 3 trees:

- `transactionsRoot`
- `receiptsRoot`
- `stateRoot`

These are the roots of Merkle Patricia Trees contained in Ethereum blocks, as shown in the Ethereum Blockchain Trees diagram.

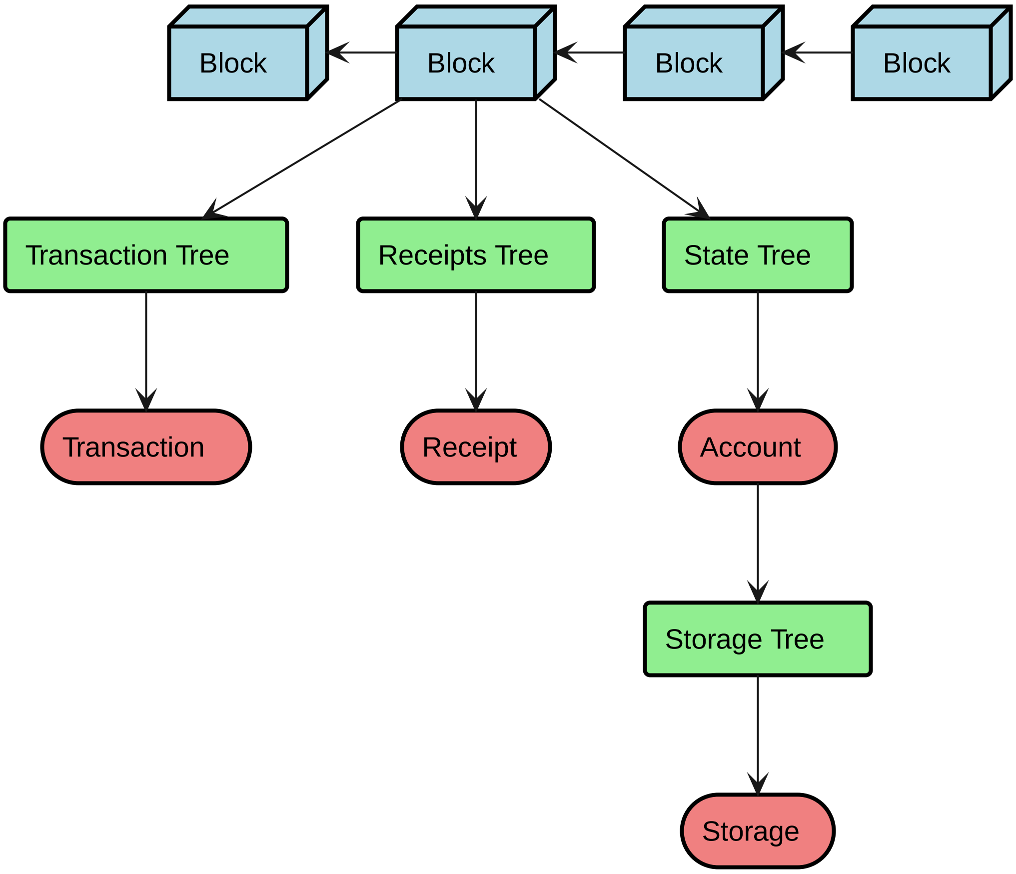

Figure 33. Ethereum Blockchain Trees

Provided with a trusted Ethereum block header, which contains the block's receiptsRoot it is possible to prove exactly what transaction receipts (which includes event logs) are included in the Ethereum blockchain.

15.1. Verification Steps

15.1.1. Foreign System Integration

The onboarding process involves configuring information from the source blockchain needed to verify, for example, the transaction details, block headers and Merkle Patricia tree proofs for the transaction receipt.

The following information can be onboarded:

- A mapping of blockchain Ids and foreign to local account Ids
- The mapping of the current block's hash to the `receiptsRoot`
- The list of current block's validators
- Notaries and Participants
- Proving Schemes

Details of integrating with a foreign system are provided in the Foreign System Integration chapter.

15.1.2. Block Header Verification

The block hash, which subsequent ethereum blocks will include in their block headers as parent block hash, differs from the block hash that the validators sign.
The block hash that is signed includes the round the block was created on (the round in the genesis block is 0), whereas the block hash used in subsequent child blocks does not.

To simplify the on-chain block header verification process, a block containing the transaction of interest is retrieved from the source blockchain from which the following two block headers are constructed:

- `rlpBlockHeader`: The rlp encoded block header that excludes the validator signatures
- `rlpBlockHeaderExcludingRound`: The rlp encoded block header that excludes the round number and validator signatures

The on-chain block header verification process will then first check that the data from these two constructed block headers are equal.

> The two block headers are constructed off-chain by manipulating the original block header and excluding the round number and validator signatures.
> An on-chain comparison of the block headers, with round number and without, ensures that they are equal and can be trusted.
> The block hash that excludes the round number is the block hash used to lookup the transaction `receiptsRoots` via a Besu client.
> Hence, if a mapping of the current block's hash to `receiptsRoot` is stored during the onboarding process, it is this hash that should be mapped to the transaction `receiptsRoots`.

Next, the validator signatures that were separated from the block header are verified by recovering the signature addresses from the hash of the `rlpBlockHeader` and checking that the signature addresses are contained in the array of validators stored during the onboarding process.

> Each block contains a full list of validators, encoded in the extra data field.
> If there are any changes to the validators it would be reflected in the list.
> If enough of the previously stored validator signatures could be verified against the block that contains the updated list, then it's a safe option to update the currently stored list of validators.
> It's possible to query by block hash to get the validator set for that block, hence for a single chain, or get all the validators for all the Ethereum chains that have been configured.

Finally, the block hash is compared to the hash of the `rlpBlockHeaderExcludingRound`.

Details of the block header verification are provided in the
Block Header Verification chapter.

15.1.3. Creating the Merkle Patricia Proof

The events emitted from the blockchain are monitored until a `CrossBlockchainCallExecuted` event is picked up from the transaction of interest.
Then the transaction receipt of the transaction of interest is extracted and used to create the Merkle Patricia proof.

This proof is created off-chain using the transaction receipt and consists of the following information:

- The path in the tree leading to value
- The terminating value in the tree (the transaction receipt containing the event to be proven)
- The rlp encoded stack of parent nodes making up the tree

An example of the structure of the proof is provided in the Ethereum Proof Generation section in the appendix.

15.1.4. Verifying the Merkle Patricia Proof

To verify the Merkle patricia proof, the event of interest must be shown to be contained in the terminating value of the Merkle patricia tree.

The following data is required in order to verify the event:

- The terminating bytes value in the tree
- The bytes corresponding to the rlp encoded stack of parent nodes in the tree
- The bytes32 root hash of the tree
- The bytes32 hash of the block

Starting at the leaf node containing the event of interest, the parent nodes are looped through, each time checking that the hash of child node is present in the parent node.
This process is repeated until the hash of parent equals the receiptsRoot, successfully verifying the proof and proving the event is contained in the receiptsRoot.

For example, the Merkle Tree diagram shows the iterative process of walking the tree from the leaf value `C` to its parent nodes, each time checking that the parent includes the hash of its children, in order to finally verify the inclusion of `C` in the tree's root node.
The solid arrows indicate the path that must be taken, while the dashed lines indicate the parts of the tree which are not required for proving the inclusion of `C`.

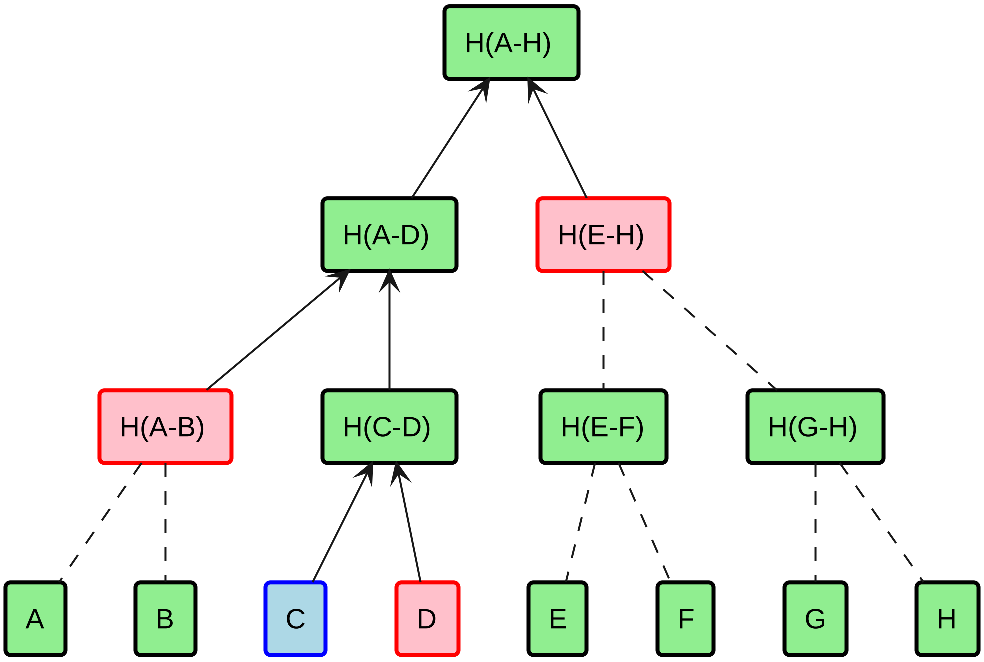

Figure 34. Merkle Tree

Details of the proof verification are provided in the Merkle Patricia Proof Verification chapter.

Chapter 16. Block Header Verification

An Ethereum block header consists of the following fields:

1. `parentHash`: The Keccak 256-bit hash of the parent block's header
2. `sha3Uncles`: The Keccak 256-bit hash of the ommers list portion of this block
3. `miner`: Miner who mined the block
4. `stateRoot`: The Keccak 256-bit hash of the root node of the state trie
5. `transactionsRoot`: The Keccak 256-bit hash of the root node of the transaction trie
6. `receiptsRoot`: The Keccak 256-bit hash of the root node of the receipts trie
7. `logsBloom`: The Bloom filter composed out of information contained in each log from the receipt of each transaction
8. `difficulty`: A scalar value corresponding to the effort required to mine the block
9. `number`: A scalar value equal to the number of ancestor blocks
10. `gasLimit`: A scalar value equal to the current limit of gas expenditure per block
11. `gasUsed`: A scalar value equal to the total gas used in transactions in this block
12. `timestamp`: A scalar value equal to the time at which the block was mined
13. `extraData`: An arbitrary byte array containing data relevant to this block
14. `mixHash`: A 256-bit hash.
    When combined with the nonce proves that a sufficient amount of work has been carried out on this block.
15. `nonce`: A 256-bit hash.
    When combined with the mixHash proves that a sufficient amount of work has been carried out on this block.

In addition, the following properties will be set equal values specific to IBFT 2.0 private networks by the IBFT 2.0 Genesis file:

- `nonce`: 0x0
- `difficulty`: 0x1
- `mixHash`: 0x63746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365 for Istanbul block identification.

The following provides an example of an Ethereum block:

Example of an Ethereum block

```
block {
  number: 102319,
  hash: '0x465bf127e9fa87e20725c506648fbdc1a1fad383894cae7a35b033484bf80122',
  mixHash: '0x63746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365',
  parentHash: '0xb90b639bc9c21e43f249da653386198f52d6504bb269cd67ae4022da2147a24c',
  nonce: '0x0000000000000000',
  sha3Uncles: '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
  logsBloom:
'0x01000000000000000000000000000000000000040001000000000000000000000000000000000000000000
00000000080020000000000000000000000000000000800002000000000000000000000000000000000000
00000000000000000000000004000000000000000000000000000000000000000800000000000000000008
00000000000000000000010000000000000000000000008000000000000000000000000000000000000000
00000000004002000000000000000000000000000000000000000000000000000000000000000000004200
00010000000000000000000008000000000000000000000000000000000010000000000002000000000000'
,
  transactionsRoot:
'0xc00880fbf3eb0cb7ffd44d7d0a01ca7b3ffbf9ed0850260f45d9cbb554982272',
  stateRoot: '0xb6ff072903ffe8b7a17c6e0bfd587e3fc1379303c2bf493854e5663b74611343',
  receiptsRoot: '0x57979dc5e69036b8c42db40a957447f779a72fd8331063873e6d0a8fd1a7bd9d',
  miner: '0xCA31306798B41BC81C43094a1E0462890Ce7a673',
  difficulty: '1',
  totalDifficulty: '102320',
  extraData:
'0xf882a000000000000000000000000000000000000000000000000000000000000000000d594ca3130679
8b41bc81c43094a1e0462890ce7a673808400000000f843b841bdc25e2a8baa8e266503d794225a240422c
b236fc2844b286022812c6d7535de616c4498437571a76acdc996169460875d66fbdad88087a284ddbaa0a
87d70a401',
  size: 4408,
  gasLimit: 100000000,
  gasUsed: 976914,
  timestamp: 1661435546,
  uncles: [],
  transactions: [
    '0x1a852331554419741d8b9612ce533c1b7b80da093fb86f810058a7a7e138089d'
  ]
}
```

16.1. Verifying the BFT Block Header

The process of verifying the block header requires the data contained in the `signatureOrProof` field of the proof.
The `signatureOrProof` field is specified by the following struct:

Structure of the SignatureOrProof

```
struct SignatureOrProof {
  bytes rlpSiblingNodes;
  bytes32 receiptsRoot;
  bytes32 blockHash;
  bytes rlpBlockHeader;
  bytes rlpBlockHeaderExcludingRound;
  bytes rlpValidatorSignatures;
}
```

where:

1. `rlpSiblingNodes` is the RLP encoded stack of nodes of the Merkle Patricia tree
2. `receiptsRoot` is the root of the receipts tree
3. `blockHash` is the hash of the block
4. `rlpBlockHeader` is the RLP encoded block header that excludes the validator signatures
5. `rlpBlockHeaderExcludingRound` is the RLP encoded block header that excludes the round the block was created on (the round in the genesis block is 0) and validator signatures
6. `rlpValidatorSignatures` is the RLP encoded validator signatures

The `rlpBlockHeader`, `rlpBlockHeaderExcludingRound`, and `rlpValidatorSignatures` can be extracted from the `signatureOrProof` as follows:

Example signatureOrProof

```
    let signatureOrProof =
'0x00000000000000000000000000000000000000000000000000000000000000c075000e4e7a45b0c26ba
d2ea3bd276f6f4fbfec4ef5751ec33859d12d3897eecc09d5df28dd921563105e8e21dc7b4ad825d644b21
8b41982b11f262806d623fa000000000000000000000000000000000000000000000000000000000000076
000000000000000000000000000000000000000000000000000000000000009c0000000000000000000000
0000000000000000000000000000000000000000c200000000000000000000000000000000000000000000
00000000000000000066af90667f90664822080b9065ef9065b01830fa6f7b901000000000000000000000
00000000000000008004000000000000000200004000000000000000200000080000000000002000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00010000000000000000000000000000000000000008000000000000000000000000004000000000000001
40000000000000004000000080000000000000020000000000000000000000000000000000400000000000
00000000000000000000008000000100000000000000000000000000000040000000000000000000000000
0000000000000000000000000000000000000010000000000002000000000000f90550f9021a94f1705d25c
81fb6eab1646a08c1eac04e1682c065e1a04a088333602fc5fc4f1782002fd83cb872f641d7564993fcae5
a70619d371b83b901e00000000000000000000000000000000000000000000000000000000000000100000
00000000000000000000000000000000000000000000000000000014000000000000000000000000000000
00000000000000000000000000000000000018000000000000000000000000000000000000000000000000
00000000000000001686f6c64457865637574696f6e0000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000632060fb000000000000000000000000000
00000000000000000000000000000000001c00000000000000000000000000000000000000000000000000
000000001348bf10000000000000000000000000000000000000000000000000000000000000013100000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

0000000000000000000000000000000000e757365724163636f756e7449643200000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000e757365724
163636f756e744964310000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000f89994f1705d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e
25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b860000000000000000000000
00000000000000000000000000000000000000000200000000000000000000000000000000000000000000
00000000000000000001310000000000000000000000000000000000000000000000000000000000000000f
8799487780452339d47a5d86c1a8e77eadb2c46c0ac3ae1a0e86cf26454ff6ddddafa95eaf2baab1df2c06
2c4341cec61ccfafae50eabf8d3b840000000000000000000000000002afcc963a8c7119b700063e17a2d468
46ba8b50e0000000000000000000000000000000000000000000000000000000000000000f9021a94d1043
43c69174447b34981c2c3e536a66c159c48e1a07a752c4d100a96be23f60f072fed1f33da2890fbe45eb15
af52750faa2772a92b901e0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000fd2898e8df49fb8de957a6b75bcd4ba55eb550f8000000000000000000000
00000000000000000000000000000000000000000600000000000000000000000000000000000000000000
0000000000000000001448903901f000000000000000000000000000000000000000000000000000000000
000008000000000000000000000000000000000000000000000000000000000000000c0000000000000000
00000000000000000000000000000000000000000000001000000000000000000000000000000000000000
00000000000000000000000000100000000000000000000000000000000000000000000000000000000000
00001310000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000e75736572416363 6f756e744964310000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00e757365724163636f756e744964320000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000023ef9023ba0e333a31e86179fb3056
49b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8dec75d7aab85b567b6ccd41ad312451
b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e0462890ce7a673a09850cf8b19f4707e4
f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018028745195ce85510ec4e51adecff39
41207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751ec33859d12d3
897eeccb9010000000000000000000000000000000000000800400000000000000002000040000000000000
00200000080000000000002000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000100000000000000000000000000000000000000080000000000
00000000000000004000000000000001400000000000000040000000800000000000000200000000000000
00000000000000000000400000000000000000000000000000000080000001000000000000000000000000
00000040000000000000000000000000000000000000000000000000000000000000000100000000000200
000000000018301ae8e8405f5e100830fa6f784632060fbb83ff83da000000000000000000000000000000
00000000000000000000000000000000000d594ca31306798b41bc81c43094a1e0462890ce7a6738084000
00000a063746963616c2062797a616e74696e65206661756c7420746f6c6572616e63658800000000000000
00000000000000000000000000000000000000000000000000000000000000000000239f90236a0e333a31
e86179fb305649b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8dec75d7aab85b567b6c
cd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e0462890ce7a673a09850c
f8b19f4707e4f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018028745195ce85510ec
4e51adecff3941207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751
ec33859d12d3897eeccb9010000000000000000000000000000000000000800400000000000000002000040
00000000000000200000080000000000002000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000100000000000000000000000000000000000000
08000000000000000000000000004000000000000001400000000000000040000000800000000000000200
00000000000000000000000000000000400000000000000000000000000000000080000001000000000000
00000000000000000040000000000000000000000000000000000000000000000000000000000000001000
000000000200000000000018301ae8e8405f5e100830fa6f784632060fbb83af838a000000000000000000
00000000000000000000000000000000000000000000000d594ca31306798b41bc81c43094a1e0462890ce
7a67380a063746963616c2062797a616e74696e65206661756c7420746f6c6572616e636588000000000000

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000045f84
3b84177d6585b25b5b165d65925decc443dbf374941b36a71c0e5ee4bb7e65f3185fc2eaea091c9ac4b8fa
f09fa6d6d21a18d7d2905667f381ee096ae8103235bc8b3010000000000000000000000000000000000000
0000000000000000'
```

