L2X Protocol v 0.6

A high performance crypto exchange protocol

Mathis Antony, Philippe Camacho, Antoine Cote,
David Leung and Lionello Lunesu

Enuma Technologies

OAX Foundation

18th June 2019
rev `33c7d42`

Contents

1 Introduction

1.1 Motivation

Since the MtGox collapse [1] in February 2014, users have learned the hard way how dangerous it can be to trade cryptocurrencies on centralized exchanges. Nonetheless most of the volume traded today comes from similar platforms and many other exchanges have been hit by cyberattacks [2].

Recently a lot of activity around decentralized exchanges such as *0x* [14] and layer 2 techniques[3] have brought a light of hope on the current state of affairs. Indeed with these approaches, users can now exchange tokens without having to delegate the custody of their funds to a trusted third party. However a lot of work needs to be done in order to achieve the same level of usability as for centralized exchanges.

Our solution aims to get the best of both worlds: at a high level the L2X protocol operates similarly to a standard exchange by collecting and processing all the orders in a single place. However our solution is *trustless*, that is the user always keeps control over their funds even though the exchange may be compromised.

The goal of this document is to provide a technical documentation of the L2X protocol for its study and implementation.

1.2 High level goals

1.2.1 Trustless

Users do not need to trust the exchange to maintain balances or execute trading instructions correctly. Users can rectify protocol violations by lodging disputes on the blockchain.

1.2.2 Non-Custodial

Users are the sole custodians of their accounts. The role of an exchange is to facilitate trading among users against authorized instructions.

1.2.3 Non-Collateralized

The exchange operator does not need to lock expensive collateral to guarantee operational integrity.

1.2.4 Performant

Performance characteristics of the exchange are in line with existing centralized exchanges.

1.2.5 Scalable

Nodes can be added to increase system capacity and throughput.

1.3 Related work

In order to exchange crypto-assets without relying on a trusted intermediary, researchers and developers started to explore solutions which involve an off-chain trading engine combined with on-chain settlement: DEXes (e.g.[13, 14]) provide much better security guarantees than traditional exchanges but are limited by the low throughput of existing blockchains. Our solution is built on an

[1] `https://en.wikipedia.org/wiki/Mt._Gox`

[2] `https://cointelegraph.com/news/crypto-exchange-hacks-in-review-proactive-steps-and-expert-advice`

[3] `https://lightning.network/`,`https://raiden.network/`

Work →	0x [14]	Loopring [13]	TEX [9]	This work
Decentralized	✓	✓	×	×
Scalable	×	×	✓	✓
Cross-chain	×	×	×	×
Collateral free	✓	✓	✓(**)	✓(**)
Partial orders	×	×	×	✓
Offline settlement (*)	✓	✓	×	✓
FRP (***)	×	×	✓	✓

Table 1: **Trustless exchanges comparison.** (*) The client can go offline after placing their order. (**) For instant trade finality collateral is needed. (***) *Front Running Prevention.*

extension of NOCUST [8] which while centralized, remains trustless as users always are in control of their funds. TEX [9] also builds on NOCUST and provides enhanced functionalities such as *Front Running Prevention*. To the best of our knowledge our solution is the only one to support partial orders. In order to trade assets that have no direct representation as ERC20 tokens, one may use cross-chain swaps solutions such as Arwen [3] or XClaim[15].

1.4 Summary

This section provides a very high level summary of the protocol described in this document. The purpose of the protocol is to enable a set of clients to perform fast, trustless, non-custodial, off-chain exchange of digital assets. The protocol centers around the interaction of four kinds of entities: the clients, the exchange, the operator, and the mediator smart contract.

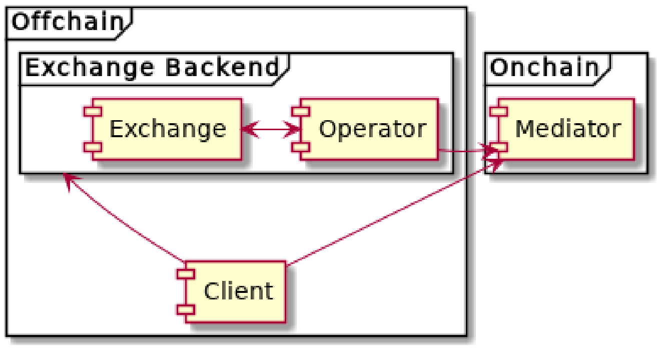

Figure 1: **Architecture overview**

The entities interact with each other around a *commit chain*. To maintain the commit chain, the operator periodically (once per *round*) publishes a cryptographic commitment to the blockchain via the mediator. The commitment is the root of an augmented Merkle tree over the balances of the clients. The tree nodes are augmented with the sum of the balances of their children.

The clients query the operator for the Merkle proof and use that to verify their balance in the Merkle tree. If this verification fails, the clients lodge a dispute on the mediator by providing the Merkle proof from the previous round and optionally other information that affects their change in balance.

After a dispute is opened, the operator is given a time window during which it can attempt to close the dispute by supplying additional evidence to the mediator. If the dispute is still open at the end of the time window, the system is halted and users can withdraw their funds. An honest operator can always close these disputes, but a dishonest operator cannot.

1.4.1 Participants

Clients: Clients who wish to trade digital assets (in the form of blockchain tokens) must be in possession of a private key. Once they have *joined* (2.4.1) an exchange, and *deposited* (2.4.2) some tokens into the mediator smart contract, they may create trading orders for the exchange by signing and submitting trade approvals (2.4.4) to the exchange for execution. The clients are the ultimate custodians of their tokens in the protocol.

Exchange: An exchange facilitates trading of digital assets by maintaining markets and order books, and providing order matching services. This document does not provide specification for the exchange.

Operator: The operator *operates* the protocol. It maintains the balances (2.3.4) of client accounts and executes matched orders received from the exchange. It periodically publishes a cryptographic commitment (2.3.2) of the global balances to the blockchain via the mediator. To support trustless operations, it is also responsible for

- proving its correct operation (2.3.3),
- closing disputes (2.3.6) opened by clients against operational integrity,
- working with the mediator to prevent malicious or misbehaving clients from double-spending (2.3.5) their digital assets.

Mediator: The mediator is responsible for enforcing the overall correctness and key characteristics of the protocol. It ensures that the operator — and by extension, the exchange — operates trustlessly. It does so by mediating the challenge-response interactions between clients and the operator. To protect client funds, the mediator can halt a malicious or defective operator, preventing it from continuing to operate.

1.4.2 Implementation

This protocol has been implemented using the Ethereum blockchain. It allows users to trade ERC20[4] tokens. Off-chain code has been implemented with Typescript and the smart contract has been written in Solidity.

1.5 Security

Analyzing the security of our protocol is a challenging task due in particular to the following factors:

- Layer 2 techniques are quite novel and more work on understanding their security is to be expected.
- The interactions between the layer 2 components (in particular the mediator smart contract) and the core logic of the application (exchange) may result in unexpected behavior. For example, the representation and handling of integers differ between the layer 1 and the layer 2 as different languages (Typescript v/s Solidity) are used.

[4]`https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`

- While we rely on relatively simple cryptographic primitives (digital signatures and Merkle trees), their instantiations need to be carefully thought through: for example Kexin etal.'s attack[6] on a proof of liabilities scheme remembers us that apparently simple protocols can be flawed.

In order to address these challenges we have used the following approach:

- **Be as explicit as possible:** our protocol is described as a set of Python scripts in order to remain concise, yet precise, and thus support a serious study of our solution.
- **Constantly look for flaws** and learn from the exercise: a detailed list of attacks that have guided our design and understanding of the problem is available in Section A.1.
- **Back your claims:** while further work is planned in order to systematize and deepen the security analysis of our protocol, we have set the foundations of a rigorous definition of our security goals and argumentation that these goals are met. Section A.2 provides a high level and preliminary analysis of the properties our protocol is expected to fulfill. These properties are another evidence of the complexity and uncommon behavior the kind of system we are building possess: The distributed / non-deterministic nature of our protocol makes it challenging even to understand in a precise manner what the guarantees of the protocol should be.

  One of the key component of our solution is a Merkle tree that is required in order to ensure the exchange/operator is always accountable for the funds it owes to its clients. In Section B we provide a formal definition and security proof for our Merkle tree instantiation.

1.6 Acknowledgements

We would like to thank Ron van der Meyden for his detailed review and in particular for pointing out a gap in the formulation of property 2.

2 L2X protocol

2.1 Preliminaries

2.1.1 Glossary

2.1.1.1 Approval

An approval represents the intention of the client to exchange tokens. It is the equivalent of an order for classical exchanges.

2.1.1.2 Client

The clients are the participants who wish to trade tokens. They will interact with the other participants (operator, mediator, exchange) in order to perform their trading activities.

2.1.1.3 Deposit

The clients deposit tokens through the mediator in order to trade these tokens on the exchange.

2.1.1.4 Exchange

The exchange is the component of the system that receives approvals (orders) and matches these with orders in its order book to create trades.

2.1.1.5 Fee

An exchange might charge fees to support its operation. Fees require a prior deposit and approval from the client, granting the exchange permission to transfer the fee from the client's balance to itself. As fees amounts tend to be small, fees need not be handled in a trustless manner. We do not describe in this document how fees are implemented.

2.1.1.6 Fill

If an approval is executed by the exchange the operator will produce one or more fills. Fills are evidence from the operator of trading activity. They are used by the client to open disputes on chain in the case of disagreement with the operator.

2.1.1.7 `HALTED` mode

The L2X protocol is designed to be resilient against a malicious operator. In practice if the operator fails to fulfill its duties, the mediator smart contract will enter a special state or mode called `HALTED`. In this state most operations (deposits, withdrawal initiations, ...) are deactivated but users can recover their funds.

2.1.1.8 Mediator

The mediator is the smart contract through which the operator and the clients interact especially in case of disagreement. The mediator holds the funds of the clients.

2.1.1.9 Operator

The operator is a participant of the protocol whose purpose is to maintain the off-chain balance ledger in accordance with the trades generated by the exchange. It is also responsible to provide evidences to the mediator when canceling illegitimate withdrawals or closing disputes.

2.1.1.10 Recovery of funds

When the mediator enters the `HALTED` mode, the clients are able to recover their funds obtained from past trading activity or deposits.

2.1.1.11 Rounds and quarters

The L2X protocol uses the blockchain in order to synchronize the participants. In practice time is measured in rounds (fixed number of blocks) and each round is divided into four quarters.

2.1.1.12 Withdrawal

A withdrawal is an operation performed by the client to transfer assets — previously deposited into the mediator *or* obtained via off-chain trading — back to the clients' wallet. Withdrawals are made through the mediator.

2.1.2 Types and Data Structures

We use some Typescript-like syntax[5] to describe the types and data structures used in our protocol. Note that square brackets denote arrays: `string[]` is an array of strings.

[5]See `https://www.typescriptlang.org/docs/handbook/basic-types.html` and `https://www.typescriptlang.org/docs/handbook/interfaces.html`.

2.1.2.1 Basic types

Listing 1 Basic types

```
type Digest = string
type Address = string
type Amount = Integer
type Signature = string
type Intent = "buyAll" | "sellAll"
type Round = Integer
type Quarter = 0 | 1 | 2 | 3
type Id = string
```

The basic types we use are listed in Listing 1.

2.1.2.2 Approval

Listing 2 Approval

```
interface Approval {
  round: Round
  id: Id
  buy_asset: Address
  buy_amount: Amount
  sell_asset: Address
  sell_amount: Amount
  owner: Address
  intent: Intent
}
```

Approvals are created by, and signed by the client. An approval captures the intention to swap certain amounts of tokens. They take the role of orders in the traditional exchange setting. Notable differences are

- the use of buy and sell quantities instead of a price,
- the use of an "intent" field in favor of an order "side" (e. g. "buy" or "sell").

As a result, the smart contract does not need to be aware of the trading markets the exchange chooses to support. Absent a market, there are four types of orders we can create for a given pair of assets A, B:

side	buy asset	sell asset
buy	A	B
buy	B	A
sell	A	B
sell	B	A

A buy ETH/USD order is not the same as a sell USD/ETH order because of the restriction this imposes on the amounts to be filled.

- For an order to buy 10 ETH @ 100 USD the order is filled when the issuer has bought 10 ETH. The issuer will pay at most 1000 USD for it but may pay less.

- For an order to sell 1000 USD @ 0.01 ETH the order is filled when the issuer has sold 1000 USD. The issuer will receive at least 10 ETH for it but may receive more.

We therefore use the “intent” property to describe that the issuer would like to either fill until the sell side is satisfied (“sellAll”) or fill until the buy side is satisfied (“buyAll”). It is desirable for the mediator to support all of these orders that the exchange can decide which markets to offer without having to make modifications to the smart contract.

For a more detailed description we refer to section 7.1 of the *Loopring protocol* paper [13] where the variable *BuyNoMoreThanAmountB* is created to support the four types of orders.

2.1.2.2.1 Approval (or Order) Lifecycle An approval received by the operator is valid for the current round only. If the operator fills it in another round the mediator may be halted via disputes. The layer 2 solution presented in this work does not implement trustless cancellation or any other forms of order expiry, such as *good-till-canceled*.

Because the exchange has full autonomy over the order matching they may choose to implement cooperative order cancellation themselves. Note that a malicious operator may refuse to cancel approvals at any time. Furthermore, an approval could still be filled by a malicious operator after cancellation until the round is over.

To address the problem with orders expiring at the end of the round the client software could for example re-create approvals at the beginning of the round.

2.1.2.3 Fill

Listing 3 Fill

```
interface Fill {
  round: Round
  fillId: Id
  approvalId: Id
  bought_asset: Address
  bought_amount: Amount
  sold_asset: Address
  sold_amount: Amount
  client_address: Address
}
```

Fills are created by, and signed by the operator. Note that a fill represents only a client’s side of the trade and thus must contain the corresponding client’s address. Moreover a fill must be related to a specific approval through the `approvalId` field.

When an approval is matched by the trading engine, the output is a set of fills[6] At a high level, these fills must be such that:

- In the case the approval intent is *buyAll*:
  - The sum of the buy amount of the fills must be equal to the buy amount of the approval.
  - The sum of the sell amount of the fills must be lower or equal to the sell amount of the approval.
- In the case the approval intent is *sellAll*:
  - The sum of the sell amount of the fills must be equal to the sell amount of the approval.

[6]Recall that our protocol supports partial orders.

  - The sum of the buy amount of the fills must be greater or equal to the buy amount of the approval.

- In both cases the buy (resp. sell) asset of each fill must match the buy (resp. sell) asset of the approval.

These rules ensure that the client obtains at least what they expect from the trade. In Listing 4, we can see an example where Alice wants to buy 10 OAX tokens for at most 100 WETH tokens. After the match she will receive her 10 OAX tokens and will release 90 WETH tokens.

Listing 4 Example of a filled approval

```
A = {
  round: 1
  id: 1234
  buy_asset: OAX
  buy_amount: 10
  sell_asset: WETH
  sell_amount: 100
  owner: Alice
  intent: 'buyAll'
}

F1 = {
  round: 1
  fillId: 789
  approvalId: 1234
  bought_asset: OAX
  bought_amount: 5
  sold_asset: WETH
  sold_amount: 50
  client_address: Alice
}

F2 = {
  round: 1
  fillId: 10009
  approvalId: 1234
  bought_asset: OAX
  bought_amount: 5
  sold_asset: WETH
  sold_amount: 40
  client_address: Alice
}
```

2.1.2.4 Proof

Listing 5 Proof

```
interface Proof {
    client_opening_balance: Amount
    client_address: Address
    hashes: Digest[]
    sums: Amount[]
    token_address: Address
    height: Number
    width: Number
}
```

A proof enables a client to verify their balance for a specific token periodically based on some public information. In practice the proof contains the necessary information to reconstruct a Merkle root from the client's balance and address. The fields `height` and `width` are used for padding techniques described in Section B.2.

2.1.2.5 Withdrawal request

Listing 6 Withdrawal request

```
interface WithdrawalRequest {
  round: Round
  amount: Amount
  token_address: Address
}
```

At some point a client will withdraw their off-chain balance and store the corresponding tokens on their personal wallet. This is done by sending a `WithdrawalRequest` message to the mediator.

Each client is allowed only one withdrawal request at a time. Each withdrawal request contains the token address, the amount and the round when the request was made.

2.1.2.6 Messages

Messages exchanged between the client and operator follow the format given in Listing 7. When a client wants to join they will send a `GetAuthorizationMessage` message to the operator who will reply with an `AuthorizationMessage` message. Approvals and fills are described by the `ApprovalMessage` and `FillMessage` message respectively. Note that these messages contain the signature of the client (for an approval) and the operator (for a fill).

Periodically each client will ask for a proofs array to the operator by sending the `GetProofsMessage`. The operator will then provide the proofs contained in the `ProofsMessage` message.

Listing 7 Message

```
interface GetAuthorizationMessage {
  msg_type: "get_authorization"
  content: {
      client_address: Address
      signature: Signature
  }
}

interface AuthorizationMessage {
    msg_type: "authorization"
    content: {
      client_address: Address
      signature: Signature
    }
}

interface ApprovalMessage {
  msg_type: "approval"
  content: {
    approval: Approval,
    signature: Signature
  }
}

interface FillMessage {
  msg_type: "fill"
  content: {
    fill: Fill,
    signature: Signature
  }
}

interface GetProofsMessage {
  msg_type: "get_proofs"
  content: {
    round: Round
    client_address: Address
  }
}

interface ProofsMessage {
  msg_type: "proofs"
  content: Proof[]
}
```

2.1.2.7 Merkle tree

Given that it would be unpractical to store all the balances of the client inside the mediator smart contract, we need to rely on a data structure that enables the participants to: (1) provide a short representation for the balances of each client as well as the total sum of these balances, (2) provide an efficient way to verify the value of each balance, (3) ensure that the balance of every client is considered in the total sum.