Extracting data from the signatureOrProof

```
    const BLOCK_HEADER_INDEX = 3
    const BLOCK_HEADER_EXCLUDING_ROUND_INDEX = 4
    const VALIDATOR_SIGNATURES_INDEX = 5
    let abiCoder = new ethers.utils.AbiCoder
    let decodedSignatureOrProof = abiCoder.decode(['bytes', 'bytes32', 'bytes32',
'bytes', 'bytes', 'bytes'], signatureOrProof);
    let rlpBlockHeader = decodedSignatureOrProof[BLOCK_HEADER_INDEX]
    let rlpBlockHeaderExcludingRound =
decodedSignatureOrProof[BLOCK_HEADER_EXCLUDING_ROUND_INDEX]
    let rlpValidatorSignatures = decodedSignatureOrProof[VALIDATOR_SIGNATURES_INDEX]
    console.log({rlpBlockHeader, rlpBlockHeaderExcludingRound,
rlpValidatorSignatures})
```

Example output

```
    {
      rlpBlockHeader:
'0xf9023ba0e333a31e86179fb305649b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8d
ec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e046
2890ce7a673a09850cf8b19f4707e4f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018
028745195ce85510ec4e51adecff3941207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd2
76f6f4fbfec4ef5751ec33859d12d3897eeccb901000000000000000000000000000000000000080040000
00000000000200004000000000200000000000008000000000002000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000100000000000000000000
00000000000000000008000000000000000000000000004000000000000001400000000000000040000000
80000000000000020000000000000000000000000000000000400000000000000000000000000000000080
00000100000000000000000000000000000040000000000000000000000000000000000000000000000000
00000000000000100000000000020000000000018301ae8e8405f5e100830fa6f784632060fbb83ff83da
00000000000000000000000000000000000000000000000000000000000000000d594ca31306798b41bc81
c43094a1e0462890ce7a673808400000000a063746963616c2062797a616e74696e65206661756c7420746
f6c6572616e636588000000000000000000',
      rlpBlockHeaderExcludingRound:
'0xf90236a0e333a31e86179fb305649b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8d
ec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e046
2890ce7a673a09850cf8b19f4707e4f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018
028745195ce85510ec4e51adecff3941207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd2
76f6f4fbfec4ef5751ec33859d12d3897eeccb901000000000000000000000000000000000000080040000
00000000000200004000000000200000000000008000000000002000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000100000000000000000000
00000000000000000008000000000000000000000000004000000000000001400000000000000040000000
80000000000000020000000000000000000000000000000000400000000000000000000000000000000080
00000100000000000000000000000000000040000000000000000000000000000000000000000000000000
00000000000000100000000000020000000000018301ae8e8405f5e100830fa6f784632060fbb83af838a
00000000000000000000000000000000000000000000000000000000000000000d594ca31306798b41bc81
c43094a1e0462890ce7a67380a063746963616c2062797a616e74696e65206661756c7420746f6c6572616
e636588000000000000000000',
      rlpValidatorSignatures:
'0xf843b84177d6585b25b5b165d65925decc443dbf374941b36a71c0e5ee4bb7e65f3185fc2eaea091c9a
c4b8faf09fa6d6d21a18d7d2905667f381ee096ae8103235bc8b301'
    }
```

16.1.1. Comparing Headers

A comparison must be made between the `rlpBlockHeader` and `rlpBlockHeaderExcludingRound` to verify that the header fields contain the same `parentHash`, `sha3Uncles`, `miner`, `transactionsRoot`, `receiptsRoot`, `logsBloom`, `difficulty`, `number`, `gasLimit`, `gasUsed`, `timestamp`, `mixHash`, `nonce`.

Additionally, in order to check that the validator addresses portion of the headers are equal, the extra data must be extracted, which contains the list of validator addresses.

The extraData field in the block header, defined in IBFT 2.0 Extra Data, is an RLP encoded structure containing the following information:

1. 32 bytes of vanity data.
2. A list of validator addresses.
3. Any validator votes.
   No vote is included in the genesis block.
4. The round the block was created on.
   The round in the genesis block is 0.
5. A list of seals of the validators (signed block hashes).
   No seals are included in the genesis block.

> The blocks could also potentially originate from a QBFT network, in which the extra data fields might differ.
> It is therefore recommended that the chain identifiers are included during the onboarding process, together with the specific verification and decoding schemes.

The `rlpBlockHeader` is the block header with the `extraData` field replaced by `extraDataExcludingValidatorSeals`, which is a rlp encoding of the block extra data excluding the list of validator seals.

The following provides an example of how the `extraDataExcludingValidatorSeals` would be calculated from the `extraData`:

Example extraData

```
    let extraData =
'0xf882a00000000000000000000000000000000000000000000000000000000000000000d594ca3130679
8b41bc81c43094a1e0462890ce7a673808400000000f843b84177d6585b25b5b165d65925decc443dbf374
941b36a71c0e5ee4bb7e65f3185fc2eaea091c9ac4b8faf09fa6d6d21a18d7d2905667f381ee096ae81032
35bc8b301'
```

Calculating extraDataExcludingValidatorSeals

```
    const VANITY_DATA_INDEX = 0
    const VALIDATOR_ADDRESSES_INDEX = 1
    const VALIDATOR_VOTES = 2
    const ROUND_INDEX = 3
    const VALIDATOR_SEALS = 4
    let decodedExtraData = ethers.utils.RLP.decode(extraData)
    let extraDataExcludingValidatorSeals = ethers.utils.RLP.encode(
    [decodedExtraData[VANITY_DATA_INDEX], decodedExtraData[VALIDATOR_ADDRESSES_INDEX],
decodedExtraData[VALIDATOR_VOTES], decodedExtraData[ROUND_INDEX]])
    console.log({extraDataExcludingValidatorSeals})
```

Example output

```
{
  extraDataExcludingValidatorSeals:
'0xf83da0000000000000000000000000000000000000000000000000000000000000000d594ca3130679
8b41bc81c43094a1e0462890ce7a673808400000000'
}
```

The `rlpBlockHeaderExcludingRound` is the block header with the `extraData` field replaced by `extraDataExcludingRoundAndValidatorSeals`, which is a rlp encoding of the block extra data excluding the list of validator seals and the round the block was created on.
It would be calculated as follows:

Calculating extraDataExcludingRoundAndValidatorSeals

```
let extraDataExcludingRoundAndValidatorSeals = ethers.utils.RLP.encode(
[decodedExtraData[VANITY_DATA_INDEX], decodedExtraData[VALIDATOR_ADDRESSES_INDEX],
decodedExtraData[VALIDATOR_VOTES]])
console.log({extraDataExcludingRoundAndValidatorSeals})
```

Example output

```
{
  extraDataExcludingRoundAndValidatorSeals:
'0xf838a0000000000000000000000000000000000000000000000000000000000000000d594ca3130679
8b41bc81c43094a1e0462890ce7a67380'
}
```

Therefore, the extra data in the two headers can be extracted and the validator signatures compared as follows:

Viewing block headers for comparison

```
const EXTRA_DATA_INDEX = 12;
let header = ethers.utils.RLP.decode(rlpBlockHeader);
let headerNoRoundNumber = ethers.utils.RLP.decode(rlpBlockHeaderExcludingRound);
let rlpExtraDataHeader1 = header[EXTRA_DATA_INDEX]
console.log({rlpExtraDataHeader1})
let decodedRlpExtraDataHeader1 = ethers.utils.RLP.decode(rlpExtraDataHeader1)
let rlpExtraDataHeader2 = headerNoRoundNumber[EXTRA_DATA_INDEX]
console.log({rlpExtraDataHeader2})
let decodedRlpExtraDataHeader2 = ethers.utils.RLP.decode(rlpExtraDataHeader2)
console.log("Decoded rlpExtraDataHeader1: ", decodedRlpExtraDataHeader1)
console.log("Decoded rlpExtraDataHeader2: ", decodedRlpExtraDataHeader2)
```

Example output

```
    {
      rlpExtraDataHeader1:
'0xf83da00000000000000000000000000000000000000000000000000000000000000000d594ca3130679
8b41bc81c43094a1e0462890ce7a673808400000000'
    }
    {
      rlpExtraDataHeader2:
'0xf838a00000000000000000000000000000000000000000000000000000000000000000d594ca3130679
8b41bc81c43094a1e0462890ce7a67380'
    }
    Decoded rlpExtraDataHeader1:  [
      '0x0000000000000000000000000000000000000000000000000000000000000000',
      [ '0xca31306798b41bc81c43094a1e0462890ce7a673' ],
      '0x',
      '0x00000000'
    ]
    Decoded rlpExtraDataHeader2:  [
      '0x0000000000000000000000000000000000000000000000000000000000000000',
      [ '0xca31306798b41bc81c43094a1e0462890ce7a673' ],
      '0x'
    ]
```

16.1.2. Verifying the Validator Signatures

The `chainHeadValidators` must be requested from the Ethereum blockchain and then be verified against the validator signatures.
Seals of the validators are signed block hashes.

Verifying chainHeadValidators

```
    const chainHeadValidators = [ '0xca31306798b41bc81c43094a1e0462890ce7a673' ]
    let signedHash = ethers.utils.keccak256(rlpBlockHeader)
    console.log("signedHash: ", signedHash)
    let validatorSignatures = ethers.utils.RLP.decode(rlpValidatorSignatures)
    console.log("validatorSignatures: ", validatorSignatures)
    let validSeals = 0;
    let addressReuseCheck = []
    for (let i = 0; i < validatorSignatures.length; i++) {
        res = ethJsUtil.fromRpcSig(validatorSignatures[i])
        pub = ethJsUtil.ecrecover(ethJsUtil.toBuffer(signedHash), res.v, res.r,
res.s);
        addrBuf = ethJsUtil.pubToAddress(pub);
        signatureAddress = ethJsUtil.bufferToHex(addrBuf);
        console.log("signatureAddress: ", signatureAddress)
        for (let j = 0; j < chainHeadValidators.length; j++) {
            if (signatureAddress == chainHeadValidators[j]) {
                for (let k = 0; k < i; k++) {
                    if (addressReuseCheck[k] == signatureAddress) {
                        console.log("Error: Not allowed to submit multiple seals from
the same validator")
                        break;
                    }
                }
                validSeals = validSeals + 1;
                addressReuseCheck[i] = signatureAddress;
                break;
            }
        }
    }
    if (validSeals < chainHeadValidators.length / 2) {
        console.log("Error: Not enough valid validator seals");
    }
```

Example output

```
    signedHash:  0x91f6630fe2fea7836171b5cc343387a64020f7fa2754e2256c7274d32f8e3afa
    validatorSignatures:  [

'0x77d6585b25b5b165d65925decc443dbf374941b36a71c0e5ee4bb7e65f3185fc2eaea091c9ac4b8faf0
9fa6d6d21a18d7d2905667f381ee096ae8103235bc8b301'
    ]
    signatureAddress:  0xca31306798b41bc81c43094a1e0462890ce7a673
```

16.1.3. Verifying the Calculated Block Hash

The block hash can be calculated by hashing the `rlpBlockHeaderExcludingRound` parameter. This step checks that the calculated value equals the blockhash obtained from the `signatureOrProof`.

Extracting and comparing the calculated block hash

```
const BLOCK_HASH_INDEX = 2
let blockHash = decodedSignatureOrProof[BLOCK_HASH_INDEX]
let calculatedBlockHash = ethers.utils.keccak256(rlpBlockHeaderExcludingRound)
console.log(calculatedBlockHash)
console.log(blockHash)
```

Example output

```
0x09d5df28dd921563105e8e21dc7b4ad825d644b218b41982b11f262806d623fa
0x09d5df28dd921563105e8e21dc7b4ad825d644b218b41982b11f262806d623fa
```

Chapter 17. Merkle Patricia Proof Verification

The following describes how the proof generated by Ethereum is decoded in order to verify that a transaction of interest occurred on the Ethereum chain.
The code examples provided make use of the Ethers npm module.

The proof generated by Ethereum contains the following fields:

1. `tradeId` is a string identifying the trade
2. `event` is a string which can either be completeLeadLeg or cancelLeadLeg
3. `proof` is the struct consisting of the following:
4. `blockchainId` is the uint256 identifying the buying bank's ledger
5. `eventSig` is the EventSig struct
6. `encodedInfo` is the EncodedInfo Struct
7. `signatureOrProof` is the Merkle Patrica tree proof

An example of an Ethereum proof is as follows:

Structure of an Ethereum proof

```
{
  "tradeId": "1",
  "event": "completeLeadLeg",
  "proof": {
    "blockchainId": 1,
    "eventSig": "0x7a752c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92",
    "encodedInfo":
"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000007a752c4d100a96be23f60f072fed1f33da2890fbe
45eb15af52750faa2772a9200000000000000000000000000000000000000000000000000000000000008
00000000000000000000000000000000000000000000000000000000000065ef9065b01830fa6f7b9010
0000000000000000000000000000000000008004000000000000000200040000000000000002000000800
00000000020000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000100000000000000000000000000000000000000080000000000000000000000
00040000000000000140000000000000000000400000008000000000000000020000000000000000000000000
0000000040000000000000000000000000000000008000000100000000000000000000000000000400000
0000000000000000000000000000000000000000000000000000000001000000000000200000000000f90
550f9021a94f1705d25c81fb6eab1646a08c1eac04e1682c065e1a04a088333602fc5fc4f1782002fd83cb
872f641d7564993fcae5a70619d371b83b901e000000000000000000000000000000000000000000000000
00000000000001000000000000000000000000000000000000000000000000000000000000000014000000
00000000000000000000000000000000000000000000000000000000180000000000000000000000000000
0000000000000000000000000000000000001686f6c64457865637574696f6e00000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000632060fb0000000
0000000000000000000000000000000000000000000000000000001c000000000000000000000000000000
00000000000000000000000000001348bf1000000000000000000000000000000000000000000000000000
00000000000013100000000000000000000000000000000000000000000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000e757365724163636f756e74496432000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000e757365724163636f756e74496431000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000f89994f1705d25c81fb6eab1646a08c1e
ac04e1682c065e1a0f0e25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b8600
0000000000000000000000000000000000000000000000000000000000000200000000000000000000000
000000000000000000000000000000000131000000000000000000000000000000000000000000000000000
0000000000000000000f8799487780452339d47a5d86c1a8e77eadb2c46c0ac3ae1a0e86cf26454ff6dddd
afa95eaf2baab1df2c062c4341cec61ccfafae50eabf8d3b8400000000000000000000000002afcc963a8c
7119b700063e17a2d46846ba8b50e000000000000000000000000000000000000000000000000000000000
0000000f9021a94d104343c69174447b34981c2c3e536a66c159c48e1a07a752c4d100a96be23f60f072fe
d1f33da2890fbe45eb15af52750faa2772a92b901e00000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000fd2898e8df49fb8de957a6b75bcd4ba55eb550f80
00000000000000000000000000000000000000000000000000000000000060000000000000000000000000
00000000000000000000000000000000000000001448903901f00000000000000000000000000000000000
00000000000000000000000080000000000000000000000000000000000000000000000000000000000000
000c0000000000000000000000000000000000000000000000000000000000000010000000000000000000
00000000000000000000000000000000000000000000001000000000000000000000000000000000000000
00000000000000000000001310000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000e757365724163636f756
e74496431000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000e757365724163636f756e7449643200000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000",
    "signatureOrProof":
"0x00000000000000000000000000000000000000000000000000000000000000c075000e4e7a45b0c26ba
d2ea3bd276f6f4fbfec4ef5751ec33859d12d3897eecc09d5df28dd921563105e8e21dc7b4ad825d644b21
8b41982b11f262806d623fa000000000000000000000000000000000000000000000000000000000000076
000000000000000000000000000000000000000000000000000000000000009c0000000000000000000000
000000000000000000000000000000000000000c2000000000000000000000000000000000000000000000
000000000000000066af90667f90664822080b9065ef9065b01830fa6f7b90100000000000000000000000
00000000000080040000000000000000000200040000000000000000200008000000000000002000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00010000000000000000000000000000000000000008000000000000000000000000400000000000000001
40000000000000004000000080000000000000002000000000000000000000000000000000004000000000
00000000000000000000000800000100000000000000000000000000000040000000000000000000000000
00000000000000000000000000000000000000100000000000020000000000f90550f9021a94f1705d25c
81fb6eab1646a08c1eac04e1682c065e1a04a088333602fc5fc4f1782002fd83cb872f641d7564993fcae5
a70619d371b83b901e00000000000000000000000000000000000000000000000000000000000000100000
00000000000000000000000000000000000000000000000000000000001400000000000000000000000000
000000000000000000000000000000000000180000000000000000000000000000000000000000000000
00000000000000001686f6c6445786563757469 6f6e0000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000632060fb00000000000000000000000000000
00000000000000000000000000000000001c00000000000000000000000000000000000000000000000000
000000001348bf100000000000000000000000000000000000000000000000000000000000000000131000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000e757365724163636f756e7449643200000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000e757365724
163636f756e74496431000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000f89994f1705d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e
25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b860000000000000000000000
00000000000000000000000000000000000000000200000000000000000000000000000000000000000000
```

```
0000000000000000000013100000000000000000000000000000000000000000000000000000000000000f
8799487780452339d47a5d86c1a8e77eadb2c46c0ac3ae1a0e86cf26454ff6ddddafa95eaf2baab1df2c06
2c4341cec61ccfafae50eabf8d3b8400000000000000000000000002afcc963a8c7119b700063e17a2d468
46ba8b50e0000000000000000000000000000000000000000000000000000000000000000f9021a94d1043
43c69174447b34981c2c3e536a66c159c48e1a07a752c4d100a96be23f60f072fed1f33da2890fbe45eb15
af52750faa2772a92b901e0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000fd2898e8df49fb8de957a6b75bcd4ba55eb550f8000000000000000000000
00000000000000000000000000000000000000000600000000000000000000000000000000000000000000
0000000000000000001448903901f000000000000000000000000000000000000000000000000000000000
000008000000000000000000000000000000000000000000000000000000000000000c0000000000000000
00000000000000000000000000000000000000000000001000000000000000000000000000000000000000
00000000000000000000000000100000000000000000000000000000000000000000000000000000000000
00001310000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000e75736572416363 6f756e7449643100000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00e757365724163636f756e74496432000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000023ef9023ba0e333a31e86179fb3056
49b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8dec75d7aab85b567b6ccd41ad312451
b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e0462890ce7a673a09850cf8b19f4707e4
f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018028745195ce85510ec4e51adecff39
41207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751ec33859d12d3
897eeccb901000000000000000000000000000000000000080040000000000000002000040000000000000
00200000080000000000002000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000100000000000000000000000000000000000000000000800000
00000000000004000000000000000001400000000000000040000080000000000000200000000000000000
00000000000000000000400000000000000000000000000000000800000001000000000000000000000000
00000040000000000000000000000000000000000000000000000000000000000000001000000000000200
000000000018301ae8e8405f5e100830fa6f784632060fbb83ff83da000000000000000000000000000000
00000000000000000000000000000000000d594ca31306798b41bc81c43094a1e0462890ce7a6738084000
00000a063746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365880000000000000
00000000000000000000000000000000000000000000000000000000000000000000239f90236a0e333a31
e86179fb305649b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8dec75d7aab85b567b6c
cd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e0462890ce7a673a09850c
f8b19f4707e4f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018028745195ce85510ec
4e51adecff3941207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751
ec33859d12d3897eeccb9010000000000000000000000000000000000000800400000000000000020000040
00000000000000200000080000000000002000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000100000000000000000000000000000000000000
08000000000000000000000000004000000000000000001400000000000000040000080000000000000200
00000000000000000000000000000000400000000000000000000000000000000800000001000000000000
00000000000000000040000000000000000000000000000000000000000000000000000000000000001000
00000000020000000000001 8301ae8e8405f5e100830fa6f784632060fbb83af838a000000000000000000
00000000000000000000000000000000000000000000000d594ca31306798b41bc81c43094a1e0462890ce
7a67380a063746963616c2062797a616e74696e65206661756c7420746f6c6572616e63658800000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000045f84
3b84177d6585b25b5b165d65925decc443dbf374941b36a71c0e5ee4bb7e65f3185fc2eaea091c9ac4b8fa
f09fa6d6d21a18d7d2905667f381ee096ae8103235bc8b30100000000000000000000000000000000000000
00000000000000000"
  }
}
```

17.1. Encoded Information

The `encodedInfo` field is defined as a struct with the following fields:

Struture of the EncodedInfo

```
struct EncodedInfo {
  uint256 blockchainId;
  address contractAddress;
  bytes32 eventSignature;
  bytes   eventData;
}
```

where:

1. `blockchainId` is the blockchain identifier
2. `contractAddress` is the 160-bit Ethereum address of the crosschain control contract
3. `eventSignature` is the event function signature
4. `eventData` is the event data needed to re-construct the Merkle Patricia tree

The `encodedInfo` contained in the proof is a hex encoded string created using the ABI encoding structure.
It must therefore be ABI decoded in order to extract the `blockchainId`, `contractAddress`, `eventSignature`, and `eventData` as follows:

Example encodedInfo

```
    let encodedInfo =
'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000007a752c4d100a96be23f60f072fed1f33da2890fbe
45eb15af52750faa2772a920000000000000000000000000000000000000000000000000000000000000008
0000000000000000000000000000000000000000000000000000000000000065ef9065b01830fa6f7b9010
00000000000000000000000000000000000080040000000000000002000040000000000000002000000800
00000000002000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000100000000000000000000000000000000000000080000000000000000000000
00004000000000000000000000140000000040000000800000000000000200000000000000000000000000
00000000400000000000000000000000000000000008000001000000000000000000000000000000400000
00000000000000000000000000000000000000000000000000000000010000000000002000000000000f90
550f9021a94f1705d25c81fb6eab1646a08c1eac04e1682c065e1a04a088333602fc5fc4f1782002fd83cb
872f641d7564993fcae5a70619d371b83b901e000000000000000000000000000000000000000000000000
00000000000000100000000000000000000000000000000000000000000000000000000000000014000000
00000000000000000000000000000000000000000000000000000000000000000180000000000000000000
0000000000000000000000000000000000001686f6c64457865637574696f6e00000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000632060fb0000000
00000000000000000000000000000000000000000000000000000001c00000000000000000000000000000
00000000000000000000000000001348bf1000000000000000000000000000000000000000000000000000
00000000000013100000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000e757365724163636f756e74496432000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000e757365724163636f756e74496431000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000f89994f1705d25c81fb6eab1646a08c1e
ac04e1682c065e1a0f0e25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b8600
00000000000000000000000000000000000000000000000000000000000002000000000000000000000000
00000000000000000000000000000000000000001310000000000000000000000000000000000000000000
0000000000000000000f8799487780452339d47a5d86c1a8e77eadb2c46c0ac3ae1a0e86cf26454ff6dddd
afa95eaf2baab1df2c062c4341cec61ccfafae50eabf8d3b840000000000000000000000002afcc963a8c
7119b700063e17a2d46846ba8b50e000000000000000000000000000000000000000000000000000000000
0000000f9021a94d104343c69174447b34981c2c3e536a66c159c48e1a07a752c4d100a96be23f60f072fe
d1f33da2890fbe45eb15af52750faa2772a92b901e00000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000fd2898e8df49fb8de957a6b75bcd4ba55eb550f80
00000000000000000000000000000000000000000000000000000000000000600000000000000000000000
000000000000000000000000000000000000001448903901f0000000000000000000000000000000000000
00000000000000000000000008000000000000000000000000000000000000000000000000000000000000
000c0000000000000000000000000000000000000000000000000000000000001000000000000000000000
00000000000000000000000000000000000000000000001000000000000000000000000000000000000000
00000000000000000000000013100000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000e757365724163636f756
e74496431000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000e757365724163636f756e74496432000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000'
```

Decoding encodedInfo

```
    let abiCoder = new ethers.utils.AbiCoder
    let decodedEncodedInfo = abiCoder.decode(['uint256', 'address', 'bytes32',
'bytes'], encodedInfo);
    console.log("Decoded encodedInfo: ", decodedEncodedInfo)
```

Example output

```
    Decoded encodedInfo:  [
      BigNumber { _hex: '0x00', _isBigNumber: true },
      '0x0000000000000000000000000000000000000000',
      '0x7a752c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92',

'0xf9065b01830fa6f7b9010000000000000000000000000000000000008004000000000000000000002000040
00000000000000200000080000000000002000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000100000000000000000000000000000000000000
08000000000000000000000000004000000000000001400000000000000040000000800000000000000200
00000000000000000000000000000000040000000000000000000000000000000080000001000000000000
00000000000000000040000000000000000000000000000000000000000000000000000000000000001000
0000000000200000000000f90550f9021a94f1705d25c81fb6eab1646a08c1eac04e1682c065e1a04a08833
3602fc5fc4f1782002fd83cb872f641d7564993fcae5a70619d371b83b901e000000000000000000000000
00000000000000000000000000000000000000100000000000000000000000000000000000000000000000
00000000000000001400000000000000000000000000000000000000000000000000000000000000180000
0000000000000000000000000000000000000000000000000000000000001686f6c64457865637574696f6
e0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000632060fb00000000000000000000000000000000000000000000000000000000000001c000000
00000000000000000000000000000000000000000000000000001348bf1000000000000000000000000000
00000000000000000000000000000000000013100000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000e7573657
24163636f756e74496432000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000e757365724163636f756e7449643100000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000f89994f17
05d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e25c63981e9a617375c8244c8ac144e4e520acc2cb0
8e2d4c3781b1a02c067b86000000000000000000000000000000000000000000000000000000000000002
0000000000000000000000000000000000000000000000000000000000000000131000000000000000000
0000000000000000000000000000000000000000000f8799487780452339d47a5d86c1a8e77eadb2c46c0a
c3ae1a0e86cf26454ff6ddddafa95eaf2baab1df2c062c4341cec61ccfafae50eabf8d3b84000000000000
00000000000002afcc963a8c7119b700063e17a2d46846ba8b50e000000000000000000000000000000000
0000000000000000000000000000000f9021a94d104343c69174447b34981c2c3e536a66c159c48e1a07a7
52c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92b901e00000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000fd2898e8df49fb8de
957a6b75bcd4ba55eb550f800000000000000000000000000000000000000000000000000000000000006
000000000000000000000000000000000000000000000000000000000000001448903901f0000000000000
00000000000000000000000000000000000000000000000080000000000000000000000000000000000000
000000000000000000000000000c0000000000000000000000000000000000000000000000000000000000
00001000000000000000000000000000000000000000000000000000000000000000000001000000000000000
00000000000000000000000000000000000000000000000013100000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000e757365724163636f756e744964310000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000e757365724163636f756e74496432000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000'
    ]
```

The `eventData`, which is always the last element in the decoded `encodedInfo` array equals the `rlpEncodedReceipt` obtained from the Merkle Patricia proof generated off-chain and can be extracted as follows:

Extracting rlpEncodedReceipt

```
const RECEIPT_INDEX = 3
let rlpEncodedReceipt = decodedEncodedInfo[RECEIPT_INDEX]
console.log("rlpEncodedReceipt: ", rlpEncodedReceipt)
```

Example rlpEncodedReceipt

```
    rlpEncodedReceipt:
0xf9065b01830fa6f7b9010000000000000000000000000000000000000800400000000000000020000400
00000000000002000000800000000000002000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000001000000000000000000000000000000000000000
80000000000000000000000000040000000000000014000000000000000400000080000000000000002000
00000000000000000000000000000004000000000000000000000000000000000800000010000000000000
00000000000000000400000000000000000000000000000000000000000000000000000000000000010000
000000002000000000000f90550f9021a94f1705d25c81fb6eab1646a08c1eac04e1682c065e1a04a088333
602fc5fc4f1782002fd83cb872f641d7564993fcae5a70619d371b83b901e0000000000000000000000000
00000000000000000000000000000000000001000000000000000000000000000000000000000000000000
00000000000000014000000000000000000000000000000000000000000000000000000000000001800000
000000000000000000000000000000000000000000000000000000000001686f6c64457865637574696f6e
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000632060fb00000000000000000000000000000000000000000000000000000000000001c0000000
0000000000000000000000000000000000000000000000000001348bf10000000000000000000000000000
00000000000000000000000000000000000131000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000e75736572
4163636f756e74496432000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000e757365724163636f756e74496431000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000f89994f170
5d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e25c63981e9a617375c8244c8ac144e4e520acc2cb08
e2d4c3781b1a02c067b86000000000000000000000000000000000000000000000000000000000000020
00000000000000000000000000000000000000000000000000000000000000013100000000000000000000
000000000000000000000000000000000000000000f8799487780452339d47a5d86c1a8e77eadb2c46c0ac
3ae1a0e86cf26454ff6ddddafa95eaf2baab1df2c062c4341cec61ccfafae50eabf8d3b840000000000000
0000000000002afcc963a8c7119b700063e17a2d46846ba8b50e0000000000000000000000000000000000
000000000000000000000000000000f9021a94d104343c69174447b34981c2c3e536a66c159c48e1a07a75
2c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92b901e000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000fd2898e8df49fb8de9
57a6b75bcd4ba55eb550f800000000000000000000000000000000000000000000000000000000000000060
00000000000000000000000000000000000000000000000000000000000001448903901f00000000000000
00000000000000000000000000000000000000000000000080000000000000000000000000000000000000
00000000000000000000000000c00000000000000000000000000000000000000000000000000000000000
00010000000000000000000000000000000000000000000000000000000000000000010000000000000000
00000000000000000000000000000000000000000000000131000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000e757365724163636f756e74496431000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000e757365724163636f756e74496432000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

The `rlpEncodedReceipt` contains an array of transaction receipts event logs, one of which is the `CrossBlockchainCallExecuted`.

In order to retrieve the Ethereum logs from the `rlpEncodedReceipt`, it must be RLP decoded as follows:

Decoding rlpEncodedReceipt

```
let decodedRlpEncodedReceipt = ethers.utils.RLP.decode(rlpEncodedReceipt);
console.log("Decoded rlpEncodedReceipt", decodedRlpEncodedReceipt)
```

Example output

```
Decoded rlpEncodedReceipt [
  '0x01',
  '0x0fa6f7',

'0x000000000000000000000000000000000008004000000000000000200040000000000000000200000008
00000000000020000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000001000000000000000000000000000000000000000800000000000000000000
00000040000000000000140000000000000000400000008000000000000002000000000000000000000000
00000000004000000000000000000000000000000000800000010000000000000000000000000000004000
0000000000000000000000000000000000000000000000000000000010000000000002000000000000'
,
  [
    [
      '0xf1705d25c81fb6eab1646a08c1eac04e1682c065',
      [Array],

'0x0000000000000000000000000000000000000000000000000000000000000001000000000000000000000
00000000000000000000000000000000000000000140000000000000000000000000000000000000000000
00000000000000000000180000000000000000000000000000000000000000000000000000000000000000
1686f6c64457865637574696f6e00000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000632060fb00000000000000000000000000000000000000000000
0000000000000000001c00000000000000000000000000000000000000000000000000000000001348bf10
00000000000000000000000000000000000000000000000000000000000000131000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000e757365724163636f756e74496432000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000e757365724163636f756e74496
43100000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000'
    ],
    [
      '0xf1705d25c81fb6eab1646a08c1eac04e1682c065',
      [Array],

'0x0000000000000000000000000000000000000000000000000000000000000002000000000000000000000
00000000000000000000000000000000000000000000131000000000000000000000000000000000000000
00000000000000000000000'
    ],
    [
      '0x87780452339d47a5d86c1a8e77eadb2c46c0ac3a',
      [Array],
```

```
'0x0000000000000000000000002afcc963a8c7119b700063e17a2d46846ba8b50e0000000000000000000
000000000000000000000000000000000000000000000'
        ],
        [
          '0xd104343c69174447b34981c2c3e536a66c159c48',
          [Array],

'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000fd2898e8df49fb8de957a6b75bcd4ba55eb550f800000000000000000000000000000000000000000
000000000000000000000600000000000000000000000000000000000000000000000000000000000000014
48903901f00000000000000000000000000000000000000000000000000000000000000800000000000000
0000000000000000000000000000000000000000000000000c000000000000000000000000000000000000
00000000000000000000000000100000000000000000000000000000000000000000000000000000000000
00000010000000000000000000000000000000000000000000000000000000000000000131000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000e757365724163636f756e744964310000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000e757365724163636f7
56e74496432000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000'
        ]
      ]
    ]
```

The Ethereum logs contain the `CrossBlockchainCallExecuted` emitted event log data, which can be extracted using the event's signature, defined as `eventSig`, as follows:

Extracting eventSig

```
    const LOGS_INDEX = 3
    const LOG_DATA_INDEX = 2
    let logs = decodedRlpEncodedReceipt[LOGS_INDEX]
    const eventSig =
ethers.utils.keccak256(ethers.utils.toUtf8Bytes('CrossBlockchainCallExecuted(uint256,a
ddress,bytes)'))
    const EVENT_SIG_INDEX = 1
    let crossBlockchainCallExecutedEventLogData
    for(let log of logs){
        if(log[EVENT_SIG_INDEX] == eventSig){
            crossBlockchainCallExecutedEventLogData = log[LOG_DATA_INDEX]
            break
        }
    }
    console.log("CrossBlockchainCallExecuted event signature: ", eventSig)
    console.log("CrossBlockchainCallExecuted log data: ",
crossBlockchainCallExecutedEventLogData)
```

Example eventSig and log data

```
    CrossBlockchainCallExecuted event signature:
0x7a752c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92
    CrossBlockchainCallExecuted log data:
0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000fd2898e8df49fb8de957a6b75bcd4ba55eb550f8000000000000000000000000000000000000000000
00000000000000000000600000000000000000000000000000000000000000000000000000000000000144
8903901f00000000000000000000000000000000000000000000000000000000000000800000000000000000
000000000000000000000000000000000000000000000000c0000000000000000000000000000000000000
00000000000000000000000001000000000000000000000000000000000000000000000000000000000000
00000100000000000000000000000000000000000000000000000000000000000000001310000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000e75736572416363636f756e7449643100000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000e757365724163636f75
6e744964320000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
```

17.2. Function Call Data

The `CrossBlockchainCallExecuted` event is defined as follows:

Structure of the CrossBlockchainCallExecuted event

```
event CrossBlockchainCallExecuted(
    uint256 destinationBlockchainId,
    address contractAddress,
    bytes functionCallData
)
```

where:

1. `destinationBlockchainId` is the uint256 blockchain ID of the destination blockchain
2. `contractAddress` is the 160-bit Ethereum address of the contract on the remote blockchain to be called
3. `functionCallData` is the ABI encoded function signature and parameter data

This data can be extracted using ABI decoding as follows:

Decoding the CrossBlockchainCallExecuted event

```
    const DESTINATION_BLOCKCHAIN_ID_INDEX = 0
    const CONTRACT_ADDRESS_INDEX = 1
    const FUNCTION_CALL_DATA_INDEX = 2
    let decodedCrossBlockchainCallExecutedEventLogData = abiCoder.decode([ "uint256",
"address", "bytes" ], crossBlockchainCallExecutedEventLogData);
    const decodedCrossBlockchainCallExecutedObj = {
        destinationBlockchainId:
decodedCrossBlockchainCallExecutedEventLogData[DESTINATION_BLOCKCHAIN_ID_INDEX],
        contract:
decodedCrossBlockchainCallExecutedEventLogData[CONTRACT_ADDRESS_INDEX],
        functionCallData:
decodedCrossBlockchainCallExecutedEventLogData[FUNCTION_CALL_DATA_INDEX]
    }
    console.log({decodedCrossBlockchainCallExecutedObj})
```

Example output

```
    {
      decodedCrossBlockchainCallExecutedObj: {
        destinationBlockchainId: BigNumber { _hex: '0x00', _isBigNumber: true },
        contract: '0xFd2898e8DF49fb8de957a6b75Bcd4Ba55EB550F8',
        functionCallData:
'0x8903901f00000000000000000000000000000000000000000000000000000000000000008000000000000
000000000000000000000000000000000000000000000000000c0000000000000000000000000000000000
0000000000000000000000000001000000000000000000000000000000000000000000000000000000000
00000000100000000000000000000000000000000000000000000000000000000000000013100000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000e757365724163636f756e7449643100000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000e757365724163636
f756e74496432000000000000000000000000000000000000'
      }
    }
```

The `functionCallData` encodes the function signature and parameter data of the `completeLeadLeg` function, which is defined as follows:

Function signature of completeLeadLeg