The technique to satisfy these requirements is called *proofs of liabilities* (see [2], Section 2.2). Proof of liabilities extend Merkle trees to track all the balances (placed at the leaves) and compute the total sum (obtained at the root).

The tree is built following this rule: the values stored in each node must be positive and equal to the sum of the values inside the left child and right child. This way we can guarantee that the balances of all clients are taken into account, and that the total sum representing the amount owed by the operator is greater or equal to the sum of the balances for all clients.

We expand on the security of this data structure, including padding techniques, in Section B. The algorithms related to this data structure are described in Listings 8 and 9. The programming language used for this listing and the others is *Python*[7].

Listing 8 Merkle tree functions 1/2

```
LEFT = 0
RIGHT = 1
SEPARATOR = "||"


def pack(*content):
    """
    Packs the content to be hashed
    :param content ( string[] ): a list of content to be concatenated and separated by SEPARATOR
    :return ( string ): the concatenated content
    """
    return SEPARATOR.join(str(c) for c in content)


def build_root(proof):
    """
    @param proof ( Proof ): merkle proof from which we want to recompute the root.
    @return ( Digest ): root of the original Merkle tree.
    """

    # Here we assume the nodes are ordered by increasing height (i.e. the first
    # node has height 1, the second node has height 2 etc...)
    nodes = get_nodes_list(proof)
    nodes_in_reverse_order = reversed(nodes)
    leaf_right , leaf_left , *internal_nodes = nodes_in_reverse_order

    amount = leaf_left.amount + leaf_right.amount

    hash_value = Hash(pack(leaf_left.value, leaf_right .value, amount))

    for node in internal_nodes:

        amount += node.amount

        if node.location == LEFT:
            hash_content = pack(node.value, hash_value)
        else:
            hash_content = pack(hash_value, node.value)

        hash_value = Hash(pack(hash_content, amount))

    # Apply padding
    hash_value = Hash(pack(hash_value, proof.height, proof.width))

    return hash_value
```

[7] `https://www.python.org/`. We use the version *3.7.2*.

Listing 9 Merkle tree functions 2/2

```

def validate(proof, root):
    """
    @param proof ( Proof ): merkle proof of a tree T
    @param root ( Digest ): root published in smart contract V
    @rtype: boolean
    @return: true if the verification passes, false otherwise
    """
    nodes = get_nodes_list(proof)

    # The proof length should be equal to the height of the tree
    # Note that the proof contains two leaves (the leaf to be tested and its sibling)
    if len(nodes) != proof.height:
        return False

    *internal_nodes, leaf_left , leaf_right = nodes

    # Here we assume the nodes are ordered by increasing height (i.e. the first
    # node has height 1, the second node has height 2 etc...)
    # Each node has an attribute that defines whether it is the left (integer 0)
    # or right child (integer 1) of its parent in the tree.

    # We take the right most leaf at the end
    positions = ["0" if N.location == RIGHT else "1" for N in internal_nodes] + [str(RIGHT)]
    leaf_position = "".join(positions)
    leaf_position_int = int( leaf_position , 2)

    # We compare the leaf position to the right most leaf in the tree
    if leaf_position_int > proof.width:
        return False

    # Check that one of the two leaves correspond to the client's opening balance
    value_client_balance_leaf = Hash(
        pack(proof.client_address , proof.client_opening_balance)
    )
    if value_client_balance_leaf not in [ leaf_left .value, leaf_right .value]:
        return False

    # Verify if the root can be recovered from proof
    reconstructed_root = build_root(proof)

    if reconstructed_root != root:
        return False

    # Negative balance would jeopardize security.
    for node in [ leaf_left , leaf_right ]:
        if node.amount < 0:
            return False

    for node in internal_nodes:
        validation_errors = [
            node.amount != node.left.amount + node.right.amount,
            node.left .amount < 0,
            node.right .amount < 0,
        ]
        if any( validation_errors ):
            return False
    return True
```

2.1.2.8 Root information

Due to padding techniques (see Section B.2) for building the Merkle tree, computing the root of the tree involves not only some balance and a digest but also the width and height of the tree. All this information is aggregated in the `RootInfo` fields described in Listing 10.

Listing 10 Root information

```
interface RootInfo {
  content: Digest
  height: Number
  width: Number
}
```

2.1.3 Time handling

The L2X protocol uses the blockchain in order to synchronize the participants. More specifically the time is divided into rounds. The size of a round is measured in number of blocks and is fixed at the beginning of the protocol when the mediator smart contract is deployed. Each round is divided into 4 quarters as shown in Figure 2.

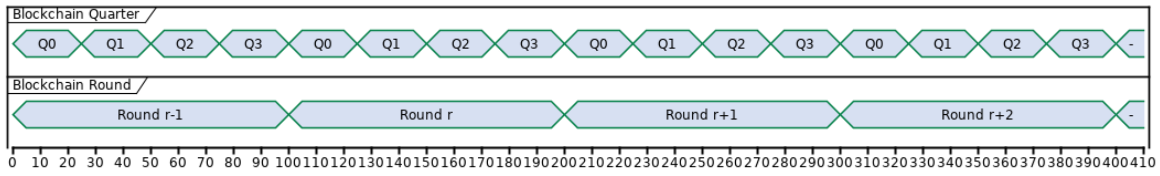

Figure 2: **Time progression**
The smallest unit of time is a block. In this figure a round contains 100 blocks and a quarter 25 blocks.

Depending on the interaction of the participants, rounds have a special status as described in Figure 3. Intuitively, like in blockchain protocols, the information stored in recent rounds is less reliable than the information from older rounds.

Round status	**Description**: The result of the off-chain trading activity for the round ...	**Distance to most recent round**
Uncommitted	... has not been committed.	0
Disputable	... has been committed and is open for clients to dispute.	1
Confirmed	... has been committed and can no longer be disputed.	≥ 2

Figure 3: Round status

2.2 $\mathcal{M}$: on-chain mediator

In this section we describe the algorithms used by the mediator. Note that we keep using the *Python* programming language to describe the code of this component which in practice is a smart contract.

2.2.1 `get_current_round`

The design of our protocol relies on a timeline which is composed of rounds and quarters (see Section 2.1.3). The mediator smart contract computes the current round via the function `get_current_round` (see Listing 11). If the contract is active (i.e. not halted), the current round number is computed based on the current block number, the block number when the contract was created and the round size. If the contract is halted, time progression stops and this function will always return the latest active round (which is the round in which the contract was halted).

Freezing the round number serves to prevent double spending between withdrawal (see Section 2.2.7) and funds recovery (see sections 2.2.11 and 2.2.12). Namely, it prevents malicious clients from confirming withdrawals made during round $r-1$ where r is the last active round. At the same time if an honest client has made a withdrawal request during round r' with $r' \leq r-2$ then they will be able to confirm this request at a time of their choosing (even if the contract is halted).

Listing 11 `get_current_round`

```
def get_current_round(self):
    """
    Returns the current round.
    If the Contract is in HALTED mode returns the last active round
    """

    if self.halted:  # Contract is halted
        current_round = self.last_active_round

    else:  # Contract is still active
        current_block_number = self.blockchain.get_current_block_number()
        current_round = (
            current_block_number - self.block_number_at_creation
        ) // self.round_size

    return current_round
```

2.2.2 `get_current_quarter`

The mediator computes the current quarter via the function `get_current_quarter` (see listing 12).

If the contract is active (i.e. not halted), the current quarter number is computed based on the current block number, the block number when the contract was created, and the quarter size (computed at contract creation).

If the contract is halted, time progression stops, and this function returns the latest active quarter determined by the `update_halted_state` function.

Listing 12 `get_current_quarter`

```
def get_current_quarter(self):
    """
    Returns the current quarter.
    If the Contract is in HALTED mode returns the last active quarter
    """

    if self.halted:
        current_quarter = self.last_active_quarter

    else:
        current_block_number = self.blockchain.get_current_block_number()
        blocks_since_creation = current_block_number - self.block_number_at_creation
        quarters_since_creation = blocks_since_creation // self.quarter_size

        current_quarter = quarters_since_creation % 4

    return current_quarter
```

2.2.3 `deposit_tokens`

Before trading, clients must deposit tokens in the mediator smart contract. Recall that in practice ERC20 tokens deposits must be done in two steps: 1) approve the mediator contract for the amount, 2) invoke the function `mediator.deposit_tokens` that will transfer the tokens to the mediator's account.

As shown in Listing 13, only a predetermined list of ERC20 tokens can be deposited. Trying to deposit Ethers or non-listed ERC20 tokens will raise an exception.

Listing 13 `deposit_tokens`

```
def deposit_tokens(self, sender_address, token_address, amount):
    """
    @param token_address ( Address ): type of tokens to be deposited
    @param amount ( Amount ): amount of tokens to be deposited
    """

    if transaction_sends_ether():
        raise Error("Ether deposits not accepted.")

    if self.state == HALTED:
        raise Error("Not possible to deposit in HALTED mode.")

    if not self.is_token_registered(token_address):
        raise Error("The specified token is not accepted.")

    current_round = self.get_current_round()
    status = self.blockchain[token_address].transfer(
        sender=sender_address, recipient=self.address, amount=amount
    )

    if not status:
        raise Error("The token transfer failed.")

    self.deposits[current_round][token_address][sender_address] += amount
    self.total_deposits[current_round][token_address] += amount

    emit(Deposit(current_round, token_address, sender_address, amount))
```

2.2.4 `commit`

At the beginning of each round, the operator must invoke the `commit` function (see Listing 14) for every supported asset. By doing so the mediator will store a short representation of the clients' balances for all the tokens. This information will be used in the case of a dispute or when a client wants to withdraw their tokens.

The `commit` function takes two arguments:

- `root_info`: The information corresponding to the root of the Merkle tree containing the clients' balances.
- `token_address`: the address of the ERC20 token.

The balance at the beginning of round r is computed recursively as follows:

$$A(r) = A(r-1) + D(r-1) - W(r-1) \tag{1}$$

where

- $A(r)$ is the opening balance for the current round.
- $D(r-1)$ is the total amount deposited during the previous round.
- $W(r-1)$ is the total amount requested for withdrawal during the previous round.

Note that $W(r-1)$ corresponds only to the withdrawal requests (made through `initiate_withdrawal`, see Section 2.2.5), not the confirmed withdrawals. Moreover we assume that these withdrawal requests are legitimate, that is they cannot be canceled by the operator (see Section 2.2.6).

Equation 1 does not involve quantities related to trading activity, because the mediator is not meant to handle operations that occur off-chain. Thus we assume that trades performed between the clients do not alter the total balance $A(r)$ for each token.

Listing 14 `commit`

```
@only_by("operator_address")
def commit(self, root_info: RootInfo, token_address: str):
    """
    @param root_info ( RootInfo ): Information needed to compute the root
    @param token_address ( Address ): address of the token the commit corresponds to
    """
    if self.state == HALTED:
        raise Error("Not possible to commit in HALTED mode.")

    if not self.is_token_registered(token_address):
        raise Error("The specified token is not accepted.")

    current_round = self.get_current_round()
    current_quarter = self.get_current_quarter()

    if current_round == 0:
        raise Error("Committing a new root is not allowed during round 0.")

    if current_quarter != 0:
        raise Error("Committing is only allowed during quarter 0.")

    if self.commits[current_round][token_address] != "":
        raise Error("Commit value already provided for this round.")

    last_round = current_round - 1

    prev_deposits = self.total_deposits[last_round][token_address]
    prev_withdrawals = self.total_requested_withdrawals[last_round][token_address]
    prev_opening_balance = self.opening_balances[last_round][token_address]

    prev_closing_balance = prev_opening_balance + prev_deposits - prev_withdrawals

    # Reconstructing the root
    # The pack function comes from the merkle_tree listing

    root_before_padding = Hash(pack(root_info.content, prev_closing_balance))
    root = Hash(pack(root_before_padding, root_info.height, root_info.width))

    self.opening_balances[current_round][token_address] = prev_closing_balance
    self.commits[current_round][token_address] = root
    self.commits_counter[current_round] += 1
    if self.commits_counter[current_round] == self.num_tokens:
        self.committed_rounds += 1
```

2.2.5 `initiate_withdrawal`

To withdraw assets from the mediator the client first calls `init_withdrawal` (see listings 16 and 15 for the helper function to validate a proof.). This method stores the client's intention to withdraw on-chain.

The arguments are:

- `proof`: the Merkle proof for the client's leaf. This also contains the asset address.
- `withdrawal_amount`: the amount to be withdrawn.

The proof enables the mediator to verify the client holds enough funds to carry out the withdrawal. As explained in Section 2.2.6, this request made during round r can be canceled by the operator during the same round r or during round $r + 1$. Absent cancellation by the operator, the client will be able to confirm the withdrawal from round $r + 2$ onwards. Each client can only have a single active withdrawal request at a time.

Listing 15 `is_proof_valid`

```
def is_proof_valid(self, proof, round):
    """
    @param proof ( Proof ): proof to be validated
    @param round ( Round ): round to check the proof against
    """
    asset = proof.token_address
    commitment = self.commits[round][asset]

    return validate(proof, commitment)
```

Listing 16 `initiate_withdrawal`

```
def initiate_withdrawal(self, sender_address, proof, withdrawal_amount):
    """
    @param proof ( Proof ): proof of previous round
    @param withdrawal_amount ( Amount ): the amount the client wants to withdraw
    """
    if self.state == HALTED:
        raise Error("Not possible to initiate a withdrawal in HALTED mode.")

    if withdrawal_amount == 0:
        raise Error("Not possible to initiate a withdrawal with an amount equal to 0.")

    current_round = self.get_current_round()

    if current_round == 0:
        raise Error("Not possible to initiate a withdrawal during round 0.")

    token_address = proof.token_address
    client_address = proof.client_address

    if sender_address != client_address:
        raise Error("A client can only initiate withdrawal for themselves.")

    if self.active_withdrawal_round[token_address][client_address] != 0:
        raise Error("A withdrawal for this token has already been initiated.")

    if not self.is_proof_valid(proof, current_round - 1):
        raise Error("The proof is invalid.")

    if withdrawal_amount > proof.client_opening_balance:
        raise Error(
            "The withdrawal amount exceeds the opening balance of the previous round."
        )

    request = WithdrawalRequest(withdrawal_amount, proof.client_opening_balance)

    self.requested_withdrawals[current_round][token_address][client_address] = request
    self.total_requested_withdrawals[current_round][token_address] += withdrawal_amount
    self.active_withdrawal_round[token_address][client_address] = current_round

    emit(InitWithdrawal(round, token_address, client_address, withdrawal_amount))
```

2.2.6 `cancel_withdrawal`

The `cancel_withdrawal` method is called by the operator to prevent a client from double spending.

Its arguments are:

- `approvals`: a list of approvals submitted by the client.
- `sigs`: a list of signatures of the approvals by the client.
- `token_address`: the address of the token corresponding to the withdrawal request.
- `client_address`: the address of the client who made the request.

If a **request for withdrawing W coins of an asset is made during round** r, a malicious client may try to create orders to sell S coins during the previous or same round so that $W+S > B$ where B is the client's balance at the beginning of round r. This double-spending (withdrawing on-chain and selling off-chain) scenario needs to be prevented as otherwise the malicious client can steal money from other users. Thus the operator is given until the beginning of the next round $r+1$ to cancel a withdrawal request from round r.

In order to do so, the operator will present the list of approvals where the sell side corresponds to the asset on which the client is trying to double-spend. The mediator checks if the client is double spending before proceeding with the cancellation. A successful cancellation prevents the client from ever confirming the withdrawal.

The operations related to withdrawal when the request and the cancellation happen during different rounds is depicted in Figure 4. The other case (the client's request and the operator's cancellation occur during the same round r) is described in Figure 5.

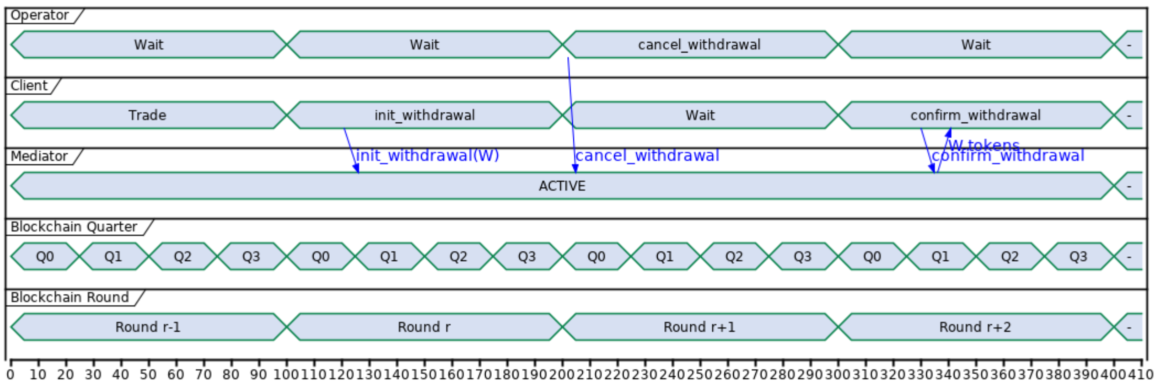

Figure 4: **Withdrawal initiation and cancel made during different rounds**
In this figure the time progresses from left to right through the block numbers shown at the bottom. Following trading activity in round $r-1$ the client initiates a withdrawal in round r. The operator may cancel the withdrawal until the end of round $r+1$ using the trading activity of the client during round $r-1$ and r. If the withdrawal is not canceled, the client can confirm it and obtain their tokens.