```
function completeLeadLeg(
    string calldata tradeId,
    string calldata sender,
    string calldata receiver,
    uint256 amount
) public returns (bool)
```

The function signature is identified by `0x` followed by the first 4 bytes of the hash of the actual function signature string.

These hex characters are selected from the hash by using slice(0, 10), where 10 = 2 + 2*(4), since 2 hex characters is one byte:

Example function signature

```
    const functionSig =
ethers.utils.keccak256(ethers.utils.toUtf8Bytes('completeLeadLeg(string,string,string,
uint256)')).slice(0, 10)
    console.log("Function signature: ", functionSig)
```

Example output

```
    Function signature:  0x8903901f
```

Since the `functionCallData` contains the function signature as well as the `completeLeadLeg` function parameters, the function signature must be stripped away in order to retrieve the parameters, which can then be decoded:

Retrieving function parameters

```

if(decodedCrossBlockchainCallExecutedObj.functionCallData.startsWith(functionSig)){
        const abiEncodedFunctionParams =
'0x'+decodedCrossBlockchainCallExecutedObj.functionCallData.replace(functionSig, '')
        const functionParams = abiCoder.decode(['string', 'string', 'string',
'uint256'], abiEncodedFunctionParams)
        console.log("Function parameters: ", functionParams)
    }
```

Example output

```
    Function parameters:  [
      '1',
      'userAccountId1',
      'userAccountId2',
      BigNumber { _hex: '0x01', _isBigNumber: true }
    ]
```

The Ethereum Information Decoder section in the appendix defines a decodeInfo function, which can be used to decode the `endcodedInfo` into a data object.

17.3. Application Authentication Parameters

Application authentication parameters are used by the function call contract to verify the contract instantiating the crosschain function call, which is the `XvP` contract.
The authentication parameters consist of the blockchain id and contract address of the `XvP` contract, which are appended to the end of the function call data.

These parameters must then also be registered with the function call contract on the destination blockchain so that the destination blockchain can decode the function call data and then verify them.

The function call contract inherits the `NonAtomicHiddenAuthParams` contract, which is an abstract contract containing two functions, one for encoding the authentication parameters and one for decoding them.

Encode Non-atomic Hidden Authentication Parameters

```
function encodeNonAtomicAuthParams(
    bytes memory _functionCall,
    uint256 _sourceBlockchainId,
    address _sourceContract
  ) internal pure returns (bytes memory) {
    return bytes.concat(_functionCall, abi.encodePacked(_sourceBlockchainId,
_sourceContract));
```

Decode Non-atomic Hidden Authentication Parameters

```
function decodeNonAtomicAuthParams() internal pure returns (uint256
_sourceBlockchainId, address _sourceContract) {
    bytes calldata allParams = msg.data;
    uint256 len = allParams.length;

    assembly {
      calldatacopy(0x0, sub(len, add(52, 8)), 32)
      _sourceBlockchainId := mload(0)
      calldatacopy(12, sub(len, add(20, 8)), 20)
      _sourceContract := mload(0)
    }
}
```

17.4. Signature Or Proof

The information passed by the `signatureOrProof` field is specified by the following struct:

Structure of the signatureOrProof

```
struct SignatureOrProof {
  bytes rlpSiblingNodes;
  bytes32 receiptsRoot;
  bytes32 blockHash;
  bytes rlpBlockHeader;
  bytes rlpBlockHeaderExcludingRound;
  bytes rlpValidatorSignatures;
}
```

where:

1. `rlpSiblingNodes` is the RLP encoded stack of nodes of the Merkle Patricia tree
2. `receiptsRoot` is the root of the receipts tree
3. `blockHash` is the hash of the block
4. `rlpBlockHeader` is the RLP encoded block header that excludes the validator signatures
5. `rlpBlockHeaderExcludingRound` is the RLP encoded block header that excludes the round the block was created on (the round in the genesis block is 0) and validator signatures
6. `rlpValidatorSignatures` is the RLP encoded validator signatures

The `signatureOrProof` is decoded as follows:

Example signatureOrProof

```
    let signatureOrProof =
'0x00000000000000000000000000000000000000000000000000000000000000c075000e4e7a45b0c26ba
d2ea3bd276f6f4fbfec4ef5751ec33859d12d3897eecc09d5df28dd921563105e8e21dc7b4ad825d644b21
8b41982b11f262806d623fa000000000000000000000000000000000000000000000000000000000000076
000000000000000000000000000000000000000000000000000000000000009c0000000000000000000000
0000000000000000000000000000000000000000c200000000000000000000000000000000000000000000
00000000000000000066af90667f90664822080b9065ef9065b01830fa6f7b901000000000000000000000
0000000000000000800400000000000000002000040000000000000000200000080000000000002000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0001000000000000000000000000000000000000000800000000000000000000000000400000000000000001
4000000000000000400000008000000000000020000000000000000000000000000000000040000000000
0000000000000000000000800000100000000000000000000000000000004000000000000000000000000
00000000000000000000000000000000000000100000000000200000000000000f90550f9021a94f1705d25c
81fb6eab1646a08c1eac04e1682c065e1a04a088333602fc5fc4f1782002fd83cb872f641d7564993fcae5
a70619d371b83b901e00000000000000000000000000000000000000000000000000000000000000100000
00000000000000000000000000000000000000000000000000000000001400000000000000000000000000
00000000000000000000000000000000000018000000000000000000000000000000000000000000000000
00000000000000001686f6c64457865637574696f6e0000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000632060fb0000000000000000000000000000
00000000000000000000000000000000001c00000000000000000000000000000000000000000000000000
000000001348bf1000000000000000000000000000000000000000000000000000000000000000013100000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000e757365724163636f756e74496432000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000e757365724
163636f756e744964310000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000f89994f1705d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e
25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b860000000000000000000000
00000000000000000000000000000000000000000200000000000000000000000000000000000000000000
0000000000000000001310000000000000000000000000000000000000000000000000000000000000000f
8799487780452339d47a5d86c1a8e77eadb2c46c0ac3ae1a0e86cf26454ff6dddafa95eaf2baab1df2c06
2c4341cec61ccfafae50eabf8d3b840000000000000000000000002afcc963a8c7119b700063e17a2d468
46ba8b50e0000000000000000000000000000000000000000000000000000000000000000f9021a94d1043
43c69174447b34981c2c3e536a66c159c48e1a07a752c4d100a96be23f60f072fed1f33da2890fbe45eb15
af52750faa2772a92b901e0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000fd2898e8df49fb8de957a6b75bcd4ba55eb550f8000000000000000000000
```

```
00000000000000000000000000000000000000000600000000000000000000000000000000000000000000
0000000000000000001448903901f000000000000000000000000000000000000000000000000000000000
00000800000000000000000000000000000000000000000000000000000000000000c0000000000000000
0000000000000000000000000000000000000000000000100000000000000000000000000000000000000
0000000000000000000000000010000000000000000000000000000000000000000000000000000000000
0000131000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000e75736572416363 6f756e7449643100000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00e7573657241636 36f756e744964320000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000023ef9023ba0e333a31e86179fb3056
49b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8dec75d7aab85b567b6ccd41ad312451
b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e0462890ce7a673a09850cf8b19f4707e4
f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018028745195ce85510ec4e51adecff39
41207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751ec33859d12d3
897eeccb90100000000000000000000000000000008004000000000000000000000200004000000000000
00200000080000000000002000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000100000000000000000000000000000000000000080000000000
00000000000000000400000000000001400000000000000040000000800000000000000200000000000000
00000000000000000000400000000000000000000000000000000000000800000100000000000000000000000
00000400000000000000000000000000000000000000000000000000000000000000000010000000000002 00
000000000018301ae8e8405f5e100830fa6f784632060fbb83ff83da000000000000000000000000000000
00000000000000000000000000000000000d594ca31306798b41bc81c43094a1e0462890ce7a6738084000
00000a063746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365880000000000000
00000000000000000000000000000000000000000000000000000000000000000000239f90236a0e333a31
e86179fb305649b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8dec75d7aab85b567b6c
cd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e0462890ce7a673a09850c
f8b19f4707e4f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018028745195ce85510ec
4e51adecff3941207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751
ec33859d12d3897eeccb90100000000000000000000000000000008004000000000000000000002000040
00000000000000200000080000000000002000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000100000000000000000000000000000000000000
08000000000000000000000000000400000000000001400000000000000040000000800000000000000200
00000000000000000000000000000000400000000000000000000000000000000000080000010000000000000
00000000000000000400000000000000000000000000000000000000000000000000000000000000001000
000000000200000000000018301ae8e8405f5e100830fa6f784632060fbb83af838a000000000000000000
00000000000000000000000000000000000000000000000d594ca31306798b41bc81c43094a1e0462890ce
7a67380a063746963616c2062797a616e74696e65206661756c7420746f6c6572616e636588000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000045f84
3b84177d6585b25b5b165d65925decc443dbf374941b36a71c0e5ee4bb7e65f3185fc2eaea091c9ac4b8fa
f09fa6d6d21a18d7d2905667f381ee096ae8103235bc8b3010000000000000000000000000000000000000
0000000000000000'
```

Decoding signatureOrProof

```
    let decodedSignatureOrProof = abiCoder.decode(['bytes', 'bytes32', 'bytes32',
'bytes', 'bytes', 'bytes'], signatureOrProof);
    console.log("Decoded signatureOrProof: ", decodedSignatureOrProof)
```

Example output

```
    Decoded signatureOrProof:  [

'0xf90667f90664822080b9065ef9065b01830fa6f7b901000000000000000000000000000000000000080
04000000000000000200004000000000000000200000080000000000002000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000100000000000000
00000000000000000000000008000000000000000000000000004000000000000014000000000000000040
00000800000000000000020000000000000000000000000000000000400000000000000000000000000000
00008000001000000000000000000000000000000040000000000000000000000000000000000000000000
0000000000000000000100000000000200000000000000f90550f9021a94f1705d25c81fb6eab1646a08c1e
ac04e1682c065e1a04a088333602fc5fc4f1782002fd83cb872f641d7564993fcae5a70619d371b83b901e
00000000000000000000000000000000000000000000000000000000000000100000000000000000000000
00000000000000000000000000000000000000001400000000000000000000000000000000000000000000
00000000000000000018000000000000000000000000000000000000000000000000000000000000000016
86f6c64457865637574696f6e0000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000632060fb000000000000000000000000000000000000000000000
000000000000000001c0000000000000000000000000000000000000000000000000000000001348bf1000
00000000000000000000000000000000000000000000000000000000000013100000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000e757365724163636f756e7449643200000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000e757365724163636f756e7449643
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000f89994f1705d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e25c63981e9a617375c
8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b860000000000000000000000000000000000000000
00000000000000000000000200000000000000000000000000000000000000000000000000000000000000
00131000000000000000000000000000000000000000000000000000000000000000f8799487780452339d4
7a5d86c1a8e77eadb2c46c0ac3ae1a0e86cf26454ff6ddddafa95eaf2baab1df2c062c4341cec61ccfafae
50eabf8d3b840000000000000000000000000002afcc963a8c7119b700063e17a2d46846ba8b50e000000000
0000000000000000000000000000000000000000000000000000000f9021a94d104343c69174447b34981c
2c3e536a66c159c48e1a07a752c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92b
901e0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000fd2898e8df49fb8de957a6b75bcd4ba55eb550f8000000000000000000000000000000000000000
00000000000000000000000060000000000000000000000000000000000000000000000000000000000000
1448903901f0000000000000000000000000000000000000000000000000000000800000000000000000000
00000000000000000000000000000000000000000000c00000000000000000000000000000000000000000
00000000000000000000000000001000000000000000000000000000000000000000000000000000000000
00000000100000000000000000000000000000000000000000000000000000000000000131000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000e757365724163636f756e74496431000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000e757365724163636
f756e7449643200000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000',
      '0x75000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751ec33859d12d3897eecc',
      '0x09d5df28dd921563105e8e21dc7b4ad825d644b218b41982b11f262806d623fa',

'0xf9023ba0e333a31e86179fb305649b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8d
ec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e046
2890ce7a673a09850cf8b19f4707e4f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018
028745195ce85510ec4e51adecff3941207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd2
76f6f4fbfec4ef5751ec33859d12d3897eeccb9010000000000000000000000000000000000000080040000
00000000000200004000000000000000200000080000000000002000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000100000000000000000000
```

```
0000000000000000000800000000000000000000000000400000000000000140000000000000004000000000
8000000000000002000000000000000000000000000000000040000000000000000000000000000000000080
0000010000000000000000000000000000004000000000000000000000000000000000000000000000000
0000000000000010000000000002000000000000018301ae8e8405f5e100830fa6f784632060fbb83ff83da
00000000000000000000000000000000000000000000000000000000000000000d594ca31306798b41bc81
c43094a1e0462890ce7a67380840000000a063746963616c2062797a616e74696e65206661756c7420746
f6c6572616e6365880000000000000000',

'0xf90236a0e333a31e86179fb305649b260f93ec17355d46409fc4f80d7a8bd885975fc699a01dcc4de8d
ec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e046
2890ce7a673a09850cf8b19f4707e4f2244b2d86a54182218021a6d371969f1a96a47dbeb70d4a00be8018
028745195ce85510ec4e51adecff3941207e6defb6389ca087f98e38aa075000e4e7a45b0c26bad2ea3bd2
76f6f4fbfec4ef5751ec33859d12d3897eeccb9010000000000000000000000000000000000000080040000
0000000000020000400000000000000020000008000000000000200000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000001000000000000000000000
0000000000000000000800000000000000000000000000400000000000000140000000000000004000000000
8000000000000002000000000000000000000000000000000040000000000000000000000000000000000080
0000010000000000000000000000000000004000000000000000000000000000000000000000000000000
0000000000000010000000000002000000000000018301ae8e8405f5e100830fa6f784632060fbb83af838a
00000000000000000000000000000000000000000000000000000000000000000d594ca31306798b41bc81
c43094a1e0462890ce7a67380a063746963616c2062797a616e74696e65206661756c7420746f6c6572616
e6365880000000000000000',

'0xf843b84177d6585b25b5b165d65925decc443dbf374941b36a71c0e5ee4bb7e65f3185fc2eaea091c9a
c4b8faf09fa6d6d21a18d7d2905667f381ee096ae8103235bc8b301'
    ]
```

The rlp encoded sibling nodes can then be extracted as follows:

Extracting encoded sibling nodes

```
    const SIBLING_NODES_INDEX = 0
    let rlpSiblingNodes = decodedSignatureOrProof[SIBLING_NODES_INDEX]
    console.log("rlpSiblingNodes: ", rlpSiblingNodes)
```

Example rlpSiblingNodes

```
    rlpSiblingNodes:
0xf90667f90664822080b9065ef9065b01830fa6f7b9010000000000000000000000000000000000000800
40000000000000002000040000000000000002000008000000000000020000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000100000000000000000
00000000000000000000000080000000000000000000000000040000000000000014000000000000000400
00008000000000000000020000000000000000000000000000000004000000000000000000000000000000
00080000001000000000000000000000000000000400000000000000000000000000000000000000000000
000000000000000000100000000000000200000000000f90550f9021a94f1705d25c81fb6eab1646a08c1ea
c04e1682c065e1a04a088333602fc5fc4f1782002fd83cb872f641d7564993fcae5a70619d371b83b901e0
00000000000000000000000000000000000000000000000000000000000001000000000000000000000000
00000000000000000000000000000000000000014000000000000000000000000000000000000000000000
00000000000000000180000000000000000000000000000000000000000000000000000000000000000168
6f6c64457865637574696f6e00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000632060fb000000000000000000000000000000000000000000000
0000000000000001c00000000000000000000000000000000000000000000000000000000001348bf10000
00000000000000000000000000000000000000000000000000000000000131000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000e757365724163636f756e74496432000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000e757365724163636f756e74496431
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000f89994f1705d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e25c63981e9a617375c8
244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b8600000000000000000000000000000000000000000
00000000000000000000002000000000000000000000000000000000000000000000000000000000000000
0131000000000000000000000000000000000000000000000000000000000000000f8799487780452339d47
a5d86c1a8e77eadb2c46c0ac3ae1a0e86cf26454ff6ddddafa95eaf2baab1df2c062c4341cec61ccfafae5
0eabf8d3b8400000000000000000000000002afcc963a8c7119b700063e17a2d46846ba8b50e0000000000
000000000000000000000000000000000000000000000000000000f9021a94d104343c69174447b34981c2
c3e536a66c159c48e1a07a752c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92b9
01e00000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000fd2898e8df49fb8de957a6b75bcd4ba55eb550f80000000000000000000000000000000000000000
00000000000000000000006000000000000000000000000000000000000000000000000000000000000001
448903901f000000000000000000000000000000000000000000000000000000000000000800000000000
00000000000000000000000000000000000000000000000000c00000000000000000000000000000000000
00000000000000000000000000010000000000000000000000000000000000000000000000000000000000
00000001000000000000000000000000000000000000000000000000000000000000000131000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000e757365724163636f756e74496431000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000e757365724163636f
756e7449643200000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000
```

The rlp encoded sibling nodes are decoded as follows:

Decoding rlpSiblingNodes

```
    let decodedRlpSiblingNodes = ethers.utils.RLP.decode(rlpSiblingNodes);
    console.log("Decoded rlpSiblingNodes: ", decodedRlpSiblingNodes)
```

Example output