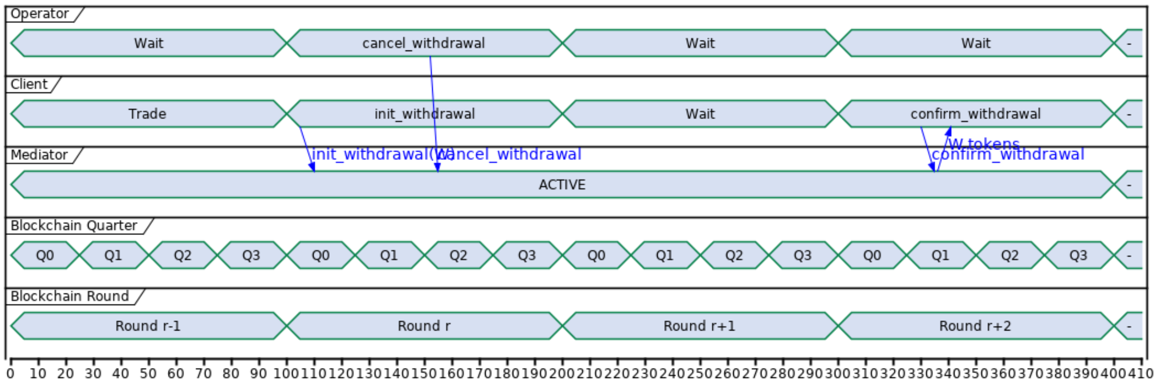

Figure 5: **Withdrawal initiation and cancel made during the same round**
The same scenario as in Figure 4 but the operator chooses to cancel the withdrawal during the same round r.

Note that a malicious operator may try to cancel legitimate withdrawal requests. Hence the mediator smart contract needs to verify the following conditions in order to allow cancellation of the request (see Listing 17):

- All the approvals are signed by the client.
- The approvals are unique (otherwise a malicious operator may artificially inflate the total sold amount by repeating approvals).
- The round of the approvals must correspond to the round of the proof (which is the round before the request was made) or the current round.

- If we have the following inequality then the withdrawal is canceled [8].

$$W(r) > A(r-1) - \Delta_{\texttt{sell}}(r-1) - \Delta_{\texttt{sell}}(r)$$

  where

  - $A(r-1)$ is the opening balance at round $r-1$
  - $\Delta_{\texttt{sell}}(r-1)$ is the amount locked during round $r-1$ due to trading activity.
  - $\Delta_{\texttt{sell}}(r)$ is the amount locked during round r due to trading activity.

Finally note that:

- We do not consider the deposits made during round $r-1$ nor round r. In all cases the client will be able to withdraw these amounts deposited in further rounds.
- The operator may need only a subset of approvals in order to establish the inequality above and cancel the withdrawal.
- The mediator ignores the approval corresponding to a buy order of the token as the operator can also omit to submit such approval.
- A malicious operator can cancel some withdrawals even though the client has sufficient off-chain balance. For example if the client has created approvals but also bought some of the underlying asset in trades. In this case the user can stop creating approvals and request a withdrawal in a later round.
- The fact that the operator cannot cancel withdrawals when the mediator is in `HALTED` mode does not mean invalid withdrawal requests can be confirmed (see Section 2.2.7). Indeed if the mediator halts during round r, then time is frozen to round r and will thus never reach round $r+1$. This means that withdrawal requests made during round $r-1$ cannot be confirmed.

[8] If the withdrawal request happens during round r and the operator tries to cancel it during the same round, then we have that $\Delta_{\texttt{sell}}(r) = 0$.

Listing 17 `cancel_withdrawal`

```python
def can_cancel_withdrawal(self, round_of_request, token_address):

    current_round = self.get_current_round()

    if (current_round == round_of_request) or (
        current_round == round_of_request + 1
        and self.commits[current_round][token_address] is None
    ):
        pass
    else:
        raise Error("This withdrawal request can no longer be cancelled")

def cancel_withdrawal(
    self, sender_address, approvals, sigs, token_address, client_address
):
    """
    @param approvals ( Approval [] ):  approvals from the client
    @param sigs ( Signature [] ):      signatures of the approvals
    @param token_address ( Address ): address of the token corresponding
                                      to the withdrawal request
    @param client_address ( Address ): address of the client who made
                                      the withdrawal request
    """

    if self.state == HALTED:
        raise Error("Not possible to cancel a withdrawal in HALTED mode.")

    if sender_address != self.operator_address:
        raise Error("Only the operator can cancel the withdrawal request.")

    if len(approvals) != len(sigs):
        raise Error("Number of approvals and sigs differs.")

    round_of_request = self.active_withdrawal_round[token_address][client_address]

    if round_of_request == 0:
        raise Error("There is currently no active withdrawal for the client.")

    round_of_proof = round_of_request - 1

    # Check that the withdrawal can still be cancelled
    self.can_cancel_withdrawal(round_of_request, token_address)

    withdrawal = self.requested_withdrawals[round_of_request][token_address][
        client_address
    ]

    # Check that all approvals are different
    approval_id_list = []
    for approval in approvals:
        approval_id = approval.id
        if approval_id in approval_id_list :
            raise Error("Approval appears more than once")
        approval_id_list .append(approval_id)

    # Check all approvals are valid
    for approval, signature in zip(approvals, sigs):
        if approval.round not in [round_of_proof, round_of_request]:
            raise Error("Approval not from round of request or round of proof.")

        if not (verify_signature(approval, signature, client_address)):
            raise Error("Invalid signature for approval.")

    # Compute amount of tokens reserved for trading
    reserved = 0
    for approval in approvals:
        if approval.sell_asset == token_address:
            reserved += approval.sell_amount

    requested = withdrawal.amount
    available = withdrawal.opening_balance - reserved

    if requested <= available:
        raise Error("Not overdrawing.")

    # Cancel the withdrawal
    self.requested_withdrawals[round_of_request][token_address][
        client_address
    ].amount = 0
    self.total_requested_withdrawals[round_of_request][token_address] -= requested
    self.active_withdrawal_round[token_address][client_address] = 0
```

2.2.7 `confirm_withdrawal`

Once the cancellation window expires without cancellation by the operator, the client calls `confirm_withdrawal` to claim the tokens.

This confirmation is done also via the mediator (see Listing 18). We check if there is a pending withdrawal and if the deadline for cancellation is over before transferring the tokens to the client. Due to the *withdraw-then-recover* attack (see Section A.1.7), some special care must be taken before confirming a withdrawal. In particular if the withdrawal request was made during round r, then it is not possible to confirm the withdrawal during the first quarter of round $r + 2$.

Listing 18 `confirm_withdrawal`. Note that we use some *Solidity*-like convention where an implicit argument corresponds to the address of the sender of the transaction: (`msg.sender` in *Solidity* corresponds to `sender_address`) in our code.

```
def confirm_withdrawal(self, sender_address, token_address):
    """
    @param token_address ( Address ) : the address for the token that
                                       needs to be withdrawn
    """
     client_address = sender_address

    request_round = self.active_withdrawal_round[token_address][client_address]

    if request_round == 0:
        raise Error("No active withdrawal at the moment")

    withdrawal_amount = self.requested_withdrawals[request_round][token_address][
        client_address
    ].amount

    current_round = self.get_current_round()
    current_quarter = self.get_current_quarter()

    # In quarter 0, we cannot conclusively determine if the self.mediator is halted
    if current_quarter == 0 or self.halted:
        last_confirmed_round = current_round - 3
    else:
        last_confirmed_round = current_round - 2

    if request_round > last_confirmed_round:
        raise Error("Too early to claim funds.")

    # Update local state to mark the withdrawal as done
    self.active_withdrawal_round[token_address][client_address] = 0

    self.send_tokens(token_address, self.address, client_address, withdrawal_amount)
```

2.2.8 `open_balance_dispute`

This method (see Listing 19) is used by the client to open a balance dispute on-chain. The client must call this method in the following scenarios:

- The array of proofs sent by the operator is invalid.
- The array of proofs does not accurately reflect the trading activity of the client during the previous round.
- The operator did not send the client any proof.

This balance dispute created on-chain contains information from the previous round:

- opening balances
- list of fills with operator signatures submitted by client

The function `close_balance_dispute` (See 2.2.9) uses this together with approvals provided by the operator to determine if the dispute is legitimate.

2.2.8.1 Dispute Elaborations

Within one quarter after the dispute is opened, the operator has to close the dispute by providing its own evidence that the opening balance for the client has been computed correctly. If the operator is malicious then it is expected to fail closing the dispute and the mediator will enter in `HALTED` mode (see Figure 6).

Note that challenging a single asset instead of all the assets at once (through the array of proofs) would not work. Indeed a malicious operator could use approvals in such a way that it is always able to close the dispute even though the opening balances are not consistent with the trading activity of the client. See sections A.1.1 and A.1.10.

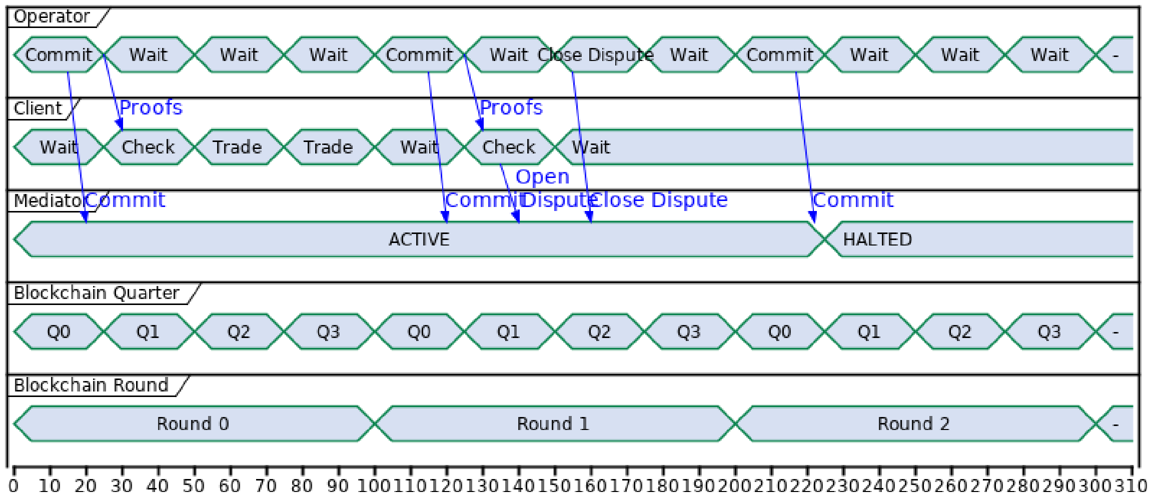

Figure 6: **Balance dispute timing**
In this figure the time unit is a block. Each round contains 100 blocks and thus a quarter has 25 blocks. We assume that the operator sends the proofs array to the client just after the end of the first quarter. The client then *checks* the proofs and *trades* off-chain if satisfied. Otherwise the client will open a dispute (Round 1, Quarter 1). The operator must try to close the dispute during the next quarter (Round 1, Quarter 2). In the example the operator fails to close the dispute and thus the mediator enters in `HALTED` mode (Round 2, Quarter 1).

Note that the client may invoke `open_balance_dispute` without any proofs array. This is necessary to protect the client's deposits in case the operator did not send a (valid) proofs array during the first round. At the same time, we must prevent a potential attack from a malicious client that would try to open a dispute without joining the protocol. If this happens the operator would not be able to close the dispute because the client's balance (even if it is 0) was not considered when computing the root. To prevent this, we force the client to provide either a proof or an authorization message along with the signature of the operator that guarantees the client is part of the protocol and allowed to trade. Also note that it is not possible to open a dispute during round 0: Indeed the operator would not be able to close it due to the lack of committed values available for this initial round.

Listing 19 `open_balance_dispute`

```
def is_proofs_array_valid (self , proofs, client_address , round):
    """
    @param proofs ( Proof [] ): array containing proofs
    @param client_address ( Address ): address of the client that opened the dispute
    @param round ( Round ): round of the dispute
    """

    if len(proofs) != len(self.tokens):
        return False

    for proof, token_address in zip(proofs, self.tokens):
        if proof.token_address != token_address:
            return False

    for proof in proofs:
        if not (proof.client_address == client_address):
            raise Error("Proof does not correspond to the client.")
        if not self. is_proof_valid (proof, round):
            raise Error("Proof is not valid for the round of the dispute.")

    return True

def open_balance_dispute(self, sender_address, proofs, fills , sig_fills , authorization):
    """
    @param proofs ( Proof [] ): proof of balances for each asset for round r−1
    @param fills ( Fill [] ): list of orders that have been processed and signed
                              by the operator
            along with the signatures
    @param sig_fills ( Signature [] ): signature on the fills
    @param authorization ( AuthorizationMessage ): signed message that states the
                                                  client has joined the protocol
    """

    if self.state == HALTED:
        raise Error("Not possible to open a dispute in HALTED mode.")

    if self.disputes[sender_address]["open"]:
        raise Error("There is already an open dispute.")

    current_round = self.get_current_round()

    if current_round == 0:
        raise Error("A client cannot open a dispute during round 0.")

    current_quarter = self.get_current_quarter()
    previous_round = current_round − 1

    if proofs is not None:

        if not (self. is_proofs_array_valid (proofs, sender_address, previous_round)):
            raise Error(f"The proofs array {proofs} is invalid.")

        balances = {}

        for proof in proofs:
            opening_balance = proof.client_opening_balance
            balances[asset] = opening_balance
    else:
        # Check the authorization_message
        if authorization.msg_type != "authorization":
            raise Error("Authorization is needed to open dispute without proof.")
        if not (authorization.content.client_address == sender_address):
            raise Error("Authorization is not granted for the client.")
        if not verify_signature(
            authorization.content.client_address,
            authorization.content.signature,
             self.operator_address,
        ):
            raise Error("The signature of authorization message is invalid.")

        if authorization.content.round >= current_round:
            raise Error("The authorization message is too recent.")

        for asset in self.tokens:
            balances[asset] = 0

    if len( fills ) != len( sig_fills ):
        raise Error("Every fill must be signed.")

    # Check the fills are valid
    for fill , sig in zip( fills , sig_fills ):
        if fill .round != previous_round:
            raise Error("Fill is not from previous round.")
        if not verify_signature( fill , sig, self .operator_address):
            raise Error("Fill not signed by operator.")
        if not fill . client_address == sender_address:
            raise Error("Fill is not assigned to the client.")

    # Check that fills are listed only once (no repetition)
    if has_duplicates( fill .id for fill in fills ):
        raise Error("Duplicate fills.")

    self .disputes[sender_address] = {
        "quarter": current_quarter,
        "round": current_round,
        "balances": balances,
        " fills ": fills ,
        "open": True,
    }

    self .dispute_counter[current_round] += 1

    emit(OpenBalanceDispute(round, sender_address))
```

2.2.9 `close_balance_dispute`

This method is called by the mediator to close a balance dispute. The operator must call this function before the end of the subsequent quarter after the dispute was opened. Failing to do so will halt the mediator.
The arguments are:

- Array of proofs of the opening balances of the client for each asset for the round r where r is the round when the dispute was open.
- List of approvals and their signature by the client corresponding to the trading approvals during round $r-1$.
- List of fills and their signature[9] by the operator corresponding to the trading activities during round r.
- The client's address.

The process of verification consists of the following high level steps:

- Verify that the approvals, fills and their respective signature are valid.
- Check that every fill is backed by an approval.
- The expected balance (roughly opening balance of round $r-1$ + deposits - pending withdrawals + changes due to trading) is equal to the opening balance of round r.

If these conditions are met, the dispute is closed and the mediator can keep operating. Otherwise the mediator will enter in `HALTED` mode during the next round as shown in Figure 6.

The pseudo-code for the `close_balance_dispute` function of the mediator is available in Listings 20, 21, and 22.

[9]Note that signatures here are not strictly necessary: Indeed the authenticity of the data provided by the operator is implicit as only this participant is allowed to call `mediator.close_balance_dispute`. Nonetheless it might be useful for clients to be able to store all fills and their respective signatures by the operator.

Listing 20 `close_balance_dispute`

```
def has_valid_open_dispute(self, dispute, current_round):
    """
    :param dispute ( Dispute ): The dispute object
    :return ( Bool ): true if the dispute is open and valid
    """

    # Check that the dispute is open
    if not dispute.open:
        raise Error("Dispute is already closed.")

    current_quarter = self.get_current_quarter()

    # Note: safe math should be used to avoid overflow
    elapsed_quarters = (
        (current_round - dispute.round) * 4 + current_quarter - dispute.quarter
    )

    # Check that it is the right moment to close the dispute
    if elapsed_quarters > 1:
        raise Error("It is too late to close the dispute.")

def check_dispute_accounting(client_address, proofs, approvals, fills, dispute_round):
    """
    :param client_address ( Address ): Address of the client who opened the dispute
    :param proofs ( Proof [] ): proofs array provided by the operator
    :param approvals ( Approval [] ): approvals provided by the operator
    :param fills ( Fill [] ): fills provided by the operator
    :param dispute_round: round of the dispute
    """

    # 1- Initial balances

    balances = {}
    for asset in self.tokens:
        balances[asset] = self.dispute.balances[asset]

    # 2- Deposits

    for asset in self.tokens:
        deposit = self.deposits[dispute_round][asset][client_address]
        if deposit > 0:
            balances[asset] += deposit

    # 3- Approvals

    approved_buys = []
    approved_sells = []

    if len(approvals) > 0:
        for i in range(0, len(approvals)):
            approval = approvals[i]
            buy_asset = approval.buy_asset
            sell_asset = approval.sell_asset

            if approval.intent == "buyAll":
                approved_buys[buy_asset] += approval.buy_amount
            approved_sells += approval.sell_amount
    # 4- Fills
    for i in range(0, len( fills )):
        fill = fills[i]
        approval = approvals[i]

        buy_asset = fill.buy_asset
        sell_asset = fill.sell_asset

        if approval.intent == "buyAll":
            if fill.buy_amount > approval.buy_amount:
                raise Error("Buy amount of fill is to large")
            approved_buys[buy_asset] -= fill.buy_amount
            if approved_buys[buy_asset] < 0:
                raise Error("Fills bought amount exceeds approved buy amount")
        else:
            if fill.sell_amount > approval.sell_amount:
                raise Error("Fill sold amount exceeds approved sell amount")
        approved_sells -= fill.sell_amount
        balances[buy_asset] += fill.buy_amount
        balances[sell_asset] -= fill.sell_amount

    # 5- Withdrawals
    for asset in self.tokens:
        withdrawal_amount = self.requested_withdrawals[dispute_round][asset][
            client_address
        ].amount
        balances[asset] -= withdrawal_amount

    # 6- Final balances
    for proof in proofs:
        asset = proof.asset
        if balance[asset] != proof.client_opening_balance:
            raise Error(
                "Expected opening balance does not match the balance of the proof"
            )
```

Listing 21 `close_balance_dispute`

```

def close_balance_dispute(
    self,
    sender_address,
    proofs,
    approvals,
    sig_approvals,
    fills,
    sig_fills,
    client_address,
):
    """
    @param proofs ( Proof [] ): proof of balances for each asset for the dispute round
    @param approvals ( Approval [] ): list of approvals generated during the round
                                  previous to the dispute
    @param sig_approvals ( Signature [] ): signatures on the approvals by the client
    @param fills ( Fill [] ): fills produced by the operator based on the approvals
    @param sig_fills ( Signature [] ): signatures of the fills
    @param client_address ( Address ): address of the client
    """

    if self.state == HALTED:
        raise Error("Not possible to close a dispute in HALTED mode.")

    if sender_address != self.operator_address:
        raise Error("Only the operator can close a balance dispute.")

    dispute = self.disputes[client_address]
    dispute_round = dispute.round

    current_round = self.get_current_round()

    self.has_valid_open_dispute(dispute, current_round)

    # Check the proofs
    if not (self.is_proofs_array_valid(proofs, client_address, dispute_round)):
        raise Error(f"The proofs array {proofs} is invalid.")

    # Check that all fills are unique
    if len(dispute.fills) > 1:
        for i in range(0, len(dispute.fills) - 1):
            if fill[i].id >= fill[i + 1].id:
                raise Error("Fill must all be different and sorted by id")

    # Check that the list fills provided by the operator includes all the fills provided
    # by the client
    client_fills = dispute.fills
    for client_fill in client_fills:
        if not (client_fill in fills):
            raise Error(
                "Some fill provided by the client has not been included "
                "by the operator."
            )

    # This is the last round for which the client agreed
    # with the opening balance
    disputed_round = dispute_round - 1

    # Check the approvals are valid
    for approval, sig in zip(approvals, sig_approvals):
        if not (verify_signature(approval, sig, client_address)):
            raise Error("Invalid signature for approval")
        if not (approval.round == disputed_round):
            raise Error("Incorrect round for approval")

    # Check the fills are valid
    for fill, sig in zip(fills, sig_fills):
        if not (verify_signature(fill, sig, self.operator_address)):
            raise Error("Invalid signature for fill")
        if not (fill.round == disputed_round):
            raise Error("Incorrect round for fill")
        if not (fill.client_address == client_address):
            raise Error("Wrong client address for fill")
```

Listing 22 `close_balance_dispute`

```
    # Check the fills are backed by approvals
    # Here we assume that the approval and fills that match are in the same position
    # Note that some approvals may be repeated as several fills can be matched to a
    # single approval.
    for approval, fill in zip(approvals, fills):

        # Check the match between the IDs
        if not (approval.approval_id == fill.approval_id):
            raise Error("Mismatch of IDs between approval and fill")

        # Check the assets involved are the same
        if not (approval.sell_asset == fill.sold_asset):
            raise Error("Assets for selling do not match")
        if not (approval.buy_asset == fill.bought_asset):
            raise Error("Assets for buying do not match")

        # check fill respects approval price

        # Avoid division by zero if buy_amount == 0 which could be a legitimate value,
        # for instance if the approval is used to pay a fee.

        # No price restriction.
        if approval.buy_amount == 0:
            continue
        # If the approval buy_amount is non-zero the fill buy_amount
        # must be non-zero too.
        if fill.bought_amount == 0:
            raise Error("Approval does not allow zero buy amount.")

        condition = (fill.sold_amount * approval.buy_amount) <= (
            approval.sell_amount * (fill.bought_amount)
        )

        # Check that fill_price > approval_price:
        if not (condition):
            raise Error("Fill price exceeds approval price")

    # Compare claimed balances from the proofs with computed balances
    self.check_dispute_accounting(
        client_address, proofs, approvals, fills, dispute.round
    )

    # Close the dispute
    self.disputes[client_address].open = False
    self.dispute_counter[dispute_round] -= 1
```

2.2.10 `update_halted_state`

The mediator halts (or enters `HALTED` mode) if any of the following is true:

- The operator does not invoke the `commit` function correctly before the end of the first quarter of each round for each asset.
- The operator fails to close all disputes of the previous round.

Once halted, the mediator remains in this state forever and the only functions that can be invoked are `recover_all_funds` (see Section 2.2.11), `recover_on_chain_funds_only` (see Section 2.2.12) and `confirm_withdrawal` (see Section 2.2.7). Moreover time progression stops which means in practice that the round number and quarter number will freeze and keep their value forever (see sections 2.2.1 and 2.2.2).

When created, the mediator smart contract has its internal variable `halted` set to `False`. When the mediator is halted that variable is set to `True`.

Listing 23 `update_halted_state`

```
def update_halted_state(self):

    if self.halted:
        return

    # Checks if the root has been committed before the end of the first quarter
    current_round = self.get_current_round()
    current_quarter = self.get_current_quarter()

    # No commit allowed in first round.
    # If the first round is ongoing, it is too early to update the halted state.
    if current_round == 0:
        return

    previous_round = current_round - 1

    no_tokens_registered = self.num_tokens == 0

    num_commits = self.commits_counter[current_round]

    if current_quarter == 0:
        missing_commits = num_committed_rounds < previous_round

        if previous_round > 0:
            has_open_disputes = (
                self.open_balance_disputes_counter[previous_round - 1] > 0
            )
    else:  # current_quarter in [1,2,3]
        missing_commits = num_committed_rounds < current_round
        has_open_disputes = self.open_balance_disputes_counter[current_round - 1] > 0

    if no_tokens_registered or missing_commits or has_open_disputes:
        self.halted = True
        self.last_active_round = current_round
        self.last_active_quarter = current_quarter
```

Note that this function is somehow subtle as its behavior differs whether it is invoked during quarter 0 or not. The reason is that the function `commit` requires to be invoked during quarter 0. In this case we can only check that all the commits were made for the previous round. Similarly during quarter 0 the operator can still close disputes created during the previous round, so in this case we only check that all the disputes opened two rounds ago have been closed.

Finally note that the function `update_halted_state` is not meant to be called directly by the participants but is invoked at the beginning of other functions which behavior depends on whether the mediator is halted or not.

2.2.11 `recover_all_funds`

When the mediator enters in `HALTED` mode the client should be able to recover its funds. By invoking `recover_all_funds` with a proof for the opening balance of round $r-2$ where r is first round in `HALTED` mode they can obtain:

- The opening balance at the beginning of round $r-2$.
- All deposits made during rounds $r-2$, $r-1$ and r.

The reason why funds from the beginning of round $r-2$ are recovered is because $r-2$ is the last confirmed round when the mediator halts in round r. Indeed the balances from round $r-1$ might not be trustworthy: for example a malicious operator may not allocate correctly the amounts to some client, inviting this same client to open a dispute during round $r-1$, which will cause the mediator to halt during round r.

Listing 24 `recover_all_funds`

```
def recover_all_funds(self, sender_address, proof):
    """
    @param proof (Proof): proof from two rounds ago
    """

    self.update_halted_state()

    if not self.halted:
        raise Error("The contract must be halted.")

    client_address = sender_address
    token_address = proof.token_address

    if self.recovered[token_address][client_address]:
        raise Error("The client has already recovered her funds.")

    if not (proof.client_address == client_address):
        raise Error("The proof does not belongs to the client.")

    current_round = self.get_current_round()

    if not self.is_proof_valid(proof, current_round - 2):
        raise Error("The proof provided is not valid for round r-2")

    opening_balance_two_rounds_ago = proof.client_opening_balance

    funds_to_recover = (
        opening_balance_two_rounds_ago
        + self.deposits[current_round][token_address][client_address]
        + self.deposits[current_round - 1][token_address][client_address]
        + self.deposits[current_round - 2][token_address][client_address]
    )

    self.recovered[token_address][client_address] = True
    self.send_tokens(token_address, self.address, client_address, funds_to_recover)
```

2.2.12 `recover_on_chain_funds_only`

If the client joined the protocol "recently" and thus is not in possession of any proof while the mediator enters in `HALTED` mode, they need to invoke `recover_on_chain_funds_only`. By doing so they will obtain all their deposits from rounds r, $r-1$ and $r-2$ where r is the round when the mediator entered in `HALTED` mode.

Note that it is not possible to invoke both `recover_on_chain_funds_only` and `recover_all_funds` successfully. Thus a client with a positive open balance at round $r-2$ should not call `recover_on_chain_funds_only`. Funds recovery must be done for each different token address and thus according to the previous observation, the client will have to call `recover_on_chain_funds_only` or `recover_all_funds` for each token address.

Listing 25 `recover_on_chain_funds_only`

```python
def recover_on_chain_funds_only(self, sender_address, token_address):
    """
    @param token_address ( Address ): Address of the token to be recovered
    """

    self.update_halted_state()

    if not self.halted:
        raise Error("The contract must be halted.")

    client_address = sender_address

    if self.recovered[token_address][client_address]:
        raise Error("The client has already recovered her funds.")

    current_round = self.get_current_round()

    funds_to_recover = self.deposits[current_round][token_address][client_address]

    if current_round >= 1:
        funds_to_recover += self.deposits[current_round - 1][token_address][
            client_address
        ]
    if current_round >= 2:
        funds_to_recover += self.deposits[current_round - 2][token_address][
            client_address
        ]

    self.recovered[token_address][client_address] = True
    self.send_tokens(token_address, self.address, client_address, funds_to_recover)
```

2.3 $\mathcal{O}$: off-chain operator

2.3.1 `admit`

In order to enable trading, a client must get registered. This is done via the `admit` function. Note that this registration process allows the client to trade any of the listed tokens. In practice a manual process for enrolling the client may happen, in particular to fulfill legal obligations.

Listing 26 `admit`

```python
def admit(self, client_address):
    """
    @param client_address ( Address ): address of the client who wants to join.
    """

    for ledger in self.ledgers.values():
        ledger.register(client_address)

    if client_address not in self.client_addresses:
        self.client_addresses.append(client_address)

        sig = self.sign(client_address, self.private_key)

        return {"client_address": client_address, "signature": sig}
    else:
        return {}
```

2.3.2 `commit`

The `commit` function enables the operator to publish the root of every asset's Merkle tree on the blockchain. This function must be called before the end of the first quarter. As shown in Section 2.3.7.1, all illegitimate withdrawals must be canceled before committing.

Listing 27 commit

```python
def commit(self):
    """
    This function commits the roots corresponding to each supported token to
    the mediator.
    """

    for token_address, ledger in self.ledgers.items():
        root_info = ledger.get_root()
        self.mediator.commit(root_info, token_address)
```

2.3.3 provide_proof

At the beginning of each round, the operator commits a new root for each asset. During the subsequent quarter every client will request a proof for all assets in order to verify the correctness of their opening balance.

Listing 28 provide_proof

```python
def provide_proof(self, client_address, round):
    """
    @param client_address ( Address ): address of the client who asked for the proof.
    @param round ( Round ): round when the root corresponding to the proof was computed.
    """
    return [
        ledger.get_opening_balance_proof(client_address, round)
        for ledger in self.ledgers.values()
    ]
```

2.3.4 credit_deposit

The operator must react to deposit events from `mediator.deposit_tokens` and credit the client's balance according to the deposit outcome.

Failure to account for on-chain deposits will prevent the operator from being able to commit the ledger root to the mediator in the next round. In addition, incorrectly calculated balance will invite balance dispute from an honest client, who will not be able to successfully validate the proof given by the operator against the root committed to the mediator.

Listing 29 `credit_deposit`

```
def credit_deposit(self, client_address, token_address, amount, round):
    """
    @param client_address ( Address ): address of the client who made a deposit.
    @param token_address ( Address ): token of the deposit
    @param amount ( Amount ): amount of the deposit
    @param round ( Round ): round when the deposit was made
    """

    ledger = self.ledgers[token_address]

    ledger.credit_deposit(client_address, amount, round)
```

2.3.5 `moderate_withdrawal`

If the mediator becomes aware of a withdrawal request where a client overdraws their balance, it must invoke the `moderate_withdrawal` on the mediator, or risks being challenged by an honest client for incorrect bookkeeping. This *double spending attack* is further examined in Section 2.2.6.

The function `moderate_withdrawal` will check if some double-spending happened and if that is the case `mediator.cancel_withdrawal()` will be called.

Listing 30 `moderate_withdrawal`

```
def moderate_withdrawal(self, client_address, withdrawal_request):
    """
    @param client_address ( Address ): address of the client who did a
                                       withdrawal request.
    @param withdrawal_request ( WithdrawalRequest ): withdrawal request information.
    """

    # Check the conditions
    amount = withdrawal_request.amount
    token_address = withdrawal_request.token_address
    round = withdrawal_request.round

    ledger = self.ledgers[token_address]

    offchain_sell = ledger.get_total_sold(client_address, round)
    opening_balance = ledger.get_opening_balance(client_address, round)

    journal = ledger.get_journal()

    # Cancel withdrawal
    if offchain_sell + amount > opening_balance:
        approvals, sig_approvals = journal.get_sell_approvals(client_address, round)

        self.mediator.cancel_withdrawal(
            approvals, sig_approvals, token_address, client_address
        )
    # Update opening balance for next round
    else:
        journal.withdraw(client_address, amount)
```

2.3.6 `close_balance_dispute`

When a client opens a dispute, an honest operator needs to be able to close this dispute, otherwise the mediator will enter in `HALTED` mode. The information needed to close the dispute is on the blockchain and locally stored by the operator. Note that a dispute stored on-chain is implicitly valid as the mediator does the necessary verifications when the client calls `mediator.open_balance_dispute`.

Listing 31 close_balance_dispute

```
def close_balance_dispute(self, client_address, dispute_round):
    """
    @param client_address ( Address ): address of the client who opened the dispute.
    @param dispute_round ( Round ): round when the dispute was open
    """

    last_confirmed_round = dispute_round - 1

    proofs = []
    approvals = []
    sig_approvals = []
    fills = []
    sig_fills = []

    sorted_list_of_registered_tokens = self.mediator.get_sorted_tokens_list()
    for token in sorted_list_of_registered_tokens:
        ledger = self.ledgers[token]
        proof = ledger.get_opening_balance_proof(client_address, dispute_round)
        proofs.append(proof)

    fills, sig_fills = store_fills.retrieve(
        "All fills for client %s and round %d" % (client_address, last_confirmed_round)
    )

    approvals = []
    sig_approvals = []
    for signed_fill in signed_fills:
        approval_id = fill.approval_id
        approval, sig_approval = store_approvals.retrieve(
            "Approval with identifier %s" % (approval_id)
        )
        approvals.append(approval)
        sig_approvals.append(sig_approval)

    self.mediator.close_balance_dispute(
        proofs, approvals, sig_approvals, fills, sig_fills, client_address
    )
```

2.3.7 Events

2.3.7.1 OnNewQuarter

When a new quarter starts the operator must always try to cancel withdrawal requests and close balance disputes generated during the quarter. If the quarter coincides with the beginning of a new round, then the operator must commit a root for every asset. Note that the order of execution in Listing 32 matters: Illegitimate withdrawal requests must be canceled before the roots can be committed. Not respecting this rule would expose the operator to not being able to close a dispute during the next quarter. Withdrawal requests made during the current quarter cannot be processed right away as it might not be able in practice to catch them all. These requests will be processed during the next quarter.

Furthermore, implementation must ensure all the withdrawal requests and balance disputes from the previous quarter have been fully enqueued for processing before `on_new_quarter` is invoked.