```
    Decoded rlpSiblingNodes:  [
      [
        '0x2080',

  '0xf9065b01830fa6f7b9010000000000000000000000000000000000000800400000000000000002000040
  00000000000000200000080000000000002000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000100000000000000000000000000000000000000
  08000000000000000000000000000400000000000000140000000000000004000000080000000000000200
  00000000000000000000000000000000400000000000000000000000000000000080000001000000000000
  00000000000000000040000000000000000000000000000000000000000000000000000000000000001000
  0000000002000000000000f90550f9021a94f1705d25c81fb6eab1646a08c1eac04e1682c065e1a04a08833
  3602fc5fc4f1782002fd83cb872f641d7564993fcae5a70619d371b83b901e000000000000000000000000
  00000000000000000000000000000000000000100000000000000000000000000000000000000000000000
  00000000000000001400000000000000000000000000000000000000000000000000000000000000180000
  0000000000000000000000000000000000000000000000000000000000001686f6c64457865637574696f6
  e0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  000000000632060fb00000000000000000000000000000000000000000000000000000000000001c000000
  00000000000000000000000000000000000000000000000000001348bf1000000000000000000000000000
  00000000000000000000000000000000000013100000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000000000000000000000e7573657
  24163636f756e74496432000000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000e757365724163636f756e7449643100000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000000000000000000f89994f17
  05d25c81fb6eab1646a08c1eac04e1682c065e1a0f0e25c63981e9a617375c8244c8ac144e4e520acc2cb0
  8e2d4c3781b1a02c067b86000000000000000000000000000000000000000000000000000000000000002
  00000000000000000000000000000000000000000000000000000000000000001310000000000000000000
  0000000000000000000000000000000000000000000f8799487780452339d47a5d86c1a8e77eadb2c46c0a
  c3ae1a0e86cf26454ff6ddddafa95eaf2baab1df2c062c4341cec61ccfafae50eabf8d3b84000000000000
  00000000000002afcc963a8c7119b700063e17a2d46846ba8b50e000000000000000000000000000000000
  0000000000000000000000000000000f9021a94d104343c69174447b34981c2c3e536a66c159c48e1a07a7
  52c4d100a96be23f60f072fed1f33da2890fbe45eb15af52750faa2772a92b901e00000000000000000000
  000000000000000000000000000000000000000000000000000000000000000000000fd2898e8df49fb8de
  957a6b75bcd4ba55eb550f800000000000000000000000000000000000000000000000000000000000006
  000000000000000000000000000000000000000000000000000000000000001448903901f0000000000000
  00000000000000000000000000000000000000000000000008000000000000000000000000000000000000
  000000000000000000000000000c0000000000000000000000000000000000000000000000000000000000
  00010000000000000000000000000000000000000000000000000000000000000000001000000000000000
  00000000000000000000000000000000000000000000000013100000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  0000e757365724163636f756e7449643100000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000e757365724163636f756e7449643200000000000
  000000000000000000000000000000000000000000000000000000000000000000000000000000000'
      ]
    ]
```

Verifying the proof first requires that the hash of the leaf node equals the hash of the `rlpEncodedReceipt`, which contains the `CrossBlockchainCallExecuted` log data.

Verifying the hashes

```
    const LEAF_NODE_VALUE_INDEX = 1
    let leafNode = decodedRlpSiblingNodes[decodedRlpSiblingNodes.length-1]
    let leafNodeValue = leafNode[LEAF_NODE_VALUE_INDEX]
    let leafNodeValueHash = ethers.utils.keccak256(leafNodeValue)
    let eventLogHash = ethers.utils.keccak256(rlpEncodedReceipt)
    if (leafNodeValueHash == eventLogHash) {
        console.log("The leaf node value matches the rlpEncodedReceipt (event log
value)")
    }
    console.log(leafNodeValueHash)
    console.log(eventLogHash)
```

Example output

```
    The leaf node value matches the rlpEncodedReceipt (event log value)
    0x576d1d0c5725246abfb79644eede32391bdc413aceeeecbbcd8ac831cd6d9cd4
    0x576d1d0c5725246abfb79644eede32391bdc413aceeeecbbcd8ac831cd6d9cd4
```

Then the `verifyEVMEvent` function verifies the Merkle Patricia tree proof by looping through the parent nodes, each time checking that the hash of child node is present in the parent node.

Verifying the Merkle Patricia tree proof

```
async function verifyEVMEvent(receiptsRoot, decodedRlpSiblingNodes){
    for(let i = 1; i <= decodedRlpSiblingNodes.length; i++){
        const childIndex = decodedRlpSiblingNodes.length - i
        const childNode = decodedRlpSiblingNodes[childIndex]
        const rlpEncodedChildNode = ethers.utils.RLP.encode(childNode)
        const childHash = ethers.utils.keccak256(rlpEncodedChildNode)
        console.log({childHash})
        if(childIndex == 0) {
            console.log("The Merkle Patricia tree only contains the leaf node")
            if(childHash == receiptsRoot) {
                console.log("Proof has been verified. The child node equals the
receiptsRoot.")
                console.log(childHash)
                console.log(receiptsRoot)
                return Promise.resolve()
            }
        } else {
            console.log("The Merkle Patricia tree has multiple levels")
            let parentNode = decodedRlpSiblingNodes[childIndex - 1]
            console.log(childIndex)
            if(parentNode.includes(childHash)){
                console.log("Parent contains hash of child")
            } else {
                return Promise.reject(Error("Proof verification failed"))
            }
            const rlpEncodedParentNode = ethers.utils.RLP.encode(parentNode)
            const parentHash = ethers.utils.keccak256(rlpEncodedParentNode)
            console.log({parentHash})
            if(parentHash === receiptsRoot){
                console.log("Proof has been verified")
                return Promise.resolve()
            }
        }
    }
    return Promise.reject(Error("Proof verification failed"))
}
```

If the Merkle Patricia tree only has one node then the hash of the leaf node is compared to the `receiptsRoot` from the proof:

Verifying the receiptsRoot

```
const RECEIPTS_ROOT_INDEX = 1
let receiptsRoot = decodedSignatureOrProof[RECEIPTS_ROOT_INDEX]
await verifyEVMEvent(receiptsRoot, decodedRlpSiblingNodes)
```

Example output

```
{
  childHash: '0x75000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751ec33859d12d3897eecc'
}
The Merkle Patricia tree only contains the leaf node
Proof has been verified. The child node equals the receiptsRoot.
0x75000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751ec33859d12d3897eecc
0x75000e4e7a45b0c26bad2ea3bd276f6f4fbfec4ef5751ec33859d12d3897eecc
```

Otherwise, if the Merkle Patricia has multiple levels, it must be shown that each child is contained in the hash of the parent:

Example signatureOrProof for multi-leveled tree

```
let multiLevelSignatureOrProof =
'0x00000000000000000000000000000000000000000000000000000000000000c0a68ef2fde868da11460
b5336768c946905e14d91a8429c379822916638491cfc98b89875c6ed258e2119a75a103084b31c7983256
5c509a585c58e607815d1230000000000000000000000000000000000000000000000000000000000004c
00000000000000000000000000000000000000000000000000000000000000000720000000000000000000
00000000000000000000000000000000000000009800000000000000000000000000000000000000000000
0000000000000000003c7f903c4f891a07fc981519f65c2d912d03b3ca485d67c6e5cd50acc7bee7cb6ad4
15b23cb5a87a02c11b3e86221bde1491db39919e3a224f639ad4ca79b42f58677e51f81e6d85ca0eead90d
874d153964545591f90b82519757876c1426e90b7588c44806e39d9408080808080a0a2fb659762dc45522
b563c44c728aaf7513d8169ae24865daaa8c3398d990c668080808080808080f9032e30b9032af90327018
2744eb901000000000000000000000000000000000000080000008000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000040000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000400000000000000000000100000000000000000000000000
00000000000100000000000000000000000000000000000000000000000000000000000000000000000000
0000000f9021df9021a941859c8fe918b3ad74d5cb5d2a22e17dd79cd1b0ce1a07a752c4d100a96be23f60
f072fed1f33da2890fbe45eb15af52750faa2772a92b901e00000000000000000000000000000000000000
000000000000000000000000001000000000000000000000006366b9df60d52c3e3bc9878e0d2d6aa1c2f
0e414000000000000000000000000000000000000000000000000000000000000006000000000000000000
000000000000000000000000000000000000000000001448903901f0000000000000000000000000000000
00000000000000000000000000000008000000000000000000000000000000000000000000000000000000
000000000c0000000000000000000000000000000000000000000000000000000000000010000000000000
00000000000000000000000000000000000000000000000000001000000000000000000000000000000000
00000000000000000000000000000054f2d343032000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000003426f620000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000005416c69636500000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0023ef9023ba0103f1a5f34f2d02c828fd50ad891b921845ae6913ea7ee88e94e14ac27221297a01dcc4de
8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc81c43094a1e0
462890ce7a673a0e9f7a77361aefdee06cd8aeda1708a7d87171a8ec05c4963482100f8b663c868a0956c9
faf6389aa3bc25441efb2fc2de8d742137c4f18da08f6a8a5f9e80e7036a0a68ef2fde868da11460b53367
68c946905e14d91a8429c379822916638491cfcb90100000000000000000000000000000000000000800000
```

```
00800000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000004000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000040000000000000
00000001000000000000000000000000000000000000010000000000000000000000000000000000000000
0000000000000000000000000000000000000000001830276808405f5e100830c4f768463246fc9b83ff83
da00000000000000000000000000000000000000000000000000000000000000000d594ca31306798b41bc
81c43094a1e0462890ce7a673808400000000a063746963616c2062797a616e74696e65206661756c74207
46f6c6572616e63658800000000000000000000000000000000000000000000000000000000000000000000
00000000000000239f90236a0103f1a5f34f2d02c828fd50ad891b921845ae6913ea7ee88e94e14ac27221
297a01dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4934794ca31306798b41bc
81c43094a1e0462890ce7a673a0e9f7a77361aefdee06cd8aeda1708a7d87171a8ec05c4963482100f8b66
3c868a0956c9faf6389aa3bc25441efb2fc2de8d742137c4f18da08f6a8a5f9e80e7036a0a68ef2fde868d
a11460b5336768c946905e14d91a8429c379822916638491cfcb9010000000000000000000000000000000
00000800000008000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000004000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000040
00000000000000000001000000000000000000000000000000000000010000000000000000000000000000
0000000000000000000000000000000000000000000000000000001830276808405f5e100830c4f7684632
46fc9b83af838a00000000000000000000000000000000000000000000000000000000000000000d594ca3
1306798b41bc81c43094a1e0462890ce7a67380a063746963616c2062797a616e74696e65206661756c742
0746f6c6572616e636588000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000045f843b84109248520c47341fb9f878f7d4f70920ad95330cfe7d1731b7
8c32863ce4088d93347500668d27c44ae25b909557fc73c7dfea1042095785bf3b845a809af61390000000
0000000000000000000000000000000000000000000000000'
    let decodedMultiLevelSignatureOrProof = abiCoder.decode(['bytes', 'bytes32',
'bytes32', 'bytes', 'bytes', 'bytes'], multiLevelSignatureOrProof);
    let multiLevelReceiptsRoot =
decodedMultiLevelSignatureOrProof[RECEIPTS_ROOT_INDEX]
    let rlpMultiLevelSiblingNodes =
decodedMultiLevelSignatureOrProof[SIBLING_NODES_INDEX]
    const decodedRlpMultiLevelSiblingNodes =
ethers.utils.RLP.decode(rlpMultiLevelSiblingNodes)
    await verifyEVMEvent(multiLevelReceiptsRoot, decodedRlpMultiLevelSiblingNodes)
```

Example output

```
    {
      childHash: '0xa2fb659762dc45522b563c44c728aaf7513d8169ae24865daaa8c3398d990c66'
    }
    The Merkle Patricia tree has multiple levels
    1
    Parent contains hash of child
    {
      parentHash: '0xa68ef2fde868da11460b5336768c946905e14d91a8429c379822916638491cfc'
    }
    Proof has been verified
```

Chapter 18. Corda Proof Verification

When a bank connected to both a Corda network and an enterprise Ethereum network,
a transaction is created from an earmarked Digital Collateral Receipt (DCR) on the Corda chain. The Ethereum platform will receive a signature-based proof of the transaction, which includes the underlying wire transaction, and the signatures of all parties involved in the transaction.

In order to verify the transaction's validity from the given signature-based proof, the transaction has to be signed by at least two registered participants.

It is important to note that this proving methodology does not achieve the same level of trustlessness achieved by proofs on Ethereum with validators. The lack of a trust-less setup on Corda is, therefore, alleviated by adding the signature of the on-boarded custodian who is intended to receive the securities
to the list of signatures that are verified. In doing this, it is trusted that the receiver has fulfilled the following Corda transaction validation requirements:

- The full Corda transaction history (full Merkle tree) has been validated.
- The Corda contract code has been executed to verify the contractual validity of the transaction.

18.1. Verification Steps

18.1.1. Onboarding a Source Chain

The EDDSA public keys of the trusted parties are on-boarded in preparation of the transaction verification.
Examples of such parties are the bank selling the securities, a Corda notary, or another trusted party.

18.1.2. AMQP/1.0 Deserialization

AMQP/1.0 deserialization is the process of reconstructing an object from an AMQP/1.0 serialized byte array, which is required in order to perform the proof verification.

The AMQP serialization format uses the concept of a `Fingerprint` to uniquely identify objects serialized in a proton graph. This means that by using known `Fingerprint` 's of a Corda wire transaction, only the elements required for the proof can be extracted from the byte array during deserialization. A detailed explanation of the Corda transaction serialization format is provided in the Corda Transaction Serialization section of the appendix.

The elements required for the proof include the following:

- The salt used for computing the nonce.
- The wire transaction's serialized component groups with their group index.
- The wire transaction's outputs and commands, containing the contract type, command, amount and currency which are specific to the Corda based securities.
- The list of signatures with public keys and metadata.

A json-formatted example of the deserialized signed transaction is shown in the Corda Transaction Data Structure section in the appendix.

18.1.3. Creating the Signature-Based Proof

An extendable list of component groups used in Corda transactions are shown in the table.
The ordinal column shows the fixed group index used to calculate the transaction nonce.

Table 53. Transaction Component Groups

Component Group	Ordinal
INPUTS_GROUP	0
OUTPUTS_GROUP	1
COMMANDS_GROUP	2
ATTACHMENTS_GROUP	3
NOTARY_GROUP	4
TIMEWINDOW_GROUP	5
SIGNERS_GROUP	6
REFERENCES_GROUP	7
PARAMETERS_GROUP	8

The Merkle tree of a Corda (wire) transaction contains leaves, which are calculated as the hashes (`hc`) of its component groups.

The hash of each serialized component group, to be used as a Merkle tree leaf, is computed using the following equation:

```
hc = HASH(HASH(HASH(HASH(nonce || serialized)) || serialized))
```

where:

1. `HASH` equals the `SHA-256` hash function and a double `SHA-256` is used to prevent length extension attacks
2. `||` denotes concatenation

The method to compute a nonce is based on the provided salt, the component group's fixed index (or ordinal) and the
component's internal index inside it's parent group. It is computed using the following equation:

```
nonce = HASH(HASH(salt || group index || internal index))
```

where:

1. `HASH` equals the `SHA-256` hash function
2. `||` denotes concatenation

18.1.4. Verifying the Signature-Based Proof

Signature-based proofs involve signed Corda transactions or trades containing signatures over the original transaction root, or over a partial tree root. The signature scheme used for a particular proof is indicated in the meta-data of the signature, which could be one of three schemes, namely SECP256K1, SECP256R1, and ED25519.

On-chain verification of the signature-based proofs are described in the Corda Signature-Based Proof Verification chapter.

Chapter 19. Corda Signature-Based Proof Verification

There are two types of signature-based verification schemes used to verify Corda transaction proofs and Corda trade proofs. A transaction-based proof requires the transaction root, witnesses, flags and values. A trade-based proof only requires the transaction root, since the transaction id equals the trade id and a single notary signature is given over the transaction id.

The Corda signature-based proof is verified on-chain by a Solidity contract. In alignment with the EEA specification, discussed in the Crosschain Protocol Stack chapter, the Solidity function that performs the verification is called `decodeAndVerifyEvent` and belongs to the `CrosschainVerifier` interface. This interface and function is defined as follows:

CrosschainVerifier interface

```
interface CrosschainVerifier {
  function decodeAndVerifyEvent(uint256 blockchainId, bytes32 eventSig, bytes calldata
encodedInfo, bytes calldata signatureOrProof) external view;
}
```

where:

1. `blockchainId` is the uint256 identifying the bank's Corda ledger
2. `eventSig` is the EventSig struct
3. `encodedInfo` is the EncodedInfo Struct
4. `signatureOrProof` is the Merkle tree proof

The following sections describe how the `encodedInfo` and `signatureOrProof` parameters are constructed.

19.1. Encoded Information

The `encodedInfo` parameter of the `decodeAndVerifyEvent` function is defined as a struct with the following fields:

Structure of EncodedInfo

```
struct EncodedInfo {
  uint256 blockchainId;
  address contractAddress;
  bytes32 eventSignature;
  bytes eventData;
}
```

where:

1. `blockchainId` is the Corda ledger identifier
2. `contractAddress` is the 160-bit Ethereum address of the crosschain control contract
3. `eventSignature` is the event function signature
4. `eventData` is the event data needed to re-construct the transaction Merkle tree

The `eventData` contains partial Corda transaction data needed to verify inclusion in the transaction tree via a Merkle multi-proof. It consists of the following fields:

Structure of EventData

```
struct EventData {
  bytes callParameters;
  string hashAlgorithm;
  bytes32 privacySalt;
  ComponentData componentData;
}
```

where:

1. `callParameters` are the parameters of the crosschain function
2. `hashAlgorithm` is the hash algorithm used in the Merkle tree - only SHA-256 is currently supported
3. `privacySalt` is the salt needed to compute a nonce when calculating a Merkle tree leaf from a Corda component group element
4. `componentData` is the component data that gets hashed and becomes the value that must be proven to be included in the Merkle tree

The structure of the `ComponentData` is as follows:

Structure of ComponentData

```
struct ComponentData {
  uint8 groupIndex;
  uint8 internalIndex;
  bytes encodedBytes;
}
```

where:

1. `groupIndex` is the global component group index
2. `internalIndex` is the internal component group index
3. `encodedBytes` contains a hex-encoded Corda component group element

The following code examples show how to create the encoded information by first encoding the function signature, followed by the event data, which are both then used in the encoding of the `encodedInfo`.

Example encoding the function signature

```
    const functionSignature =
web3.eth.abi.encodeFunctionSignature('requestFollowLeg(string,string,string,address,ui
nt256,uint256)');
```

Example encoding the call parameters

```
    const receiver = 'Tz1QYXJ0eUEsIEw9TG9uZG9uLCBDPUdC'
    const sender = 'Tz1QYXJ0eUIsIEw9TmV3IFlvcmssIEM9VVM='
    const controlContract = '0xc23cdfef6ec7b1b39c6cb898d7acc71437f167bd'
    const sourceBlockchainId = '0x03'
    const holdAmount = '0xF4240'
    const tradeId = '475a36b9'
    const functionParameters = web3.eth.abi.encodeParameters(['string', 'string',
'string', 'address', 'uint256', 'uint256'],
      [tradeId, sender, receiver, controlContract, sourceBlockchainId, holdAmount])
    const callParameters = functionSignature + functionParameters.substring(2);
```

Calculating the event data