Listing 32 `EventOperatorOnNewQuarter`

```
def on_new_quarter(self, current_round, current_quarter):
    """
    @param current_round ( Round ): current active round of the mediator.
    @param current_quarter ( Quarter ): current active quarter of the mediator.
    """

    # Cancel withdrawals if needed
    for client_address in self.client_addresses:
        for withdrawal_request in self.withdrawal_requests[client_address]:
            self.moderate_withdrawal(client_address, withdrawal_request)
            self.withdrawal_requests[client_address].remove(withdrawal_request)

    # Close dispute if needed
    for client_address in self.client_addresses:
        for dispute in self.disputes[client_address]:
            self.close_balance_dispute(client_address, dispute)
            self.disputes[client_address].remove(dispute)

    # Commit to new root if needed
    if current_quarter == 0: # a new round begins
        self.commit()
```

2.3.7.2 OnInitiateWithdrawal

When a client initiates a withdrawal the operator must record the event locally in order to use the information of the request later.

Listing 33 `EventOperatorOnInitiateWithdrawal`

```
@events.register("operator", events.InitWithdrawal)
def on_initiate_withdrawal(self, client_address, withdrawal_request):
    """
    All information comes from the blockchain
    @param client_address ( Address ): address of the client that initiates the request.
    @param withdrawal_request ( WithdrawalRequest ): information relative to the
                                                    withdrawal request.
    """
    self.withdrawal_requests[client_address].append(withdrawal_request)
```

2.3.7.3 OnDeposit

When a client makes a deposit, the operator must update the balances accordingly.

Listing 34 EventOperatorOnDeposit

```
@events.register("operator", events.Deposit)
def on_deposit(self, client_address, token_address, amount, round):
    """
    All information comes from the blockchain
    @param client_address ( Address ): address of the client making the deposit
    @param token_address ( Address ): address of the token used for the deposit
    @param amount ( Amount ): amount of tokens deposited.
    @param round ( Round ): round when the deposit is made
    """

    if client_address not in self.client_addresses:
        raise Error("Client address does not exist.")
    self.credit_deposit(client_address, token_address, amount, round)
```

2.3.7.4 OnOpenBalanceDispute

When a client opens a balance dispute the operator must store the corresponding information in order to close the dispute during the next quarter.

Listing 35 EventOperatorOnOpenBalanceDispute

```
@events.register("operator", events.OpenBalanceDispute)
def on_open_balance_dispute(self, client_address, dispute_round):
    """
    All information comes from the blockchain
    @param client_address ( Address ): address of the client who opens the dispute.
    @param dispute_round ( Round ): round when the dispute was opened.
    """

    self.disputes[client_address].append(dispute_round)
```

2.3.7.5 OnReceiveApproval

When the operator receives an approval from the exchange, it must check if the approval is valid before storing it for later use, such as dispute handling.

Listing 36 EventOperatorOnReceiveApproval

```python
def on_receive_approval(self, approval, approval_sig, client_address):
    """
    The information is obtained from the client (HTTP request)
    @param approval ( Approval ): approval received from the client.
    @param approval_sig ( Signature ): signature on the approval by the client.
    @param client_address ( Address ): address of the client.
    """

    if approval.id in self.approvals:
        raise Error("Approval already exists")

    if not verify_signature(approval, approval_sig, client_address):
        raise Error("Invalid signature for approval.")

    if approval.round != self.mediator.get_current_round():
        raise Error("Approval is not for this round")

    # Approval buy amount may be zero if the approval is used for fees
    if approval.buy_amount < 0 or approval.sell_amount <= 0:
        raise Error("Approval amounts must be positive")

    sell_asset_ledger = self.ledgers[approval.sell_asset]
    sell_asset_balance = sell_asset_ledger.free_balance(approval.client_address)

    if approval.sell_amount > sell_asset_balance:
        raise Error("Insufficient balance to sell")

    self.approvals[approval.id] = (approval, approval_sig)
```

2.4 $\mathcal{P}$: clients

2.4.1 join

In order to start trading, the client needs to join the protocol. This is done by the client sending a (signed) GetAuthorizationMessage message to the exchange. After validating the message, if the client is authorized to join, the exchange will return a (signed) AuthorizationMessage message from the operator. This message is needed in case the client wants to open a dispute but was not provided with any proof (see Section 2.2.8).

Listing 37 `join`

```python
def join(self):

    client_address = self.address

    sig = self.sign(client_address, self.private_key)

    authorization_message = self.operator.authorize(client_address, sig)

    if not (authorization_message.client_address == client_address):
        raise Error("Invalid client address in authorization message.")

    current_round = self.mediator.get_current_round()
    if not (authorization_message.round == current_round):
        raise Error("Invalid round in authorization message.")

    if not verify_signature(
        authorization_message.client_address,
        authorization_message.signature,
        OPERATOR_ADDRESS,
    ):
        raise Error("Invalid signature for authorization message.")

    self.authorization = authorization_message
```

2.4.2 `deposit`

Prior to placing orders, clients must ensure there is sufficient holdings in their account for the asset they wish to sell. This is done by making a deposit into the mediator smart contract. Note that if the mediator is in `HALTED` mode or if the client does not have an authorization then no deposit should be made.

Listing 38 `deposit`

```python
def deposit(self, token_address, amount):

    if self.mediator.is_halted():
        raise Error("Cannot make deposit into a halted mediator.")

    if self.authorization is None:
        raise Error("The client needs to be authorized before making a deposit.")

    # Note: in practice, for ERC20 tokens, we would need to approve the amount first.
    self.mediator.deposit_tokens(self.address, token_address, amount)

    current_round = self.mediator.get_current_round()
    self.deposits[token_address][current_round].append(amount)
```

2.4.3 `audit`

In order to ensure that the balances committed by the operator are correct, each client must ask for the proofs array (one proof for each asset) and verify each of these proofs. If one proof is not valid or some proof is missing, the client must open a balance dispute.

Listing 39 `audit` (1/2)

```
def is_proofs_array_valid(self, proofs):
    """
    Checks that there is exactly one proof for each token.
    @param proofs ( Proof [] ): array containing proofs
    """

    if len(proofs) != len(self.tokens):
        return False

    for proof, token_address in zip(proofs, self.tokens):
        if proof.token_address != token_address:
            return False

    return True
```

Listing 40 audit (2/2)

```

def audit(self):
    """
    Check that the proofs sent by the operator are correct.
    If the proofs are not correct open a dispute.
    """

    audit_successful = True

    current_round = self.mediator.get_current_round()

    proofs = self.operator.provide_proof(self.address, current_round)

    if not (self.is_proofs_array_valid(proofs)):
        audit_successful = False
    else:
        for proof in proofs:
            commit_value = self.mediator.get_commit_value(
                current_round, proof.token_address
            )

            # Fetch the expected balance of the client
            expected_balance = self.get_balance(proof.token_address, current_round)

            if any(
                [
                    validate(proof, commit_value) is False,
                    proof.client_opening_balance != expected_balance,
                    proof.client_address != self.address,
                ]
            ):
                audit_successful = False
                break
    # If the audit is successful, store the proofs, otherwise open a dispute
    if audit_successful:
        for proof in proofs:
            self.proofs[proof.token_address][current_round] = proof
    else:
        previous_round = current_round - 1
        proofs = [self.proofs[token][previous_round] for token in sorted(self.balances)]
        fills, sig_fills = self.get_fills(previous_round)
        self.mediator.open_balance_dispute(proofs, fills, sig_fills, self.authorization)
```

2.4.4 send_approval

When the client wants to trade they must send an approval directly to the exchange. Note that the identifier of the approval is generated by the client (`Approval.Id`) and must be unique for each approval.

Listing 41 `send_approval`

```python
def create_approval(
    self, round, buy_asset, buy_amount, sell_asset, sell_amount, intent
):

    # Generate a unique identifier
    nonce = generate_random_nonce()
    params = (
        str(round)
        + str(buy_asset)
        + str(buy_amount)
        + str(sell_asset)
        + str(sell_amount)
        + str(intent)
    )
    identifier = Hash(nonce + params)

    return Approval(
        round, identifier, buy_asset, buy_amount, sell_asset, sell_amount, intent
    )

def send_approval(self, approval):

    sig = self.sign(approval, self.private_key)
    self.operator.send(
        {"msg_type": "approval", "content": {"approval": approval, "signature": sig}}
    )
```

2.4.5 `initiate_withdrawal`

When a client wants to withdraw some tokens they need first to initiate a withdrawal. If the withdrawal is legitimate, the tokens will be available for the client in two rounds.

Listing 42 `initiate_withdrawal`

```python
def initiate_withdrawal(self, token_address, amount):

    if self.get_available_balance(token_address) < amount:
        raise Error("Not enough funds to initiate a withdrawal.")

    current_round = self.mediator.get_current_round()
    proof = self.proofs[token_address][current_round - 1]

    self.mediator.initiate_withdrawal(self.address, proof, amount)
    self.active_withdrawals[token_address] = current_round
```

2.4.6 `confirm_withdrawal`

As mentioned in Section 2.4.5, in order to obtain the tokens that have been requested for withdrawal, the client must confirm the withdrawal. In order to do so, the client must keep track of their withdrawal requests.

Listing 43 `confirm_withdrawal`

```python
def confirm_withdrawal(self, token_address):

    current_quarter = self.mediator.get_current_quarter()
    current_round = self.mediator.get_current_round()

    round_withdrawal = self.active_withdrawals[token_address]
    if round_withdrawal == 0:
        raise Error("No active withdrawal for this token address.")

    if not (
        (((current_round - round_withdrawal) == 2) and current_quarter > 0)
        or (current_round - round_withdrawal > 2)
    ):
        raise Error("Too early to withdraw")
    else:
        self.mediator.confirm_withdrawal(token_address)
        self.active_withdrawals[token_address] = 0
```

2.4.7 `recover_funds`

If during round r the mediator enters in `HALTED` mode, the client must recover their funds. If the client is in possession of a proof for round $r-2$ they will invoke `mediator.recover_all_funds` otherwise they will call `mediator.recover_on_chain_funds_only`.

Listing 44 `recover_funds`

```python
def recover_funds(self, token_address):

    if not (self.mediator.is_halted()):
        raise Error("Cannot recover funds yet.")

    current_round = self.self.mediator.get_current_round()

    if current_round >= 2:
        proof = self.proofs[token_address][current_round - 2]

    if proof is not None:
        self.mediator.recover_all_funds(proof)
    else:
        self.mediator.recover_on_chain_funds_only()
```

2.4.8 Events

2.4.8.1 OnNewQuarter

When a quarter begins, the client must check if this is the second quarter. In this case they must audit the proofs produced by the operator during the first quarter. Finally, a client needs to be constantly checking for pending withdrawals and confirm those if needed.

Listing 45 EventClientOnNewQuarter

```
def on_new_quarter(self, current_quarter):
    """
    All information comes from the blockchain
    @param current_quarter: current active quarter for the mediator
    """

    if current_quarter == 1:
        # Checks the proofs
        self.audit()

        # Confirm a withdrawal if needed
        for token_address in self.balances:
            self.confirm_withdrawal(token_address)
```

2.4.8.2 OnReceiveFill

When a client receives a fill from the operator, they need to verify its validity and store it locally in order to use it later for a potential dispute.

Listing 46 EventClientOnReceiveFill

```
def on_receive_fill(self, fill, fill_sig):

    if verify_signature(fill, fill_sig, OPERATOR_ADDRESS):
        fill_round = fill.round
        signed_fill = {"fill": fill, "sig": fill_sig}
        self.fills[fill_round].append(signed_fill)
```

2.4.8.3 OnMediatorHalted

If the mediator is in `HALTED` mode, then the client has to recover all their tokens.

Listing 47 EventClientOnMediatorHalted

```
def on_mediator_halted(self):

    for token_address in self.balances:
        self.recover_funds(token_address)
```

2.5 Behavior

2.5.1 A client joins the L2X instance

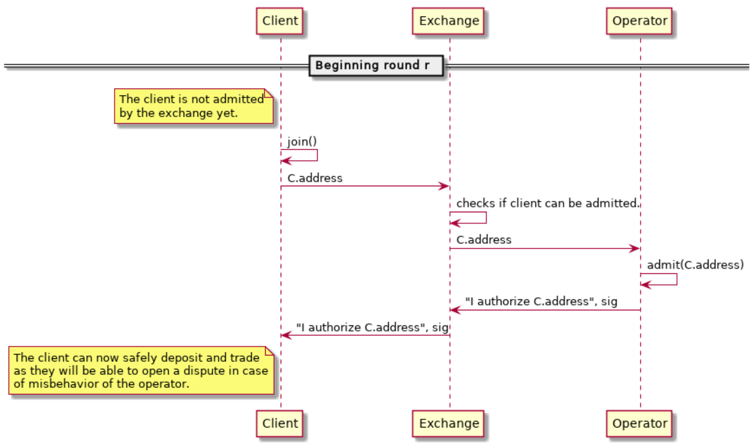
Client
Exchange
Operator
Beginning round r
The client is not admitted by the exchange yet.
join()
C.address
checks if client can be admitted.
C.address
admit(C.address)
"I authorize C.address", sig
"I authorize C.address", sig
The client can now safely deposit and trade as they will be able to open a dispute in case of misbehavior of the operator.
Client
Exchange
Operator

Figure 7: **A client joins the L2X instance**

The client must first seek authorization from the operator through the exchange, prior to depositing tokens into the mediator by executing `join()` (see Section 2.4.1).

Once the client has been authorized, the client's account will be entered into the operator's ledgers, and the authorization record returned by the operator through `join()` allows the client to safely make deposits and start trading.

2.5.2 Client deposits tokens

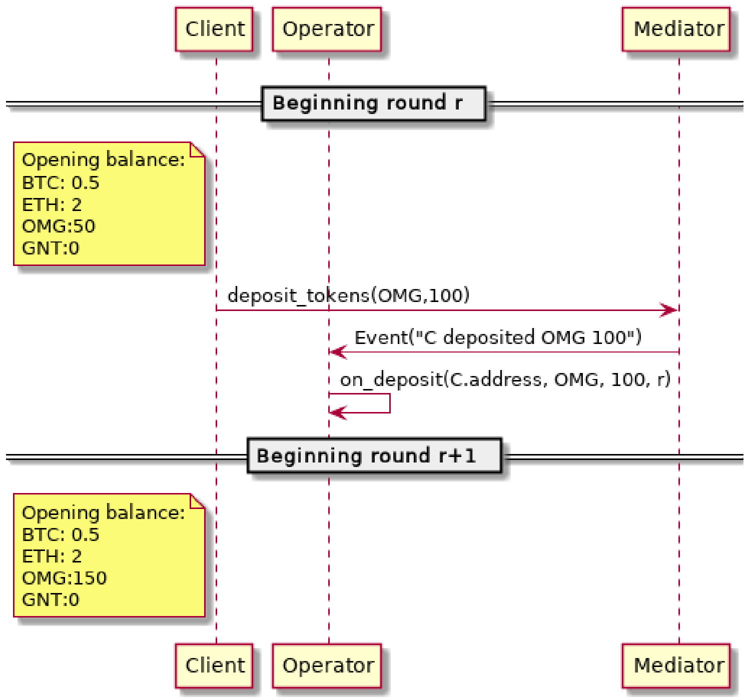

Figure 8: **A client deposits tokens**

In order to be able to trade the client must first deposit some tokens into the mediator. When this happens the mediator will generate a deposit event that will be captured by the operator. The operator will then register the client's account into the ledgers to be committed to the mediator in the next round.

2.5.3 Trade offline

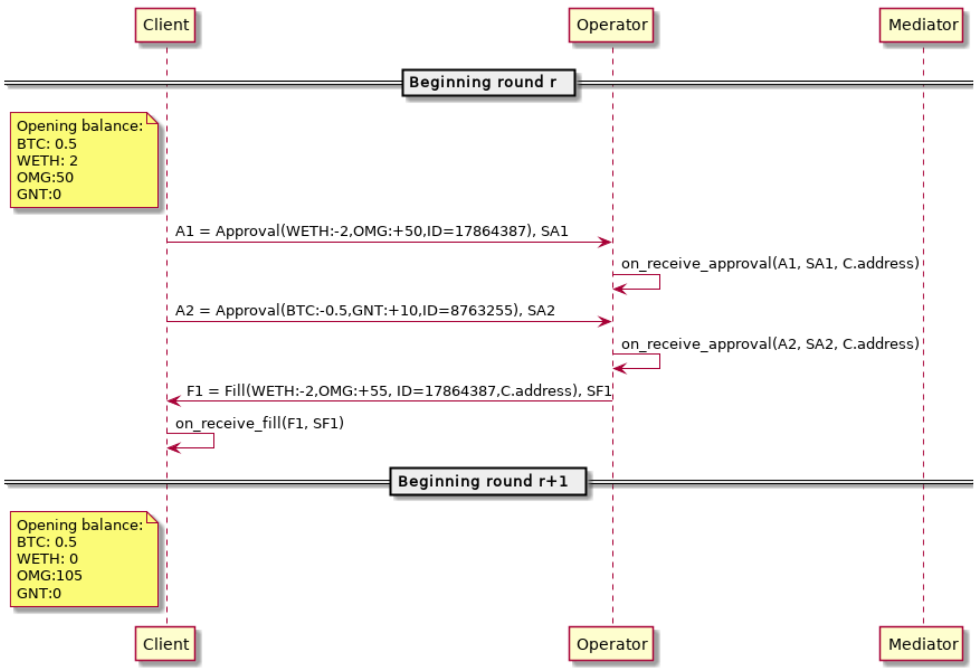
Client
Operator
Mediator
Beginning round r
Opening balance:
BTC: 0.5
WETH: 2
OMG:50
GNT:0
A1 = Approval(WETH:-2,OMG:+50,ID=17864387), SA1
on_receive_approval(A1, SA1, C.address)
A2 = Approval(BTC:-0.5,GNT:+10,ID=8763255), SA2
on_receive_approval(A2, SA2, C.address)
F1 = Fill(WETH:-2,OMG:+55, ID=17864387,C.address), SF1
on_receive_fill(F1, SF1)
Beginning round r+1
Opening balance:
BTC: 0.5
WETH: 0
OMG:105
GNT:0
Client
Operator
Mediator

Figure 9: **A client trades offline**

One of the attractive feature of L2X is that all trading activity happens offline. In practice the client will send signed approvals to the operator. These approvals will be forwarded to the exchange engine.

When an approval is matched by the engine, the operator will produce the corresponding fills and send them back to the client.

Both participants need to update their ledgers when receiving approvals/fills.

2.5.4 The client withdraws some tokens

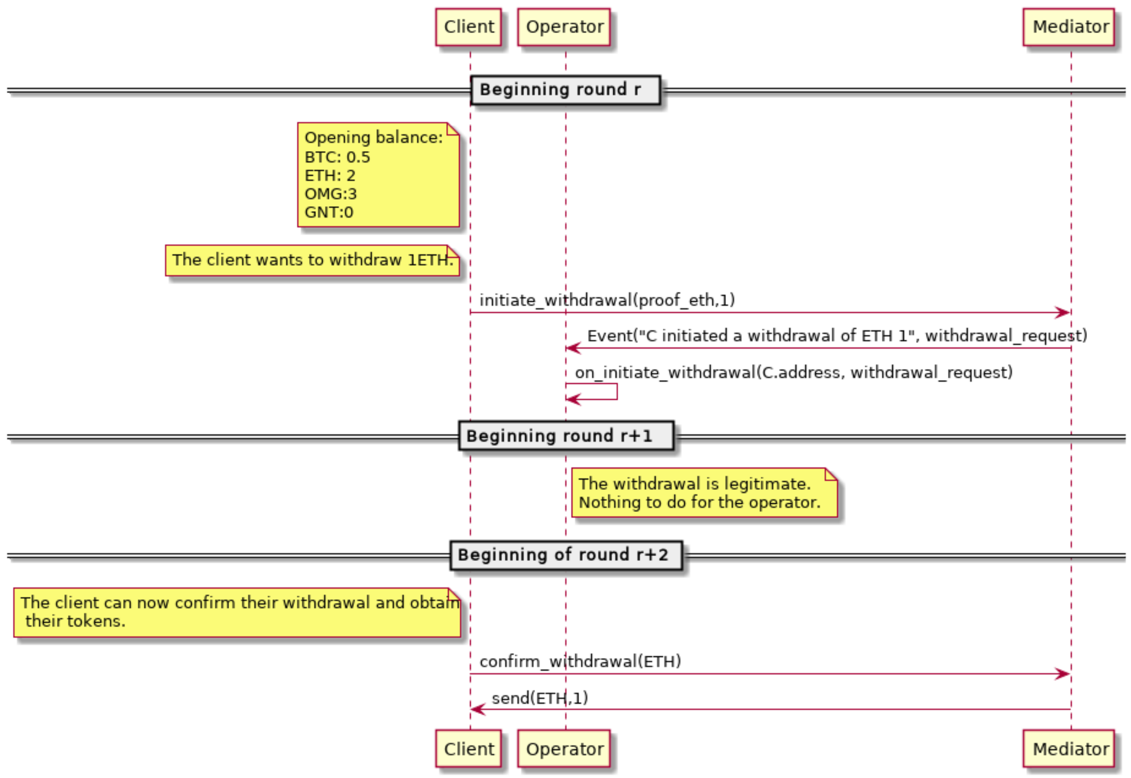
Client
Operator
Mediator
Beginning round r
Opening balance:
BTC: 0.5
ETH: 2
OMG:3
GNT:0
The client wants to withdraw 1ETH.
initiate_withdrawal(proof_eth,1)
Event("C initiated a withdrawal of ETH 1", withdrawal_request)
on_initiate_withdrawal(C.address, withdrawal_request)
Beginning round r+1
The withdrawal is legitimate.
Nothing to do for the operator.
Beginning of round r+2
The client can now confirm their withdrawal and obtain their tokens.
confirm_withdrawal(ETH)
send(ETH,1)
Client
Operator
Mediator

Figure 10: **An honest client withdraws some tokens**

When an honest client (see Figure 10) needs to withdraw some tokens they send a withdrawal request to the mediator. This withdrawal request will generate an event that is captured by the operator, who will validate the request and either update the ledgers if the request is valid, or cancel the withdrawal if the client overdraws their available balances. If the operator did not cancel the withdrawal, the client will be able to obtain the tokens after two rounds.

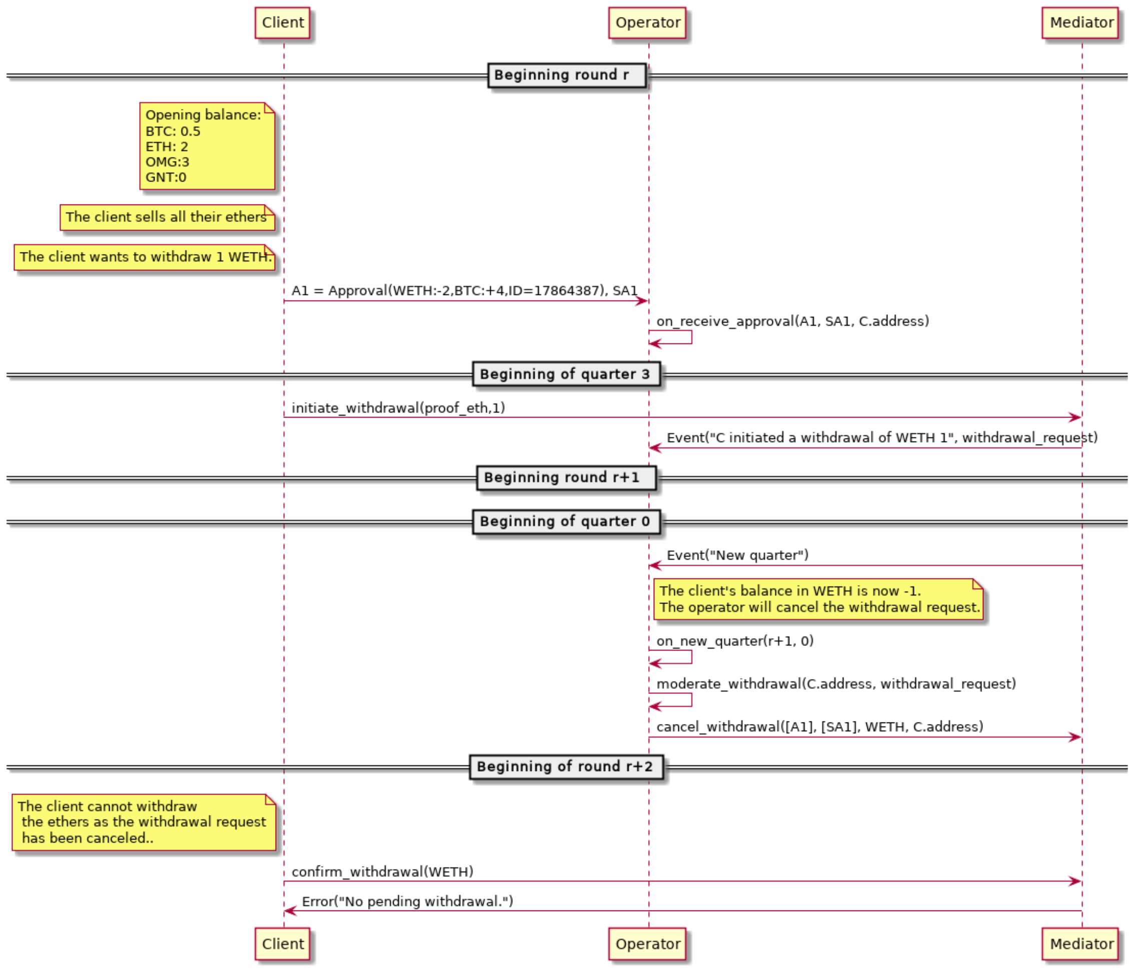
Client
Operator
Mediator
Beginning round r
Opening balance:
BTC: 0.5
ETH: 2
OMG:3
GNT:0
The client sells all their ethers
The client wants to withdraw 1 WETH.
A1 = Approval(WETH:-2,BTC:+4,ID=17864387), SA1
on_receive_approval(A1, SA1, C.address)
Beginning of quarter 3
initiate_withdrawal(proof_eth,1)
Event("C initiated a withdrawal of WETH 1", withdrawal_request)
Beginning round r+1
Beginning of quarter 0
Event("New quarter")
The client's balance in WETH is now -1.
The operator will cancel the withdrawal request.
on_new_quarter(r+1, 0)
moderate_withdrawal(C.address, withdrawal_request)
cancel_withdrawal([A1], [SA1], WETH, C.address)
Beginning of round r+2
The client cannot withdraw
the ethers as the withdrawal request
has been canceled..
confirm_withdrawal(WETH)
Error("No pending withdrawal.")
Client
Operator
Mediator

Figure 11: **A malicious client tries to withdraw some tokens**

If the client is dishonest (see Figure 11) they may try to spend the tokens off-chain before submitting an on-chain withdrawal request. When this happens the operator must cancel the withdrawal by providing the necessary evidence to the mediator that the client attempted to double-spend. This will result in removing the withdrawal request from the state of the mediator, thus making it impossible for the client to obtain the tokens.

2.5.5 The client opens a dispute

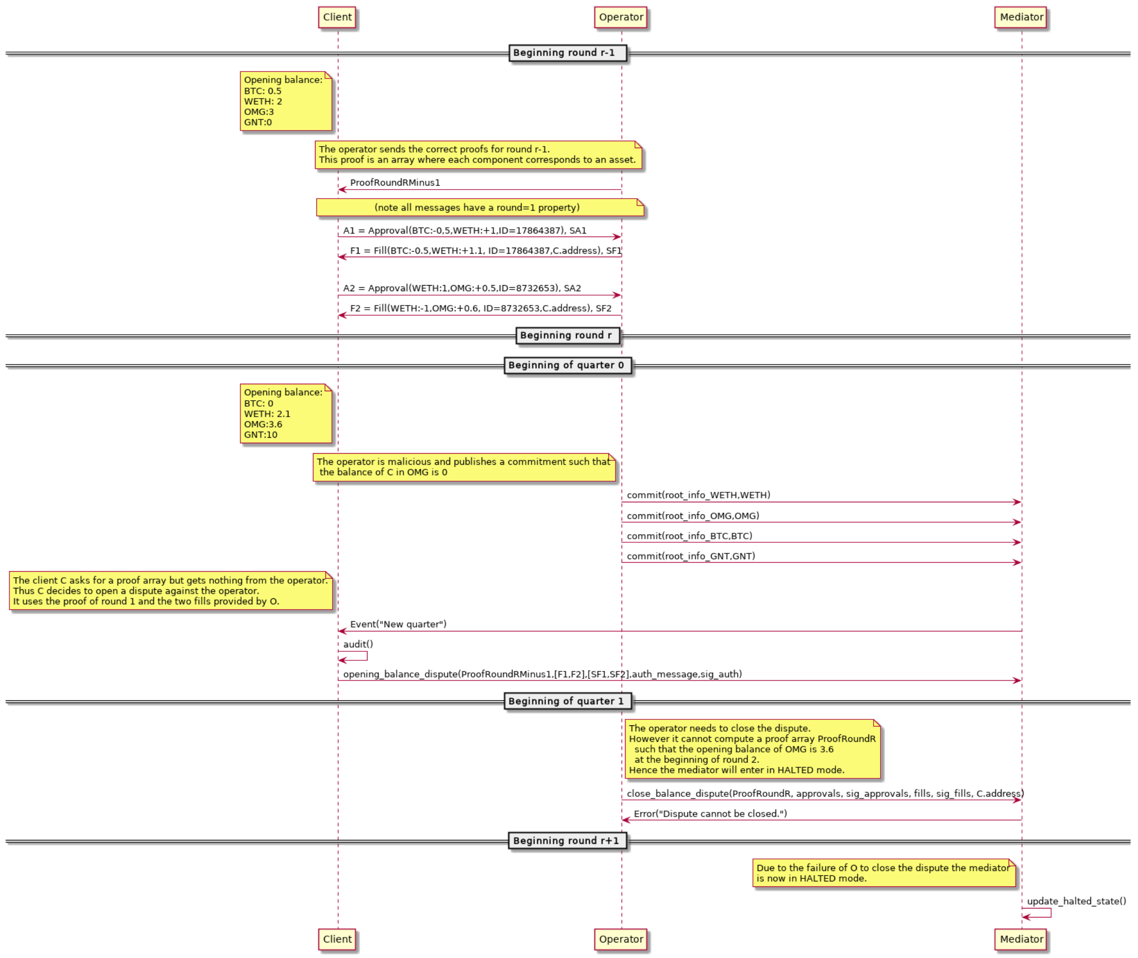
Client
Operator
Mediator
Beginning round r-1
Opening balance:
BTC: 0.5
WETH: 2
OMG:3
GNT:0
The operator sends the correct proofs for round r-1.
This proof is an array where each component corresponds to an asset.
ProofRoundRMinus1
(note all messages have a round=1 property)
A1 = Approval(BTC:-0,5,WETH:+1,ID=17864387), SA1
F1 = Fill(BTC:-0.5,WETH:+1.1, ID=17864387,C.address), SF1
A2 = Approval(WETH:1,OMG:+0.5,ID=8732653), SA2
F2 = Fill(WETH:-1,OMG:+0.6, ID=8732653,C.address), SF2
Beginning round r
Beginning of quarter 0
Opening balance:
BTC: 0
WETH: 2.1
OMG:3.6
GNT:10
The operator is malicious and publishes a commitment such that the balance of C in OMG is 0
commit(root_info_WETH,WETH)
commit(root_info_OMG,OMG)
commit(root_info_BTC,BTC)
commit(root_info_GNT,GNT)
The client C asks for a proof array but gets nothing from the operator.
Thus C decides to open a dispute against the operator.
It uses the proof of round 1 and the two fills provided by O.
Event("New quarter")
audit()
opening_balance_dispute(ProofRoundRMinus1,[F1,F2],[SF1,SF2],auth_message,sig_auth)
Beginning of quarter 1
The operator needs to close the dispute.
However it cannot compute a proof array ProofRoundR such that the opening balance of OMG is 3.6 at the beginning of round 2.
Hence the mediator will enter in HALTED mode.
close_balance_dispute(ProofRoundR, approvals, sig_approvals, fills, sig_fills, C.address)
Error("Dispute cannot be closed.")
Beginning round r+1
Due to the failure of O to close the dispute the mediator is now in HALTED mode.
update_halted_state()
Client
Operator
Mediator

Figure 12: **Dispute resolution**
The client opens a dispute and, as the operator is malicious, the dispute cannot be closed.

At the beginning of each round during the second quarter the client will ask the operator for a proofs array showing that the correct balances have been committed. If the array is not sent or is invalid, the client will open a dispute (see round 2 of Figure 12). This dispute contains the opening balance of the previous round and also some potential trading data in the form of fills. Within a quarter after the dispute has been opened, the operator needs to close the dispute by providing the proof for the balances questioned by the client and potential additional trading activity (fills that were not sent to the client). A security guarantee expected by the protocol is that a dishonest operator will not be able to close a legitimate dispute. In this case the mediator will enter in `HALTED` mode within the next round.