```
    const componentGroup =
'0x0080C5620000000000001D000000E420000000300A3226E65742E636F7264613A51307A55474E2F4B367
777777975496C4E66335261773D3DD000000398000000000500A3226E65742E636F7264613A372B307474685
24E384B742B546255446266663837413D3DC0820100A3216E65742E636F7264613A6A6176612E736563757
26974792E5075626C69634B6579A05B3059301306072A8648CE3D020106082A8648CE3D030107034200040
38D226DCD0FA574316DA478AA75225E6CE18F65CBD96E60BF3C8251B196541756E5DCF7CCAB21B712601ED
0278501F2F33D0B5FDAA4C09E62639464E4910871A12F6E65742E636F7264612E73616D706C65732E65786
16D706C652E636F6E7472616374732E444352436F6E747261637400A3226E65742E636F7264613A446C645
73979533474424F7A6536717636553451544413D3DD0000001D600000008A10347425000A3226E65742E636
F7264613A48394B4F6938616755757573674B4B69334D45423378673D3DC0920200A3226E65742E636F72646
13A6E6764776274366B5254306C356E6E313675663837413D3DC0180640A1024742A1064C6F6E646F6EA10
6506172747941404000A3216E65742E636F7264613A6A6176612E73656375726974792E5075626C69634B6
579A02C302A300506032B65700321005918F8DB2515D38F0074543A3AC2BDB5B18A40DD733EBE42B7F75E3
88F66F48700A3226E65742E636F7264613A726E69773742324D7169377A6C6B50704B6D4A3737413D3DC01
3024098FCF70523051C49968C8E4033A15AD49000A3226E65742E636F7264613A48394B4F6938616755757
3674B4B69334D45423378673D3DC0940200A3226E65742E636F7264613A6E6764776274366B5254306C356
E6E313675663837413D3DC01A0640A1025553A1084E657720596F726BA106506172747942404000A3216E6
5742E636F7264613A6A6176612E73656375726974792E5075626C69634B6579A02C302A300506032B65700
32100F415ECD394E50C1752AD515773880CD5F6FB78FA0481ACCDBFA600D05B7E549740A109454152 4D415
24B4544A10834373561333662 39A10731303030303030304000A3226E65742E636F7264613A48394B4F69386
167557573674B4B69334D45423378673D3DC0920200A3226E65742E636F7264613A6E6764776274366B525
4306C356E6E313675663837413D3DC0180640A1024742A1064C6F6E646F6EA1064E6F7461727940 4000A32
16E65742E636F7264613A6A6176612E73656375726974792E5075626C69634B6579A02C302A300506032B6
570032100A8CEA277AA9102D266D04BC3B5C7CB2B2C144EA42937E0128186FC65F256B64F0080C56200000
0000002D000000A600000001D000000A57000000 0A0080C5620000000000005D000000184000000 05A1296
E65742E636F7264612E636F72652E636F6E7472616374732E5472616E73616374696F6E537461746540450
080C5620000000000003C02602A3226E65742E636F7264613A51307A55474E2F4B3677777777975496C4E663
35261773D3D40D0000011C000000050080C5620000000000004C04607A10A636F6E73747261696E74A1012
AC03001A12D6E65742E636F7264612E636F72652E636F6E7472616374732E4174746163686D656E74436F6
```

```
E73747261696E74404041420080C5620000000000004C01807A108636F6E7472616374A106737472696E674
540404142080C5620000000000004C03907A10464617461A1012AC02901A1266E65742E636F7264612E636
F72652E636F6E7472616374732E436F6E74726163745374617465404041420080C5620000000000004C0180
7A10B656E63756D6272616E6365A103696E74454040424200800C5620000000000004C02D07A1066E6F74617
279A11D6E65742E636F7264612E636F72652E6964656E746974792E5061727479454040414200800C562000
000000005C09605A12D6E65742E636F7264612E636F72652E636F6E7472616374732E4174746163686D656
E74436F6E73747261696E7440C03001A12D6E65742E636F7264612E636F72652E636F6E7472616374732E4
174746163686D656E74436F6E73747261696E740080C5620000000000003C02602A3226E65742E636F72646
13A4D6766362F7332416A6B5A6154382F6255396E4E53513D3D40450080C5620000000000005C08805A1266
E65742E636F7264612E636F72652E636F6E7472616374732E436F6E7472616374537461746540C02901A12
66E65742E636F7264612E636F72652E636F6E7472616374732E436F6E74726163745374617465080C5620
0000000003C02602A3226E65742E636F7264613A5A326932426D6F3532456675634658A38423243505 13
D3D40450080C5620000000000005C0D105A11D6E65742E636F7264612E636F72652E6964656E746974792E5
06172747940450080C5620000000000003C02602A3226E65742E636F7264613A48394B4F693861675575736
74B4B69334D4542337867 3D3D40C07B020080C5620000000000004C03307A1046E616D65A1256E65742E636
F7264612E636F72652E6964656E746974792E436F72646158353030 4E616D654540404142080C56200000
0000004C02F07A1096F776E696E674B6579A1012AC01A01A1176A6176612E7365637572697479 2E5075626
C69634B657940404142080C5620000000000005D00000014400000005A1256E65742E636F7264612E636F7
2652E6964656E746974792E436F726461583530304E616D6540450080C5620000000000003C02602A3226E6
5742E636F7264613A6E6764776274366B5254306C356E6E313675663837413D3D40C0E3060080C56200000
0000004C01A07A10A636F6D6D6F6E4E616D65A106737472696E674540404242080C5620000000000004C01
707A107636F756E747279A106737472696E674540404142080C5620000000000004C01807A1086C6F63616
C697479A106737472696E674540404142080C5620000000000004C01C07A10C6F7267616E69736174696F6
EA106737472696E674540404142080C5620000000000004C02007A1106F7267616E69736174696F6E556E6
974A106737472696E6745404042420080C5620000000000004C01507A1057374617465A106737472696E674
5404042420080C5620000000000005C0D605A1366E65742E636F7264612E636F72652E636F6E7472616374 7
32E5369676E617475726541747461636 86D656E74436F6E73747261696E7440C03001A12D6E65742E636F7
264612E636F72652E636F6E7472616374732E4174746163686D656E74436F6E73747261696E740080C5620
0000000003C02602A3226E65742E636F7264613A372B30747468524E384B742B54625544626663837413
D3D40C036010080C5620000000000004C02907A1036B6579A1012AC01A01A1176A6176612E7365637572697
4792E5075626C69634B657940404142080C5620000000000005D00000024600000005A1296E65742E636F7
264612E73616D706C65732E6578616D706C652E7374617465732E4443525374617465 40C07603A1246E657
42E636F7264612E636F72652E636F6E7472616374732E4C696E656172537461746541266E65742E636F726
4612E636F72652E636F6E7472616374732E436F6E74726163745374617465A1256E65742E636F7264612E6
36F72652E736368656D61732E517565727961626C655374617465 0080C5620000000000003C02602A3226E6
5742E636F7264613A446C6457397953347442 4F7A653671763655345154413D3D40D000000167000000080
080C5620000000000004C01807A10863757272656E6379A106737472696E674540404242080C5620000000
00004C02D07A106697373756572A11D6E65742E636F7264612E636F72652E6964656E746974792E5061727
4794540404242080C5620000000000004C03B07A1086C696E6561724964A1296E65742E636F7264612E636
F72652E636F6E7472616374732E556E69717565496 4656E7469666965724540404142080C562000000000
004C02C07A1056F776E6572A11D6E65742E636F7264612E636F72652E6964656E746974792E50617274794
5404042420080C5620000000000004C01507A10570726F6F66A106737472696E6745404042420080C562000
000000004C01607A10673746174757 3A106737472696E6745404042420080C5620000000000004C01707A10
7747261646549 64A106737472696E674540404242080C5620000000000004C01507A10576616C7565A1067
37472696E6745404042420080C5620000000000005C0AC05A1246E65742E636F7264612E636F72652E636F6
E7472616374732E4C696E656172537461746540C04F02A1246E65742E636F7264612E636F72652E636F6E7
472616374732E4C696E6561725374617465A1266E65742E636F7264612E636F72652E636F6E74726163747
32E436F6E74726163745374617465 0080C5620000000000003C02602A3226E65742E636F7264613A416A4D5
A704D6F6A5268685A36364A6B74335762 43413D3D40450080C5620000000000005C0AE05A1256E65742E636
F7264612E636F72652E736368656D61732E517565727961626C65537461746540C05002A1256E65742E636
F7264612E636F72652E736368656D61732E517565727961626C655374617465A1266E65742E636F7264612
```

```
E636F72652E636F6E7472616374732E436F6E7472616374537461746500 80C562000000000003C02602A32
26E65742E636F7264613A423832635346365430345443506 66C67587332626B673D3D40450080C56200000
0000005C0A505A1296E65742E636F7264612E636F72652E636F6E7472616374732E556E697175654964656
E746966696572404500 80C562000000000003C02602A3226E65742E636F7264613A726E69773742324D716
9377A6C6B50704B6D4A3737413D3D40C043020080C562000000000004C01A07A10A65787465726E616C496
4A106737472696E674540404242008 0C562000000000004C01007A1026964A1047575696445404041420 08
0C562000000000009C10100'
    const hashAlgorithm = 'SHA-256'
    const privacySalt =
'0x39F4600478DAF2773F914E7CCC5230E678C4DDDD2E7F51AD43659B1645B89CFF'
    const eventData = web3.eth.abi.encodeParameters(
      [
        {
          'EventData': {
            'callParameters': 'bytes',
            'hashAlgorithm': 'string',
            'privacySalt': 'bytes32',
            'ComponentData': {
              'groupIndex': 'uint8',
              'internalIndex': 'uint8',
              'encodedBytes': 'bytes',
            },
          },
        },
      ],
      [
        {
          'callParameters': callParameters,
          'hashAlgorithm': hashAlgorithm,
          'privacySalt': privacySalt,
          'ComponentData': {
            'groupIndex': '0x1',
            'internalIndex': '0x0',
            'encodedBytes': componentGroup,
          }
        }
      ]
    )
```

Example of encodedInfo

```
    eventSig = '0x0000000000000000000000000000000000000000000000000000000000000000'
    const encodedInfo = web3.eth.abi.encodeParameters(['uint256', 'address',
'bytes32', 'bytes'],
      [sourceBlockchainId, controlContract, eventSig, eventData])
    console.log({ encodedInfo })
```

19.2. Signature Or Proof

The information passed by the `signatureOrProof` parameter is specified by the following struct:

Structure of Signatures

```
struct Signatures {
  ProofData proofData;
  Signature[] signatures;
}
```

where:

1. `proofData` is the data contained in the proof, e.g. witnesses, flags and values or just a root when used for trade verification
2. `signatures` is the array of signatures of type `Signature`

The `ProofData` type a struct defined with the following fields:

Structure of ProofData

```
struct ProofData {
  bytes32 root;
  bytes32[] witnesses;
  uint8[] flags;
  bytes32[] values;
}
```

where:

1. `root` is the Merkle tree root
2. `witnesses` are the Merkle multi-proof witnesses
3. `flags` are the Merkle multi-proof flags
4. `values` are the Merkle multi-proof leaves

The `Signature` type a struct defined with the following fields:

Structure of Signature

```
struct Signature {
  uint256 by;
  uint256 sigR;
  uint256 sigS;
  uint256 sigV;
  bytes meta;
}
```

where:

1. `by` is the 160-bit Ethereum address or the 256-bit `ED25519` public key
2. `sigR` is the ECDSA/EDDSA signature's R value
3. `sigS` is the ECDSA/EDDSA signature's S value
4. `sigV` is the ECDSA signature's V value
5. `meta` is the signature meta-data containing the platform version, schema number and (optional) partial Merkle tree root that was signed

Encoding EventData

```
    root = '0xA46E9FDC6E6DD4A659E10C1B42619CE078996AB009C8EA82BEDAEDF20C0B939A'
    witnesses = [
      '0x1A3357D5D28F6BD415D64CED45D9975D7992FCEB997DA33227E0242A36792257',
      '0x91BA378A1776FB1CDF669C9E36AA35F1CEFCAF6EF3F9F2D0013937EFC2A747E8',
      '0x868E2E6CB6197ECFF92825120DCDCB923BAA4BCD38DB9C69DFE65FE9673C8DA9',
      '0x1B58857B3475DD6B65F27F85D993C97AAB833B66E8C5270A190A50311CE11CCE',
    ]
    flags = ['0x03', '0x01', '0x01', '0x01']
    values = ['0x67DE8C49537263FBF81A69FA214DF3903EBA581B63A9D8071DD83270887FC61D']
    signatures = [{
      'by': '0x5918F8DB2515D38F0074543A3AC2BDB5B18A40DD733EBE42B7F75E388F66F487',
      'sigR': '0x63787BED632C6F7F963D5D813A582BE50D671EBB1289FCFFCCB1BA319DE212F1',
      'sigS': '0xD1C29ADBABB70F8F617981F9ED7999167695C040C2741C488B0706BE7D9FB90F',
      'sigV': '0x0000000000000000000000000000000000000000000000000000000000000000',
      'meta':
'0x0000000A00000004A46E9FDC6E6DD4A659E10C1B42619CE078996AB009C8EA82BEDAEDF20C0B939A'
    }, {
      'by': '0xF415ECD394E50C1752AD515773880CD5F6FB78FA0481ACCDBFA600D05B7E5497',
      'sigR': '0x14EAE51395AF6D278B5174D701689C3B7CF86030642392C9371CB60E3C907FC4',
      'sigS': '0xDE9FBDE6C01CBDED2F8FBF9E9399E8731F825E10B3D7341ED73C74E0FC96AB08',
      'sigV': '0x0000000000000000000000000000000000000000000000000000000000000000',
      'meta':
'0x0000000A00000004A46E9FDC6E6DD4A659E10C1B42619CE078996AB009C8EA82BEDAEDF20C0B939A'
    }, {
      'by': '0xA8CEA277AA9102D266D04BC3B5C7CB2B2C144EA42937E0128186FC65F256B64F',
      'sigR': '0x4FA8BAAF04CA29502BB4D51A267E64B9657C01E88A14969DC9D30C4C43003B57',
      'sigS': '0x6DE689B91B87ADD6D354013CC7CF454435078B4EF22F4940881090FCD6E2420A',
      'sigV': '0x0000000000000000000000000000000000000000000000000000000000000000',
      'meta':
'0x0000000A00000004CEF6CD791B094A69160E3CCACCDA07527EB594F1FE92B9044A3BCC96BA45B440'
    }]
    const signatureOrProof = web3.eth.abi.encodeParameters(
      [
        {
          'Signatures': {
            'ProofData': {
              'root': 'bytes32',
              'witnesses': 'bytes32[]',
              'flags': 'uint8[]',
              'values': 'bytes32[]',
```

```
            },
            'Signature[]': {
              'by': 'uint256',
              'sigR': 'uint256',
              'sigS': 'uint256',
              'sigV': 'uint256',
              'meta': 'bytes',
            }
          },
        },
      ],
      [
        {
          'ProofData': {
            'root': root,
            'witnesses': witnesses,
            'flags': flags,
            'values': values,
          },
          'Signature': signatures,
        }
      ]
    )
    console.log({ signatureOrProof })
```

19.3. Proof Verification

The first step in the proof verification is to validate the event by verifying the function call parameters against values extracted from the outputs component group element. The `callParameters` and `componentData` are fields contained in the `eventData` struct.

The following code block shows the `validateEvent` function belonging to the Corda Solidity library, which is called with the `eventData`, `handlers`, `proofData` and `signatures`.

Corda validateEvent

```
function validateEvent(Corda.EventData calldata eventData, Corda.ParameterHandler[]
calldata handlers, Corda.ProofData calldata proofData, Corda.Signature[] calldata
signatures) public view returns (bool)
```

The outputs component group element is AMQP-encoded, which means that registered parameter handlers are required to extract values from the encoding. The following struct shows the structure of the parameter handlers.

Structure of ParameterHandler

```
struct ParameterHandler {
  string fingerprint;
  uint8 componentIndex;
  uint8 describedSize;
  string describedType;
  bytes describedPath;
  string solidityType;
  string parser;
}
```

where:

1. `fingerprint` is the fingerprint used when extracting data from the Corda component
2. `componentIndex` is the index in list of extracted items for this fingerprint
3. `describedSize` is the size of the structure that was extract under this fingerprint
4. `describedType` is the extracted AMQP type
5. `describedPath` is the path to walk for nested objects in extracted items
6. `solidityType` is the Solidity type of the extracted value
7. `parser` is the parser used to parse the extracted element

Parameter handlers are stored in the cross-chain messaging contract which is responsible for verification of the event. It maps a system id to a map that takes a function signature to a list of parameter handlers.

The mapping of parameter handlers

```
contract CrosschainMessaging is CrosschainVerifier {
mapping(uint256 => mapping(string => Validator.ParameterHandler[])) parameterHandlers;
}
```

Next, the `validateEvent` function verifies the multivalued Merkle proof which is provided in the `proofData`.

A sparse Merkle multi-proof is an efficient way of verifying the inclusion of multiple component group elements in a Corda transaction tree, since it can prove the presence of multiple leaves in the same Merkle tree.

The Merkle Solidity library `verifyMultiProof` function takes the `proofData.root`, `proofData.witnesses`, `proofData.flags`, `proofData.values`, and verifies a Merkle multiproof, constructed from a partial Merkle tree.

Merkle verifyMultiProof function

```
function verifyMultiProof(bytes32 root, bytes32[] memory proof, uint8[] memory flags,
bytes32[] memory values) public pure returns (bool)
```

The verification process rebuilds the root hash by traversing the tree up from the leaves. The root is calculated by consuming either a leaf value from the values of the proof, or a witness from the witnesses of the proof, or a previously calculated value off the stack, and producing hashes from them. At the end of the process, the last hash should contain the root of the merkle tree.

Finally, the signatures are verified provided in the signature data. A transaction-based verification requires at least one notary signature and at least two participant signatures. A trade-based verification requires at least one notary signature.

Chapter 20. Appendix A

20.1. Ethereum Information Decoder

The following decodeInfo function decodes the `encodeInfo` data structure returned with the Ethereum proof.
It will decode either the `requestFollowLeg`, `completeLeadLeg` or `cancelLeadLeg` function parameters.

decodeInfo function