2.5.6 Mediator enters in `HALTED` mode

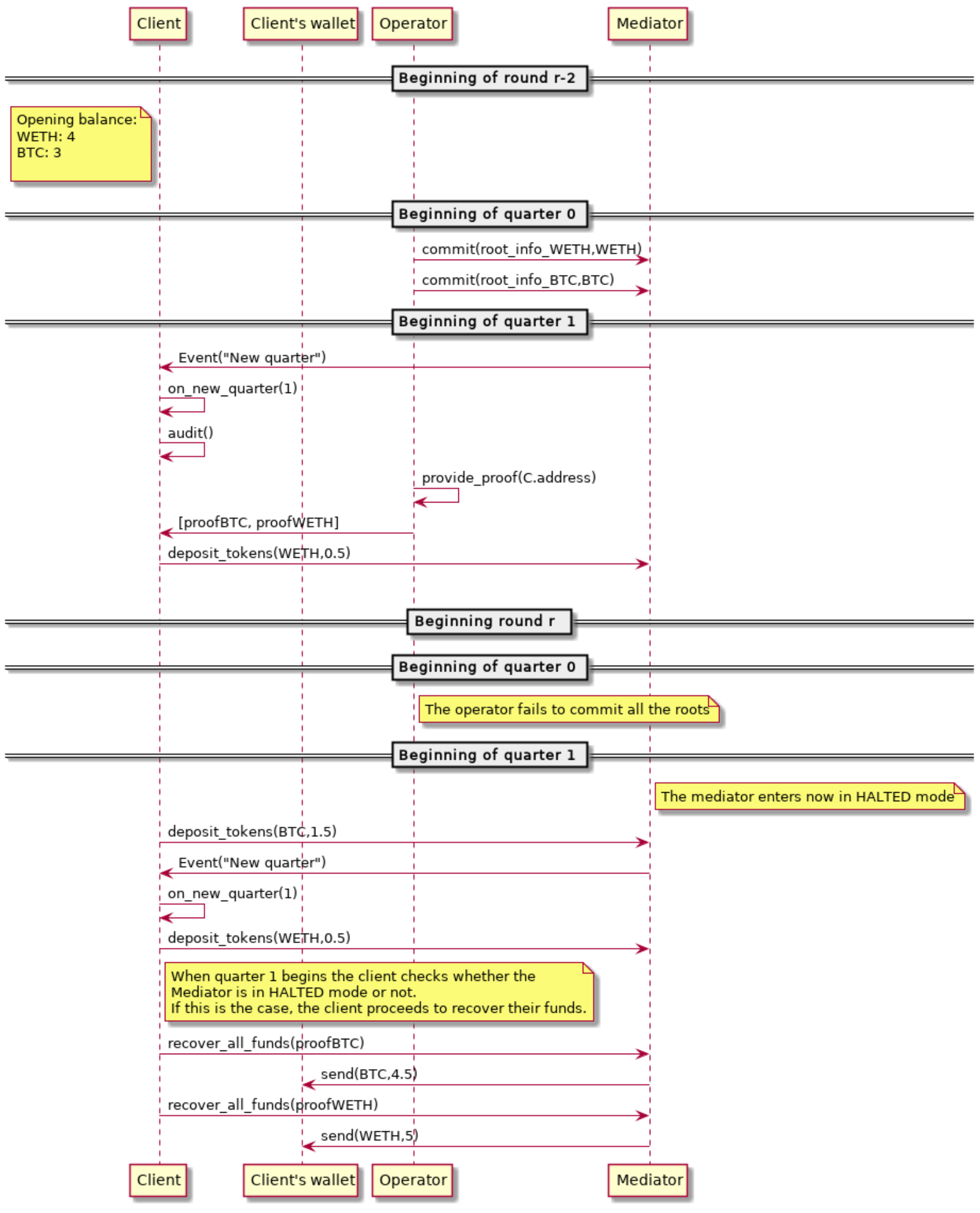

Figure 13: **Mediator enters in `HALTED` mode.**

As mentioned in the previous section, a dishonest behavior from the operator may put the mediator in `HALTED` mode. In Figure 13 we can see that if the operator does not commit the roots for all assets by the end of the first quarter of each round, the mediator will also enter in `HALTED` mode. The client must monitor during the second quarter of each round the mediator is in `HALTED` mode.

If this is the case the client must execute `mediator.recover_all_funds()` for each token. By doing so the client will receive their token corresponding to the opening balance of round $r-2$ where r is the round when the state of the mediator shifted to `HALTED` mode. All the deposits made during rounds $r-2$,$r-1$ and r will be also returned. If the client joined the L2X instance just before the `HALTED` mode is activated, they may not be in possession of a proof. In this case the client will call `mediator.recover_on_chain_funds_only()` in order to get their recent deposits back.

2.6 Security

2.6.1 Permissions

The table below describes when each function of the mediator smart contract can be invoked and by whom.

Function	notHalted	onlyBy(operatorAddress)	Private
`deposit_tokens`	x		
`initiate_withdrawal`	x		
`cancel_withdrawal`	x	x	
`confirm_withdrawal`			
`open_balance_dispute`	x		
`close_balance_dispute`	x	x	
`commit`	x	x	
`recover_all_funds`			
`recover_on_chain_funds_only`			
`update_halted_state`			x
`get_current_round`			x
`get_current_quarter`			x

2.6.2 ERC20 Token Requirements

Depending on how the smart contract is implemented, care needs to be taken off-chain to avoid problems like overflow and division by zero. Simply failing transactions on chain in these cases is not enough because the off chain actions may require the on chain TX to go through to continue operation.

As a simple example, imagine the operator trying to close a balance dispute. If this fails due to division by zero, overflow or rounding errors in the price calculations the dispute can't be closed and the mediator will halt. Hence the operator needs to be careful not to accept (or fill) any approval that could lead to this.

To err on the side of caution, it is advisable to only register tokens that satisfy the following requirements.

- The token must have 18 decimals.
- The token must have fixed supply.
- The token must have a total supply below 10^{20}.

The maximum supply number is chosen such that multiplying two amounts (with maximum value $10^{18} \cdot 10^{20} = 10^{38}$ each) will not cause a overflow with 256 bit unsigned integers. Note that most tokens have a total supply far below 10^{20}.

2.7 Limitations and Non-Goals

2.7.1 Solved

- Custody of funds. Funds are kept in smart contract where they are managed by the rules of the protocol.
- Compromised/malicious/defective operator cannot move clients' funds without the clients' prior approval.
- Withdrawals are no longer at the mercy of the exchange. The clients can withdraw all funds from confirmed rounds if they don't have any open approvals.
- Performance. All trading is done off-chain and is not affected by the performance of the underlying blockchain.

2.7.2 Not Solved

- This is not a decentralized solution.
- This not a system that is resistant to shutdown . In fact, it is shut down if the operator tries to cheat. The operator can also choose to shut down at any time.
- The operator may refuse service or any part thereof to any or all the users. Notably the operator has the power to refuse placing orders, cancellation, joining at any time to any party.
- Front-running. In case orders are over-matched, a malicious exchange could pocket the difference. The exchange has full flexibility on how to match orders provided it respects the price and quantities specified in the approvals.
- Trustless off-chain cancellation.

2.7.3 Limitations

- Orders are only valid until the end of the round they were created in.
- Orders do not have a customizable expiry. In the future we may consider different approaches to solve liquidity problems at the beginning of a round. For example clients may pre-sign and pre-submit approvals for the next round. These "future" approvals will be filled only if the corresponding "past" approvals expired.
- Order cancellation is not implemented in a trustless fashion. Orders cannot be canceled against a non-cooperative operator. If the operator is compromised all approvals that were collaboratively canceled in the current round could still be filled by the operator until the end of the round.
- Orders cannot be modified. However this can be accomplished by canceling and recreating an order with the desired parameters.
- The client must be online some time between the beginning of the second quarter and the end of the fourth quarter for each round. Otherwise the client takes the risk of being assigned a wrong opening balance without having the opportunity to dispute it.

- In the trustless regime the client is limited in the number of approvals they can create per round [10]. Indeed if too many fills (resp. approvals & fills) are used in the contract call to open (resp. close) a dispute, the transaction may run out of gas or the payload may exceed the 32kB limit set by go-ethereum (*geth*) [11]. In practice this means that honest clients may lose some of their funds if they have created too many approvals and the operator turns malicious. Similarly, if the operator creates too many fills for a user the mediator may enter `HALTED` mode, even though the operator is honest.

References

[1] Philippe Camacho. *Predicate-Preserving Collision-Resistant Hashing*. PhD thesis, 2013. Available at `http://users.dcc.uchile.cl/~pcamacho/papers/phdthesis.pdf`.

[2] Gaby G Dagher, Benedikt Bünz, Joseph Bonneau, Jeremy Clark, and Dan Boneh. Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 720–731. ACM, 2015. Available at `http://www.jbonneau.com/doc/DBBCB15-CCS-provisions.pdf`.

[3] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. The arwen trading protocols. 2018. Available at `https://www.arwen.io/whitepaper.pdf`.

[4] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical report, 2017. Available at `https://www.ideals.illinois.edu/bitstream/handle/2142/97207/hildenbrandt-saxena-zhu-rodrigues-guth-daian-rosu-2017-tr.pdf`.

[5] Yoichi Hirai. Formal verification of deed contract in ethereum name service, 2016. Available at `https://yoichihirai.com/deed.pdf`.

[6] Kexin Hu, Zhenfeng Zhang, and Kaiven Guo. Breaking the binding: Attacks on the merkle approach to prove liabilities and its applications. 2018. https://eprint.iacr.org/2018/1139.pdf.

[7] Rami Khalil and Arthur Gervais. Nocust - a non-custodial 2nd-layer financial intermediary. Cryptology ePrint Archive, Report 2018/642, 2018. Available at `https://eprint.iacr.org/2018/642/20180706:123923`.

[8] Rami Khalil, Arthur Gervais, and Guillaume Felley. Nocust – a securely scalable commit-chain. Technical report, Cryptology ePrint Archive, Report 2018/642, 2018. Available at `https://eprint.iacr.org/2018/642/`.

[9] Rami Khalil, Arthur Gervais, and Guillaume Felley. Tex - a securely scalable trustless exchange. Cryptology ePrint Archive, Report 2019/265, 2019. Available at `https://eprint.iacr.org/2019/265`.

[10] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*, 2018. Available at `https://arxiv.org/abs/1801.00687`.

[11] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. 2018. Available at `http://ilyasergey.net/papers/temporal-isola18.pdf`.

[10] The actual number remains to be determined through the implementation.

[11] We plan to solve this with a form of aggregate data structure that the client and operator can negotiate collaboratively.

[12] Ron van der Meyden. On the specification and verification of atomic swap smart contracts. 2018. Available at `https://arxiv.org/abs/1811.06099`.

[13] Daniel Wang, Jay Zhou, Alex Wang, and Matthew Finestone. Loopring: A decentralized token exchange protocol. Technical report, 2018. Available at `https://loopring.org/resources/en_whitepaper.pdf`.

[14] Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain. 2017. Available at `https://github.com/0xProject/whitepaper`.

[15] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J Knottenbelt. Xclaim: Interoperability with cryptocurrency-backed tokens. Technical report, 2019. Available at `https://eprint.iacr.org/2018/643`.

A Security analysis

A.1 Attacks

In this section we describe attacks that we have discovered while analysing the L2X protocol. All these attacks are mitigated yet they are worth documenting as they put some light on specific parts of our design.

A.1.1 Forging opening balances

At the beginning of each round the operator must publish a value representing all the balances of the customer (see Section 2.2.4). For efficiency reasons, this value is short and is by itself useless. Only when combined with a proof (see Section 2.1.2.7) it is possible for a client to verify that their balance has been computed correctly. However a malicious operator could (1) not send the proof, (2) send an invalid proof or (3) send a valid proof but corresponding to an incorrect balance.

This attack is mitigated through the mechanism of disputes (see Section 2.2.8 and 2.2.9) where the client can rely on the mediator to establish the correct opening balance.

A.1.2 Malicious Operator can cancel legitimate withdrawals when picking a subset of approvals

As we need to enable processing the approvals asynchronously, the operator can pick an arbitrary subset of approvals for computing the fills. This enables an attack we illustrate in the following example.

- The client initiates a withdrawal request at round r for `1 ETH`.
- The opening balance at round $r + 1$ is `10 ETH, 10 BTC`.
- During round r+1 the client sends the following signed approvals:
  - `(-8ETH , + 5 BTC, id = 1) // Balance:  2 ETH, 15 BTC`
  - `(+5ETH , - 5 BTC, id = 2) // Balance:  7 ETH, 10 BTC`
  - `(-4ETH , + 5 BTC, id = 3) // 3 ETH, 15 BTC`

All the approvals, if processed sequentially, are legitimate and thus the withdrawal should be confirmed successfully during round $r + 1$.

However if the operator only picks the approvals with id=1 and id=3 we have:

- `-8ETH , + 5 BTC, id = 1 // Balance:  2 ETH, 15 BTC`
- `-4ETH , + 5 BTC, id = 3 // Balance:  **-2 ETH, 20 BTC**`

The balance of ETH is overdrawn and thus the (malicious) operator can cancel the withdrawal.

This is not a devastating attack as the honest client could wait one more round, make a new withdrawal request and avoid trading. By doing so that the withdrawal request cannot be canceled.

However it seems to indicate that enabling the operator to pick any arbitrary subset of approvals yields unexpected behavior.

A.1.3 Commit and Cancel Withdrawal

As we can see in Listing 21, the closing dispute algorithm of the mediator must consider possible withdrawal requests in the disputed round in order to compute the expected balance.

However this introduces the following problem: Assume a (honest) client makes a withdrawal request during round r. Assume also that during the same round the client sends some approvals to the operator. These approvals are such that they contain a subset that can yield the cancellation of the request (see Section A.1.2).

During round $r + 1$, the operator computes and commits the new balances as if no approval has been filled. The operator sends the proof to the client who successfully verifies it.

However the malicious operator now cancels the withdrawal request made during the previous round r. As a result the new balance for round $r + 1$ is such that the withdrawal amount is subtracted, yet the client will not be able to confirm the withdrawal any more.

This attack is prevented by allowing the withdrawal cancellation only during the same round of the request or just before the operator commits (see Listing 17).

A.1.4 Steal client's first deposit

If we force a honest client Veronica to provide a proof in `mediator.open_balance_dispute` then the following attack can be executed by the operator and a colluding client Simon.

- Veronica joins the protocol at some round r.
- She deposits 100 WETH into the mediator.
- Simon that has 0 WETH at round r.
- The operator transfers the 100 WETH to Simon.
- At round $r + 1$, the operator commits the root where Veronica has a balance of 0 WETH, and Simon has a balance of 100 WETH.
- The operator does not send the proof to Veronica.

As Veronica does not have a proof of her balance, she is unable to open a balance dispute.

Hence it must be possible for a client to call `mediator.open_balance_dispute`, even without a proof.

A.1.5 DoS the Operator by opening a dispute that cannot be closed

The function `mediator.open_balance_dispute` (see Listing 19) allows a client to open a dispute without the need of a proof. However if any user is able to open a dispute then a malicious client could force the mediator to enter in `HALTED` mode as follows:

- Do not join the protocol.

- Wait for the operator to invoke `mediator.commit` for all tokens.
- Call `mediator.open_balance_dispute` at the beginning of round r.
- The operator cannot close the dispute because the Client's address – being unknown to the operator – has not been considered in the computation of the commitments.
- The mediator enters in `HALTED` mode during round $r + 1$.

In order to prevent this attack, we force the client to submit an authorization message (see Section 2.1.2.6) signed by the operator. This signed message is an evidence that the client is allowed to participate to the protocol.

A.1.6 Double spend (online/offline)

Assume a client has performed a number of trades during round r such that their opening balance for token $WETH$ goes from 100 to 50. Assume moreover that the client initiates a withdrawal request during the same round (after trading) of 100. The mediator cannot block the request as the trading transactions are not present on the blockchain.

To prevent this attack the mediator needs to invoke `mediator.cancel_withdrawal` (see Section 2.2.6) with the list of approvals in order to prove that the client is trying to double spend their tokens.

A.1.7 Withdraw-then-recover attack

If we do not freeze time, then a malicious client (colluding with the malicious operator) is able to obtain twice the amount they deposited. The attack works as follows:

- **Round 0**
  - Malicious client Eve deposits 10 WETH.
  - Victim client Bob deposits 10 WETH.
- **Round 1**
  - The operator commits to the new roots.
  - The operator sends proof π_1 to Eve corresponding to her opening balance at the beginning of round 1.
  - The operator sends proof π'_1 to Bob corresponding to their opening balance at the beginning of round 1.
- **Round 2**
  - The operator commits to the new roots.
  - The operator sends π_2 to Eve the proof corresponding to the opening balance at the beginning of round 2 which is still 10 WETH.
  - The operator sends proof π'_2 to Bob corresponding to their opening balance at the beginning of round 2.
  - Eve initiates a withdrawal of 10 WETH using π_1.
- **Round 3**
  - The operator commits to the new roots.

- **Round 4**
  - The operator does not commit any root. Thus after quarter 0, the mediator enters in `HALTED` mode.
  - Eve confirms her withdrawal and obtains 10 WETH.
  - Eve invokes `mediator.recover_all_funds(`π_2`)` and obtains 10 WETH (once again).

With time freeze, the operator missing the commit in round 4 would make round 1 the last confirmed round (see Section 2.2.1). Since withdrawal is requested in round 2, it is newer than the last confirmed round, making it impossible for Eve to withdraw, and the attack is thwarted.

Note however that even with this mitigation a similar attack would be possible if we would allow `mediator.confirm_withdrawal` to be executed during quarter 0 (when it is not clear whether the mediator is in `HALTED` mode or not.) Thus it is also fundamental to forbid withdrawal confirmations during the first quarter. This is achieved by `mediator.confirm_withdrawal` (see Listing 18, line 22).

A.1.8 Malicious Operator can cancel any withdrawal by committing a tree with two different leaves for a user

Context: In the mediator functions `initiate_withdrawal` and `cancel_withdrawal` the client (resp. the operator) *was* required[12] to provide a proof π. The purpose of this proof is to establish the opening balance for the client before initiating the withdrawal. It is assumed that both proofs are the same, yet the operator could provide a proof π' different from π.

The attack: the idea of the attack consists of building a Merkle tree where two leaves (instead of one) are assigned to a potential victim Alice. One leaf of the tree will contain the correct balance of Alice. The other leaf will contain the balance 0. Alice will be given the proof corresponding to the correct balance (let us call it π). The malicious operator will keep the proof corresponding to the other leaf with balance 0, π' for himself. After the withdrawal is initiated by Alice, the malicious operator will invoke `cancel_withdrawal` with the proof π' that corresponds to the empty balance of Alice. By doing so the withdrawal will be canceled, as according to the proof π', Alice has no funds available (we assume for simplicity that no off-line trading happen).

Note that while there is a "fake" leaf with balance 0, the operator will always be able to commit successfully the root of the tree because the total balance remains unchanged.

Here a concrete example of the attack:

- Round 1
  - Alice deposits 100 ETH.
- Round 2
  - Operator creates a tree with the following leaves `[Alice:100, Bob:50, Zoe:10, Alice:0]`.
  - Operator sends π to Alice where π corresponds to the first leaf of the tree (from the left).
- Round 3
  - Alice initiates a withdrawal of 100 ETH using π.
  - Operator invokes `cancel_withdrawal` with π', the proof that corresponds to the right-most leaf of the tree.

[12]See first version of the NOCUST paper [7].