```
    async function decodeInfo(encodedInfo){
        let abiCoder = new ethers.utils.AbiCoder
        let decodedEncodedInfo = abiCoder.decode(['uint256', 'address', 'bytes32',
'bytes'], encodedInfo);
        const RECEIPT_INDEX = 3
        let rlpEncodedReceipt = decodedEncodedInfo[RECEIPT_INDEX]
        let decodedRlpEncodedReceipt = ethers.utils.RLP.decode(rlpEncodedReceipt);
        const LOGS_INDEX = 3
        const LOG_DATA_INDEX = 2
        let logs = decodedRlpEncodedReceipt[LOGS_INDEX]
        const eventSig =
ethers.utils.keccak256(ethers.utils.toUtf8Bytes('CrossBlockchainCallExecuted(uint256,a
ddress,bytes)'))
        const EVENT_SIG_INDEX = 1
        let crossBlockchainCallExecutedEventLogData
        for(let log of logs){
           if(log[EVENT_SIG_INDEX] == eventSig){
              crossBlockchainCallExecutedEventLogData = log[LOG_DATA_INDEX]
              break
           }
        }
        const DESTINATION_BLOCKCHAIN_ID_INDEX = 0
        const CONTRACT_ADDRESS_INDEX = 1
        const FUNCTION_CALL_DATA_INDEX = 2
        let decodedCrossBlockchainCallExecutedEventLogData = abiCoder.decode([
"uint256", "address", "bytes" ], crossBlockchainCallExecutedEventLogData);
        const decodedCrossBlockchainCallExecutedObj = {
            destinationBlockchainId:
decodedCrossBlockchainCallExecutedEventLogData[DESTINATION_BLOCKCHAIN_ID_INDEX],
            contract:
decodedCrossBlockchainCallExecutedEventLogData[CONTRACT_ADDRESS_INDEX],
            functionCallData:
decodedCrossBlockchainCallExecutedEventLogData[FUNCTION_CALL_DATA_INDEX]
        }
        let functionParamsObj = {}
        const completeLeadLegFunctionSignature =
"completeLeadLeg(string,string,string,uint256)"
        const requestFollowLegFunctionSignature =
"requestFollowLeg(string,string,string,address,uint256,uint256)"
```

```
        const cancelLeadLegFunctionSignature = "cancelLeadLeg(string,string,string)"

        const completeLeadLegFunctionSelector =
ethers.utils.keccak256(ethers.utils.toUtf8Bytes(completeLeadLegFunctionSignature)).sli
ce(0, 10)
        const requestFollowLegFunctionSelector =
ethers.utils.keccak256(ethers.utils.toUtf8Bytes(requestFollowLegFunctionSignature)).sl
ice(0, 10)
        const cancelLeadLegFunctionSelector =
ethers.utils.keccak256(ethers.utils.toUtf8Bytes(cancelLeadLegFunctionSignature)).slice
(0, 10)


if(decodedCrossBlockchainCallExecutedObj.functionCallData.startsWith(completeLeadLegFu
nctionSelector)){
            const abiEncodedFunctionParams =
'0x'+decodedCrossBlockchainCallExecutedObj.functionCallData.replace(completeLeadLegFun
ctionSelector, '')
            const functionParams = abiCoder.decode(['string', 'string', 'string',
'uint256'], abiEncodedFunctionParams)
            functionParamsObj = {
                functionSignature: completeLeadLegFunctionSignature,
                tradeId: functionParams[0],
                sender: functionParams[1],
                receiver: functionParams[2],
                amount: parseInt(functionParams[3]._hex, 16)
            }
        } else
if(decodedCrossBlockchainCallExecutedObj.functionCallData.startsWith(requestFollowLegF
unctionSelector)){
            const abiEncodedFunctionParams =
'0x'+decodedCrossBlockchainCallExecutedObj.functionCallData.replace(requestFollowLegFu
nctionSelector, '')
            const functionParams = abiCoder.decode(['string', 'string', 'string',
'address', 'uint256', 'uint256'], abiEncodedFunctionParams)
            functionParamsObj = {
                functionSignature: requestFollowLegFunctionSignature,
                tradeId: functionParams[0],
                sender: functionParams[1],
                receiver: functionParams[2],
                destinationContract: functionParams[3],
                destinationBlockchainId: parseInt(functionParams[4]._hex, 16),
                amount: parseInt(functionParams[5]._hex, 16)
            }
        } else
if(decodedCrossBlockchainCallExecutedObj.functionCallData.startsWith(cancelLeadLegFunc
tionSelector)){
            const abiEncodedFunctionParams =
'0x'+decodedCrossBlockchainCallExecutedObj.functionCallData.replace(cancelLeadLegFunct
ionSelector, '')
            const functionParams = abiCoder.decode(['string','string','string'],
```

```
abiEncodedFunctionParams)
            functionParamsObj = {
                functionSignature: cancelLeadLegFunctionSignature,
                tradeId: functionParams[0]
            }
        }
        const crossBlockchainCallExecutedObj = {
            destinationBlockchainId:
parseInt(decodedCrossBlockchainCallExecutedObj.destinationBlockchainId._hex, 16),
            contractAddress: decodedCrossBlockchainCallExecutedObj.contract,
            functionCallData: functionParamsObj
        }
        return crossBlockchainCallExecutedObj
    }
```

The encodedInfo data structure containing requestFollowLeg function parameters is decoded as follows:

Example encodedInfo with requestFollowLeg

```
    encodedInfo =
"0x0000000000000000000000000000000000000000000000000000000000000001000000000000000000000
0000000000000000000000000000000000000000000007a752c4d100a96be23f60f072fed1f33da2890fbe
45eb15af52750faa2772a9200000000000000000000000000000000000000000000000000000000000008
00000000000000000000000000000000000000000000000000000000000036bf903680183014b9fb9010
0000000000000000000000000000000000800000000000000000000001000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000001000000000000000000000000000000000000000000000000
0000000000000000000040000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000100000000000000000000000000000000000
0000000000000000000000000000000000400000000000000000000000000000000000000000000000000f90
25df9025a94ece944a482d7f27aa2f0921b99fd071e9d4d6718e1a07a752c4d100a96be23f60f072fed1f3
3da2890fbe45eb15af52750faa2772a92b9022000000000000000000000000000000000000000000000000
0000000000000000200000000000000000000000000e442fbf4fdc4ad1e7d5975c59a9bb17734407ac500000
00000000000000000000000000000000000000000000000000000000000060000000000000000000000000
0000000000000000000000000000000000184c6755b7c00000000000000000000000000000000000000000
000000000000000000000c0000000000000000000000000000000000000000000000000000000000000010
0000000000000000000000000000000000000000000000000000000000001400000000000000000000000
000b580525deb85307c003e1fe1862e8ab4b0f8cede0000000000000000000000000000000000000000000
00000000000000000000100000000000000000000000000000000000000000000000000000000000000640
0000000000000000000000000000000000000000000000000000000000005746573743200000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000b464e555355533030474250000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000b464e474247423030474250000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000"
    crossBlockchainCallExecuted = await decodeInfo(encodedInfo)
    console.log({crossBlockchainCallExecuted})
```

Decoded requestFollowLeg

```
{
  crossBlockchainCallExecuted: {
    destinationBlockchainId: 2,
    contractAddress: '0xe442FBF4Fdc4Ad1E7D5975C59A9Bb17734407aC5',
    functionCallData: {
      functionSignature:
'requestFollowLeg(string,string,string,address,uint256,uint256)',
      tradeId: 'test2',
      sender: 'FNUSUS00GBP',
      receiver: 'FNGBGB00GBP',
      destinationContract: '0xb580525Deb85307c003e1Fe1862e8Ab4b0F8cedE',
      destinationBlockchainId: 1,
      amount: 100
    }
  }
}
```

The encodedInfo data structure containing `completeLeadLeg` function parameters is decoded as follows:

Example encodedInfo with completeLeadLeg

```
    encodedInfo =
"0x0000000000000000000000000000000000000000000000000000000000000002000000000000000000000
0000000000000000000000000000000000000000000007a752c4d100a96be23f60f072fed1f33da2890fbe
45eb15af52750faa2772a9200000000000000000000000000000000000000000000000000000000000000008
000000000000000000000000000000000000000000000000000000000000005e3f905e00183094e46b9010
0000000000000000000000000000000000008004000000000000000000000000000000000000000000000
00000000002000000000000000000000000000000000200000000000000000000000000000000000000000
00000080000000000000000000000000000008000000000000000000000000080000000000000000000000
00000000000000000000400000000000000000000000800000000000000000000000000008000000000020
00000000400000000000000000000000000000000000000001000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000100000000800000000000000f90
4d5f9021a94e34a2afb0a6f7bb2a25151b6bb0bac7bee5ae6bee1a04a088333602fc5fc4f1782002fd83cb
872f641d7564993fcae5a70619d371b83b901e000000000000000000000000000000000000000000000000
00000000000000100000000000000000000000000000000000000000000000000000000000000014000000
00000000000000000000000000000000000000000000000000000000180000000000000000000000000000
000000000000000000000000000000000000001686f6c64457865637574696f6e00000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000063492e040000000
000000000000000000000000000000000000000000000000000000001c000000000000000000000000000000
00000000000000000000000000001348c59000000000000000000000000000000000000000000000000000
00000000000013200000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000b464e555355533030555344000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000b464e47424742303055534400000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000f89994e34a2afb0a6f7bb2a25151b6bb0
bac7bee5ae6bee1a0f0e25c63981e9a617375c8244c8ac144e4e520acc2cb08e2d4c3781b1a02c067b8600
00000000000000000000000000000000000000000000000000000000000002000000000000000000000000
00000000000000000000000000000000000000001320000000000000000000000000000000000000000000
0000000000000000000f9021a9420e827fe967ede24f46f7b9ec9650130cfe4793fe1a07a752c4d100a96b
e23f60f072fed1f33da2890fbe45eb15af52750faa2772a92b901e00000000000000000000000000000000
000000000000000000000000000000001000000000000000000000000570715b0b32a5cda50c52367ae292
9de10fca9450000000000000000000000000000000000000000000000000000000000000006000000000000
000000000000000000000000000000000000000000000000001448903901f0000000000000000000000000
000000000000000000000000000000000000800000000000000000000000000000000000000000000000
000000000000000c00000000000000000000000000000000000000000000000000000000000000100000000
0000000000000000000000000000000000000000000000000000000001000000000000000000000000000
000000000000000000000000000000000000132000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000b464e474
24742303055534400000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000b464e55535553303055534400000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000"
    crossBlockchainCallExecuted = await decodeInfo(encodedInfo)
    console.log({crossBlockchainCallExecuted})
```

Decoded completeLeadLeg

```
{
  crossBlockchainCallExecuted: {
    destinationBlockchainId: 1,
    contractAddress: '0x570715b0b32A5Cda50c52367Ae2929de10Fca945',
    functionCallData: {
      functionSignature: 'completeLeadLeg(string,string,string,uint256)',
      tradeId: '2',
      sender: 'FNGBGB00USD',
      receiver: 'FNUSUS00USD',
      amount: 1
    }
  }
}
```

20.2. Ethereum Proof Generation

The eth-proof 2.1.6 merkle-patricia-proof module is used to generate the crosschain transaction proof, which is then submitted to the blockchain.

The transaction proof is requested using a function from the module's `GetProof` class called `receiptProof`.
It takes in the hex encoded string of the transaction hash and returns an object containing the proof information shown below:

Structure returned from GetProof

```
{
  "receiptProof": {
    "0x49d3ded1d37f16121d71cd8c6a8e0ed6b9ebcfa17fe1a69eaa3d493760e402cc": [
      {
        "type": "Buffer",
        "data": [
          180, 194, 241, 92, 158, 136, 114, 123, 46, 183, 165, 127, 86, 27, 89,
          254, 96, 20, 227, 255, 76, 43, 196, 14, 154, 137, 212, 204, 147, 175,
          165, 108
        ]
      },
      {
        "type": "Buffer",
        "data": []
      },
      { "type": "Buffer", "data": [] },
      { "type": "Buffer", "data": [] },
      { "type": "Buffer", "data": [] },
      { "type": "Buffer", "data": [] },
      { "type": "Buffer", "data": [] },
      { "type": "Buffer", "data": [] },
      {
```

```
          "type": "Buffer",
          "data": [
            239, 162, 83, 74, 37, 234, 9, 168, 108, 114, 213, 159, 81, 119, 169,
            33, 150, 189, 71, 175, 97, 76, 65, 18, 61, 125, 231, 236, 123, 121,
            216, 9
          ]
        },
        {
          "type": "Buffer",
          "data": []
        },
        { "type": "Buffer", "data": [] },
        { "type": "Buffer", "data": [] },
        { "type": "Buffer", "data": [] },
        { "type": "Buffer", "data": [] },
        { "type": "Buffer", "data": [] },
        { "type": "Buffer", "data": [] },
        {
          "type": "Buffer",
          "data": []
        }
      ],
      "0xefa2534a25ea09a86c72d59f5177a92196bd47af614c41123d7de7ec7b79d809": [
        { "type": "Buffer", "data": [48] },
        {
          "type": "Buffer",
          "data": [
            249, 1, 164, 1, 130, 205, 58, 185, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            64, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 248, 155, 248, 153,
            148, 220, 57, 15, 110, 127, 199, 60, 244, 99, 156, 227, 67, 215, 55,
            90, 208, 33, 253, 171, 18, 225, 160, 221, 242, 82, 173, 27, 226, 200,
            155, 105, 194, 176, 104, 252, 55, 141, 170, 149, 43, 167, 241, 99,
            196, 161, 22, 40, 245, 90, 77, 245, 35, 179, 239, 184, 96, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 6, 195, 244, 130, 241, 135, 17, 190, 149, 173,
            241, 6, 175, 162, 92, 209, 56, 151, 251, 231, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 4, 158, 182, 23, 251, 165, 153, 227, 212, 85, 218, 112,
            198, 115, 10, 188, 140, 196, 34, 29, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1
          ]
        }
      ]
```

```
  }
}
```

The object is then RLP encoded and HP encoded to create a data structure with the following fields:

1. `path`: The path in the tree leading to value
2. `rlpEncodedReceipt`: The value to be checked to be contained in the nodes
3. `witness`: The rlp encoded stack of nodes of the Merkle Patricia tree

Structure of bundledReceiptProof


```
"bundledReceiptProof": {
    "path": "0x1",
    "rlpEncodedReceipt":
"0xf901a40182cd3ab90100000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00800000000000000000000000000000000400000000000000000000000000000000000000000000000000
00000000000100000000000000000000000000000000000000000000000000000000000000000000000000
00100000000000000000000000000000000000000000000000000000000000000000020000000000000000
00000000000000000000000000040000000000000000000000000000000000000000000000000000000000
0000000000000000000f89bf89994dc390f6e7fc73cf4639ce343d7375ad021fdab12e1a0ddf252ad1be2c
89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3efb86000000000000000000000000006c3f48
2f18711be95adf106afa25cd13897fbe700000000000000000000000049eb617fba599e3d455da70c6730
abc8cc4221d000000000000000000000000000000000000000000000000000000000000001",
    "witness":
"0xf90201f851a0b4c2f15c9e88727b2eb7a57f561b59fe6014e3ff4c2bc40e9a89d4cc93afa56c8080808
0808080a0efa2534a25ea09a86c72d59f5177a92196bd47af614c41123d7de7ec7b79d80980808080808
080f901ab30b901a7f901a40182cd3ab901000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000008000000000000000000000000000000000400000000000000000000000000000000000
00000000000000000000000001000000000000000000000000000000000000000000000000000000000000
00000000000000001000000000000000000000000000000000000000000000000000000000000000000020
00000000000000000000000000000000000000000400000000000000000000000000000000000000000000
000000000000000000000000000000000f89bf89994dc390f6e7fc73cf4639ce343d7375ad021fdab12e1a
0ddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3efb86000000000000000000
000000006c3f482f18711be95adf106afa25cd13897fbe700000000000000000000000049eb617fba599e
3d455da70c6730abc8cc4221d0000000000000000000000000000000000000000000000000000000000000
001"
  }
```


20.3. Corda Transaction Serialization

Corda uses an extended form of the Advanced Message Queuing Protocol (AMQP) 1.0 serialization format to transform transaction objects into byte arrays which can be stored in the Corda ledger.

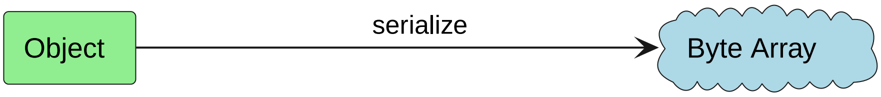

Figure 35. Serializing a Corda Object

The specific library used is called Qpid Proton, which is an AMQP messaging toolkit written in Java.

The serialized byte array is represented in a strongly typed form, which is realised through the concepts of a Payload, Schema and Transformation Schema.
This ensures that during the process of deserialization, the original transaction can be reconstructed as the object's Java classes, even if they are not already in the classpath.

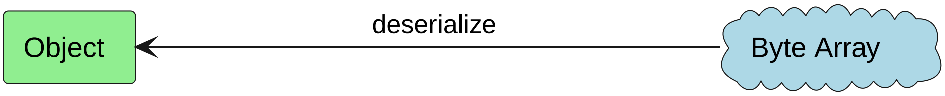

Figure 36. Corda Deserialization

Serializing the object using Corda's extended AMQP/1.0 serialisation format is achieved in two phases.
In the first phase, the object is mapped to a graph data structure called a proton graph.
The second phase then involves converting the proton graph into a byte array, which can be stored on-chain.

Creating the proton graph involves a Corda construct called Described-Descriptor-Description, which builds the graph from different node types based on the data that the node represents.
The Described Descriptor Description Construct diagram illustrates the main components of the Described-Descriptor-Description construct.
A `Described` node is connected to two children nodes, which are the object `Descriptor` and the object `Description`.

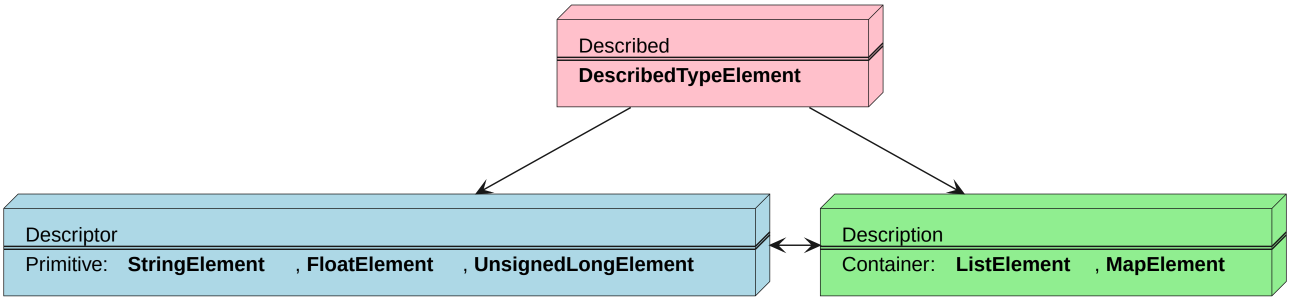

Figure 37. Described Descriptor Description Construct

The `Descriptor` nodes in a proton graph can represent primitive types, for example, string, float, unsigned long, etc.

A `Descriptor` node can also be a symbol node, which is necessary for serializing user-defined types that need to be referenced using a unique identifier called a `Fingerprint`.

`Description` nodes represent containers, which can be a list or a map.

The root node, or entry point, of the proton graph is called an Envelope.
The AMQP Mapping a Class to a Proton Graph diagram shows a Described-Descriptor-Description

construct where the Description node is a list of the following three components:

- Payload
- Schema
- Transformation Schema

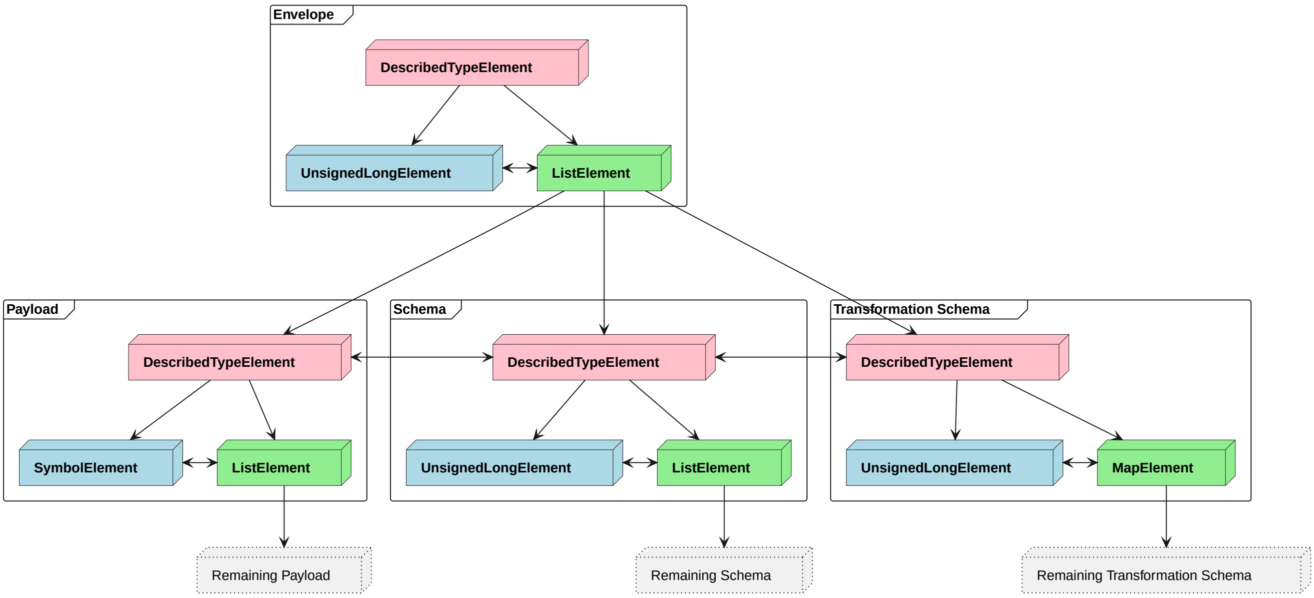

Figure 38. AMQP Mapping a Class to a Proton Graph

20.3.1. Payload

The `Payload` contains the actual data.
When the data being serialized is a primitive type, for example a string, the `Payload` would consist only of a node of type `StringElement`.

In the case that the `Payload` is a class object with input parameters, then the `Payload` would be a Described-Descriptor-Description construct as shown in the AMQP Mapping a Class to a Proton Graph Payload diagram.
A user-defined class is a custom data type which means that the payload will include the symbol node containing the `Fingerprint` to uniquely identify the user-defined class corresponding to a class in the classpath.
The node next to the symbol node is a list, which contains the values assigned to the class parameters.
If the inputs are an integer and a string, then connected nodes will be a `StringElement` and `IntegerElement` as shown in the figure.

A description of the class object is provided in the `Schema` discussed in the next section.

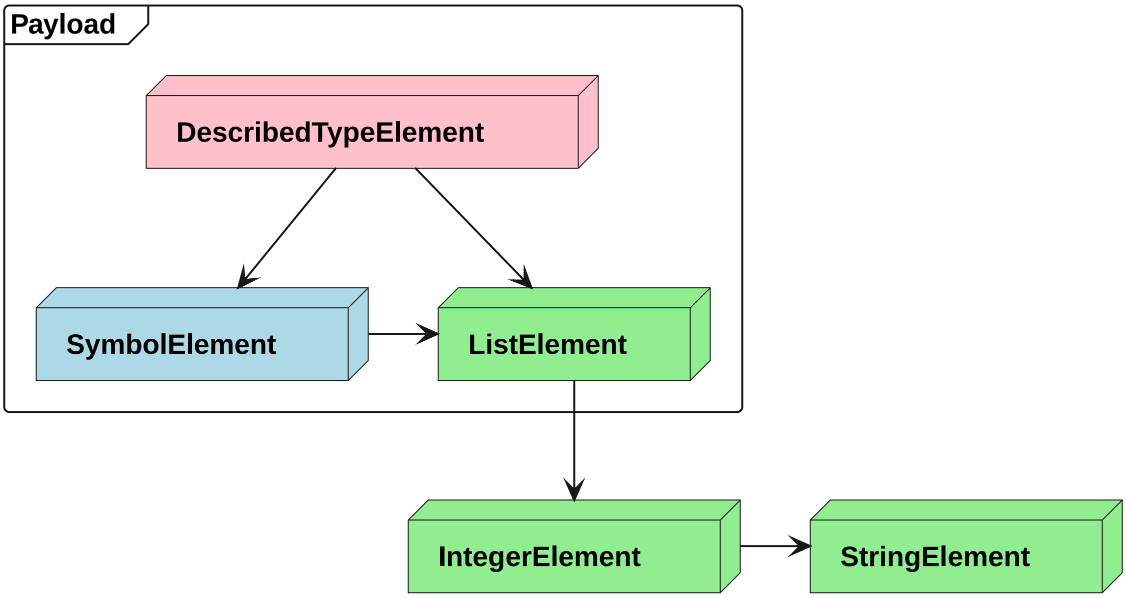

Figure 39. AMQP Mapping a Class to a Proton Graph Payload

20.3.2. Schema

The `Schema` describes non-primitive datatypes, such as class objects, occurring in the `Payload`.
A class is represented in a Corda `CompositeType`, which is also defined by the Described-Descriptor-Description construct.
The `CompositeType` has five properties, namely, name, label, provides, descriptor and fields.
The descriptor field refers to the object `Descriptor` and contains the `Fingerprint` of the class.
The fields property is a list of `Field` types that, where each Field represents a property of the class.

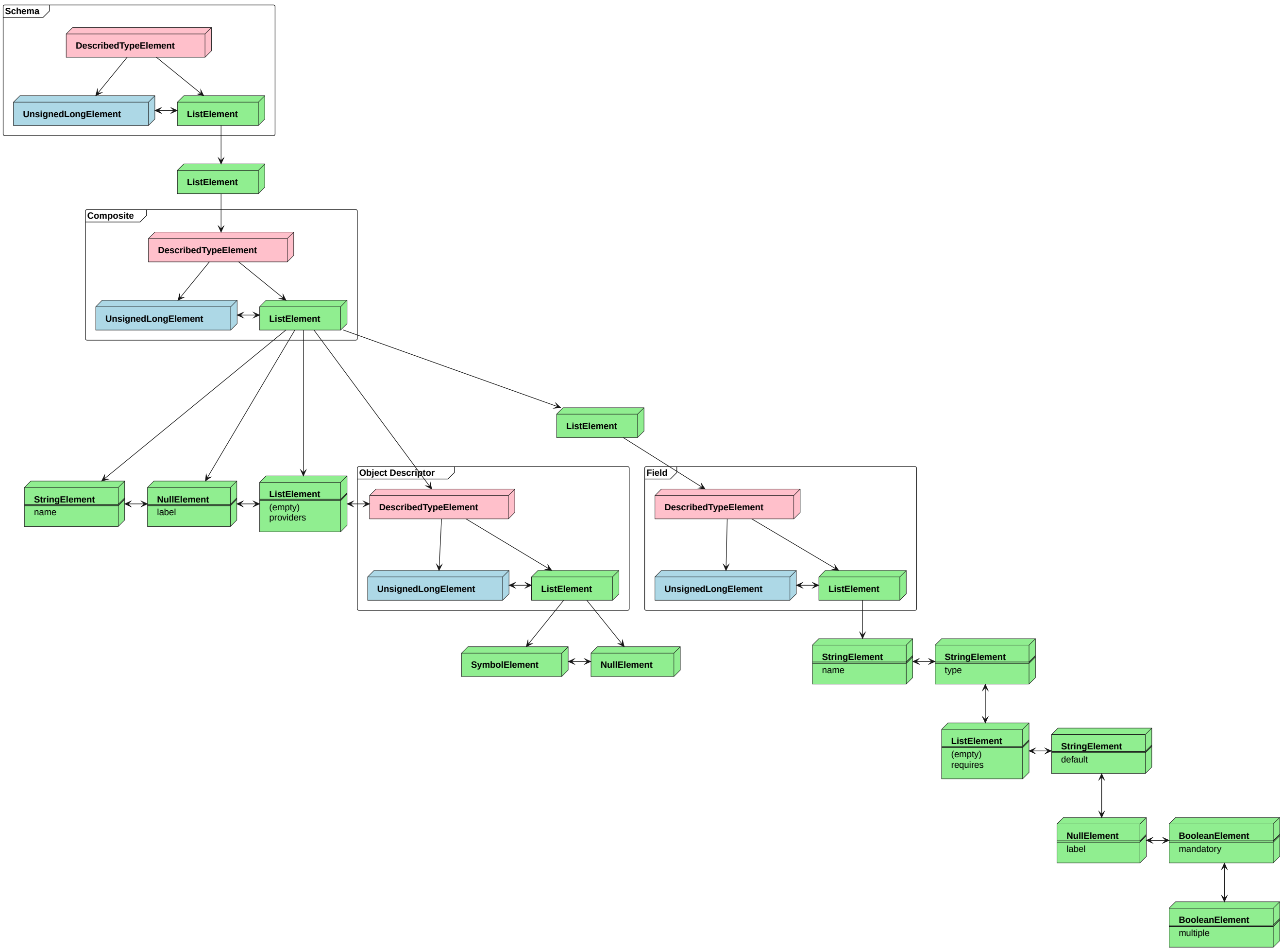

Figure 40. AMQP Mapping a Class to a Proton Graph Schema Diagram

20.3.3. Transformation Schema

The Transformation Schema provides a description of the evolution of types.
If there are no changes to the types, then the Transformation Schema will be empty.

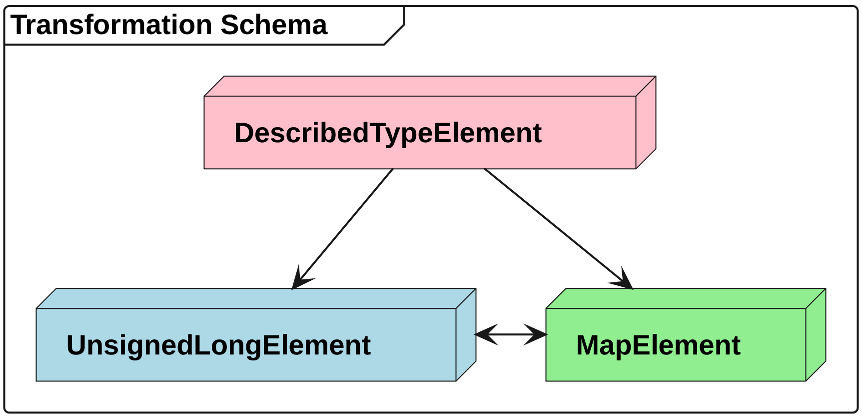

Figure 41. AMQP Mapping a Class to a Proton Graph Transformation Schema

20.4. Corda Serialisation Format

An example of a Java Primitive String data type which has been serialized into a byte array is shown below:

Serialized string


```
636F7264610100000080C562000000000001C09203
A174417070726F7665204E4557207374617465 2077
69746820747261646520696420313233342066726F
6D207061727479204F3D416C69636520436F72702C
204C3D4D61647269642C20433D455320746F20636F
756E746572706172747920 4F3D426F6220506C632C
204C3D526F6D652C20433D49540080C56200000000
0002C00201450080C562000000000009C10100
```


This serialized byte array (blob) starts with an eight-byte preamble which is `CORDA100`, encoded as 636F726461010000 and is followed by the actual encoded data.

The proton graph of the String is shown in the AMQP Mapping an Object to a Proton Graph diagram.

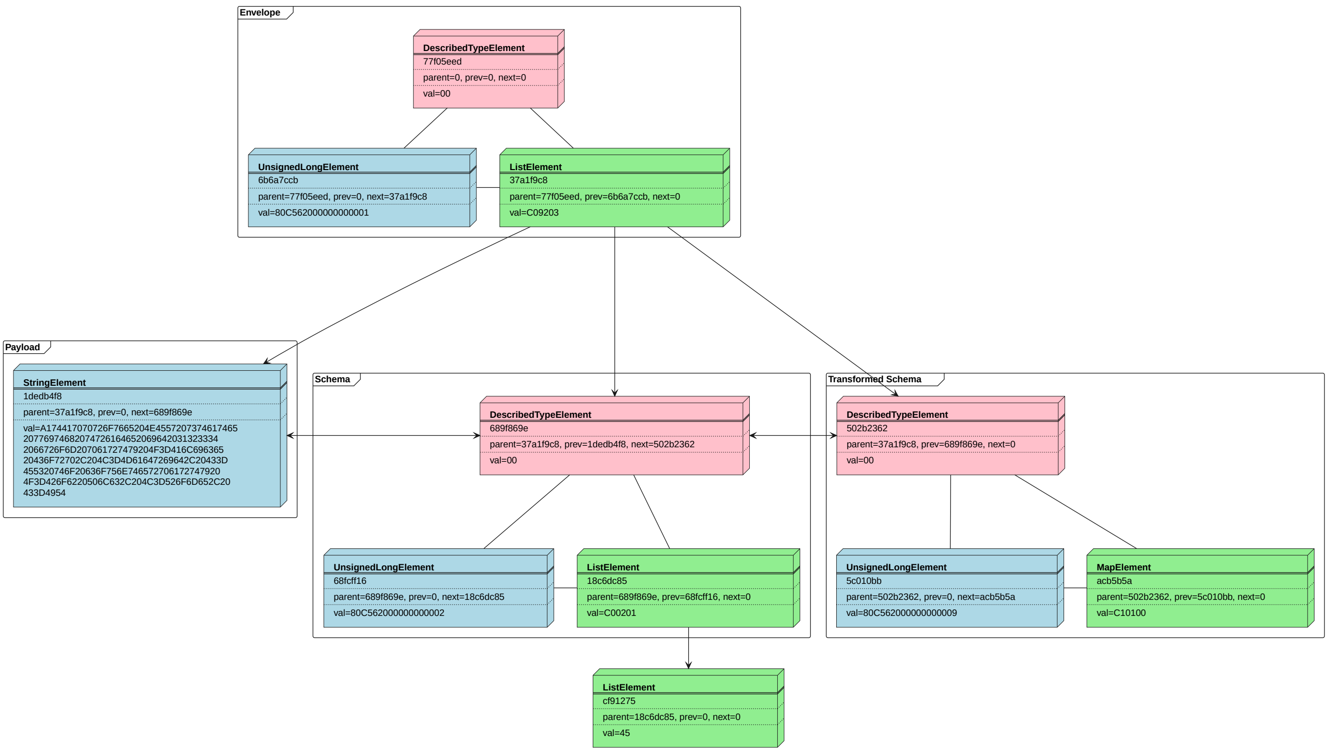

Figure 42. AMQP Mapping an Object to a Proton Graph

20.5. Corda Transaction Data Structure

A signed Corda transaction is created from the `net.corda.core.transactions.SignedTransaction` class defined in the Corda codebase.
Internal to the singed transaction class is the `net.corda.core.transactions.WireTransaction` class, which consists of the wire transaction.
The wire transaction is the part of the signed transaction that contains the data required to produce the transaction's Merkle tree.
The following is an example of the contents of a deserialized Corda transaction in json format.
The `wire` field refers to the wire transaction, which contains, for example, the `privacySalt` needed to build the Merkel tree.

Structure of Corda transaction


```
{
  "wire": {
    "digestService": {
      "hashAlgorithm": "SHA-256"
    },
    "id": "8476F686B4E12DDB34C6F451CDCC67DFDDE12CA1F0437690CBA56A8E01A74468",
    "notary": "O=Notary Service, L=Zurich, C=CH",
    "inputs": [],
    "outputs": [
      {
        "data": {
          "@class": "net.corda.finance.contracts.asset.Cash$State",
          "amount": "1000.00 GBP issued by O=BankOfCorda, L=London, C=GB[313233]",
          "owner": "O=BankOfCorda, L=London, C=GB"
        },
        "contract": "net.corda.finance.contracts.asset.Cash",
        "notary": "O=Notary Service, L=Zurich, C=CH",
        "encumbrance": null,
        "constraint": {
          "@class": "net.corda.core.contracts.WhitelistedByZoneAttachmentConstraint"
        }
      }
    ],
    "commands": [
      {
        "value": {
          "@class": "net.corda.finance.contracts.asset.Cash$Commands$Issue"
        },
        "signers": [
          "GfHq2tTVk9z4eXgyLMjZLHLQqqGxUttzF87kiBSFqaNZZXEMErBovoJ13KAy"
        ]
      }
    ],
    "timeWindow": null,
    "attachments": [
      "AB8FB243626B949C9EE904DD7052ECD78B371641BE2148DAC26E477F152218D4"
    ],
    "references": [],
    "privacySalt": "507C152BEBE450C13C2B6CAA3AD9E6756A0EB2EC11062E53A49CF23AF6D8AA36",
    "networkParametersHash": null
  },
  "signatures": [
    {
      "bytes":
"dLbR4EhZgdwDcZPZKqQdQJ0e/jht4uGwpUUr4h5PTTRwoGxBZObxLK/M9EN1b4zN1csloFYR69flo5bPDOF8B
w==",
      "by": "GfHq2tTVk9z4eXgyLMjZLHLQqqGxUttzF87kiBSFqaNZZXEMErBovoJ13KAy",
      "signatureMetadata": {
        "platformVersion": 4,
```

```
          "scheme": "EDDSA_ED25519_SHA512"
        },
        "partialMerkleTree": null
      }
    ]
  }
```