   ◦ As the balance of the leaf is 0 the withdrawal is successfully canceled.

Mitigation: the attack is prevented as follows: the client provides the proof for the balance, and the mediator stores it. This ensures that the balance used is agreed upon by the client and operator.

What happens with disputes: One may think that a similar problem could happen with disputes. However here the situation is different because the proofs used in `open_balance_dispute` and `close_balance_dispute` correspond to two different rounds. Indeed the opening balance of the client is locked when the dispute is open and the operator does not have the opportunity to use the trick above.

A.1.9 Failed recovery attack

If we only allow clients to recover their funds using `mediator.recover_all_funds` then the following problem occurs. Assume a client joins at round r, deposits some tokens and then during the same round or the next, the Mediator enters in `HALTED` mode. Now the client is not able to recover their deposit because they hold no proof. This is why we introduce the function `mediator.recover_on_chain_funds_only` so that the clients without proof can also recover their deposits.

A.1.10 Disputing a single asset at a time is insecure

If we allow to challenge only one asset at a time, then a malicious operator can close a dispute despite having committed to an inconsistent set of opening balances.

- Assume there are only two assets say ETH and BTC. The opening balance of the client for round $r-1$ is `(ETH:0, BTC:1)` There is a single approval of the client during round $r-1$: `ETH:+5, BTC:-1, ID:1, intent:"sellAll"`.
- Assume the malicious operator sets the opening balance at round r of the client to `(ETH:0, BTC:0)` and does not send any proof nor fills to the client. Clearly there is something wrong as the client should have either 1 BTC or 5 ETH or more.
- So the client opens a dispute for the ETH asset. As there are no fills, the operator is not forced to provide any approval and can thus close the challenge because the initial balance in ETH for round $r-1$ is 0.
- Then the client challenges the BTC asset. Here the malicious operator can produce a fill for the approval and also close the challenge for the BTC asset.

Because of this attack, our protocol ensures that when a client opens a dispute they challenge all the tokens at once (see Listing 19). Similarly, an honest operator closing a dispute will have to provide evidences for all the tokens (see listings 20, 21, 22).

A.1.11 Replay attack

In the case several instances of the L2X protocol are running, some additional attacks are possible. We describe here the *replay attack* which consists of reusing some messages of one instance into another. For example a malicious client may take some fills from instance A and use them into instance B so that the operator of instance B cannot close a dispute. Similarly, a malicious operator may use some approvals from instance A into instance B in order to decrease artificially the balance of a client that is trading in both instances. The mitigation for this attack consists of adding a field *instanceId* inside the messages for approvals and fills and set this field to the address of the contract. In order to keep the presentation simpler, we do not specify the details of this mitigation here and thus assume that only one instance of the protocol is running.

A.2 State diagrams

Analyzing the security of blockchain protocols involving smart contract logic and cryptographic primitives is a challenging task. Ideally we would like to automate the verification of the correctness and security of such a construction leveraging formal methods techniques [10, 12, 4, 5, 11]. Given the complexity of the protocol, we leave such approach for future work. Nonetheless in order to make a first step in that direction we provide an initial argumentation of four security properties expected by our protocol. These properties illustrate how honest players (whether the operator or some client) can find a strategy in order to protect their funds or the integrity of their operations.

A.2.1 Notations

We use *Finite State Machines* in order to model and reason about the behavior of the participants. In the following we describe the notations used to describe the states and transitions.

Each state of the diagrams in figures 14 15 is composed by a tuple (R, Q, H, P) where:

- R: is the current round.
- Q: is the current quarter.
- H: the mediator state: "active" or "halted"
- P: optional argument. Whether the clients holds a proof or not: "proof" or "no-proof".

For example a state: "r:0 active proof" corresponds to the execution of the protocol where the current round is r, the quarter is 0, the mediator is not in `HALTED` mode and the client owns a valid proof corresponding to the opening balance of round r.

We consider the following transitions:

- `NextRound`: the mediator goes to the next round.
- `NextQuarter`: the mediator goes to the next quarter.
- `HALTED` : the mediator goes into `HALTED` mode.
- `C.recover_all_funds`: the client invokes `mediator.recover_all_funds` with the right arguments.
- `C.init_withdrawal`: the client invokes `mediator.init_withdrawal` with the right arguments.
- `O.cancel_withdrawal`: the operator invokes `mediator.cancel_withdrawal` with the right arguments.
- `C.confirm_withdrawal`: the client invokes `mediator.confirm_withdrawal` with the right arguments.
- `C.open_balance_dispute`: the client invokes `mediator.open_balance_dispute` with the right arguments.
- `O.close_balance_dispute`: the operator invokes `mediator.close_balance_dispute` with the right arguments.
- `O.update_balance`: the operator updates the local balance of a client based on some event on the blockchain.

A.2.2 Funds recovery

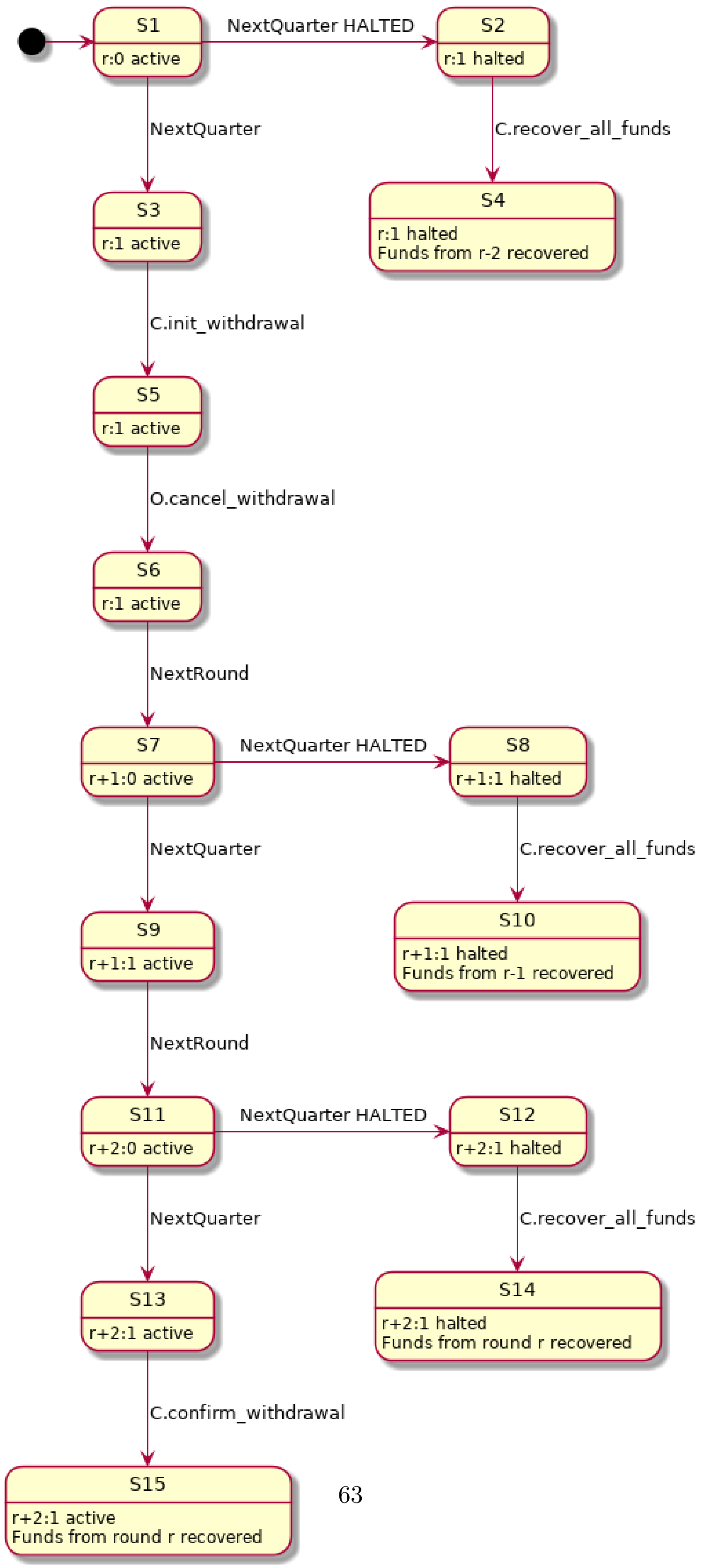

Figure 14: **A honest client can get back their funds.**

One of the main requirement of our construction is that a client should always be in control of their funds. In practice this means that at any time the client must be able to recover their tokens. The property below states that if the client is in possession of valid proofs for rounds $r-2$, $r-1$, r then there is a strategy that enables the client to recover their balance for the some of the rounds listed above.

Property 1. *If a honest client holds valid proofs $\pi_{r-2}, \pi_{r-1}, \pi_r$ at the beginning of round r, then there exists a strategy that enables him to recover* $\texttt{balance}(r-2)$*, or* $\texttt{balance}(r-1)$*, or* $\texttt{balance}(r)$*.*

Proof. (Sketch) The strategy depicted in Figure 14 consists in initiating a withdrawal of the whole opening balance during the second quarter of round r (transition $S3 \to S5$). In the case the contract enters in `HALTED` mode, then the client can invoke `mediator.recover_all_funds` and get their tokens from round $r-2$ (transition $S2 \to S4$). Then a malicious operator may try to cancel the withdrawal (transition $S5 \to S6$) but as the client is honest the withdrawal will not be canceled. Then the client will wait and possibly recover their funds of round $r-1$ in the case the contract enters in `HALTED` mode (transition $S8 \to S10$). Finally if the during round $r+2$ the contract does not enter in `HALTED` mode the client will be able to confirm the withdrawal made during round r (transition $S13 \to S15$). Otherwise they will invoke `mediator.recover_all_funds` and recover their opening balance of round r as well (transition $S12 \to S14$). □

Note that the client has no control on which balance will be recovered and hence the balance changes during rounds $r-1$ and round r cannot be guaranteed.

A.2.3 Correctness of balances

The property below states that if a round r is confirmed (which means the mediator did not enter in `HALTED` mode during the first quarter of round $r+1$), then the balance available in the mediator is enough to cover all the balances of the honest clients.

Property 2. *For any confirmed round r, we have $\sum_{i \in I} \texttt{balance}_i(r) \leq \texttt{balance}_{\mathcal{M}}(r)$, where I is the set of indices of honest clients, $\texttt{balance}_{\mathcal{M}}(r)$ is the total amount of tokens held by the mediator, part of the committed value of round r, and $\texttt{balance}_i(r)$ is the amount of tokens agreed between an honest client with index $i \in I$ and the operator at the beginning of round r. We assume that the operator may be dishonest and that all honest clients verify the correctness of their balance at the beginning of each round.*

Proof. (Sketch) If round r is confirmed this means that all the honest clients verified their proof during round r. Moreover the operator was able to close all the disputes, so this means that for each client there is a proof that passes the verifications of `mediator.close_balance_dispute`. Each proof thus contains the correct balance $\texttt{balance}_i(r)$ for each honest client $i \in I$. On the other hand the proof is verified against an augmented Merkle tree such that with overwhelming probability $\sum_i \texttt{balance}_i(r) \leq \texttt{balance}_{\mathcal{M}}(r)$ (see Proposition 1).

□

Note that this property does not discard front-running attacks where the operator would be able to place orders with the guarantee of making profit. Indeed the operator may control one or more of the clients in order to perform these trades.

A.2.4 Client owns a valid proof

Property 1 requires the client to own valid proofs for the current and the two previous rounds. While we cannot guarantee that the operator will send such valid proof for each new round r to the client, we can show that if it does not then the client can recover their funds from previous rounds $r-1$ or $r-2$. Hence either the client obtains a new proof which renews their ability to recover their funds in the future, or they will be able to recover funds from previous rounds.

Property 3. *If a honest client holds valid proofs π_{r-1}, π_{r-2} at the beginning of round r, then the honest client can obtain a valid proof π_r for round r or has a strategy to recover either* `balance`$(r-1)$ *or* `balance`$(r-2)$*.*

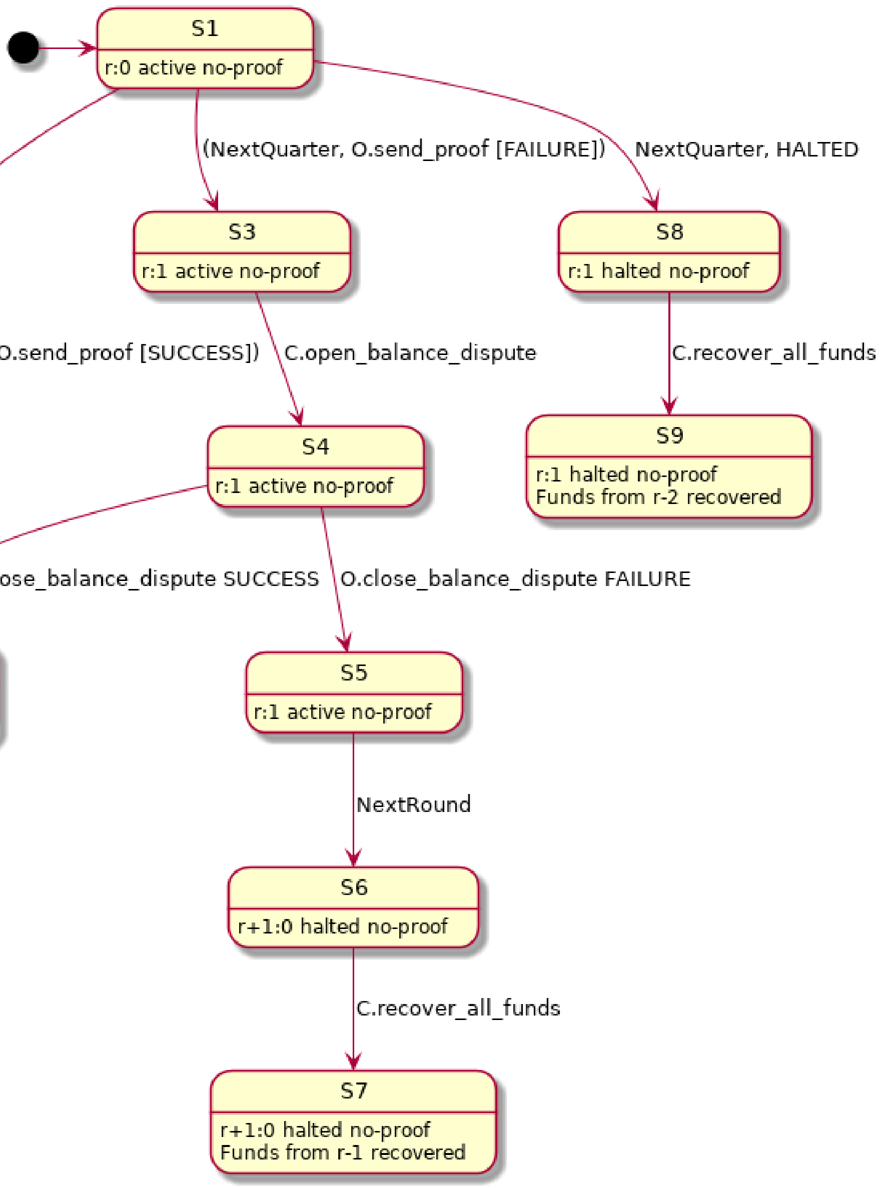

Figure 15: **A honest client can get a valid proof or recover their funds.**

Proof. (Sketch) The strategy described in Figure 15 consists of opening a balance dispute in case the proof received is not valid (transition $S3 \rightarrow S4$). If the contract enters in `HALTED` mode anyways (e.g.: the operator does not commit to some root) then the client can invoke `mediator.recover_all_funds` and get their balance from round $r-2$. (transition $S8 \to S9$). The the operator may try to close the balance. If the operator is successful in doing so (transition $S4 \to S2$) then the client will be able to fetch the valid proof from the mediator smart contract. Otherwise (transition $S4 \to S5$) the client will wait for the next round when the contract will enter in `HALTED` mode and recover their funds from round $r-1$ (transition $S6 \to S7$). Finally recall that when the mediator enters in `HALTED` mode the time gets frozen. Thus all the clients will recover funds from the same round. □

A.2.5 Honest operator can avoid `HALTED` mode

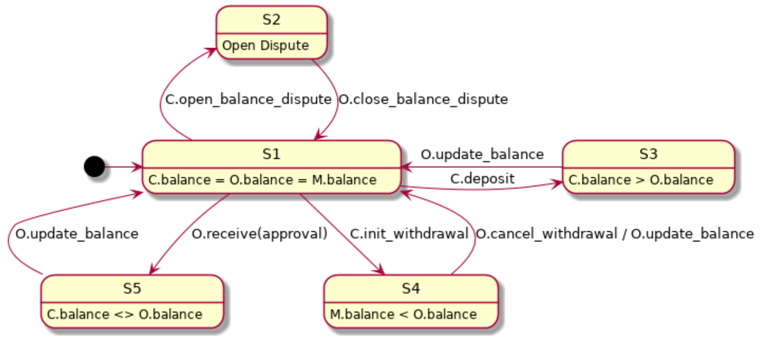

Figure 16: **A honest operator can always prevent the mediator from going into `HALTED` mode.**

While the main objective of the protocol is to protect the funds of clients, our solution must also ensure that a honest operator will be able to keep functioning despite the presence of malicious clients. In particular the property below shows that a honest operator can ensure that the mediator smart contract will never enter in `HALTED` mode.

Property 4. *If at the beginning of round r (quarter 0) the balances of the clients held by the operator and the balance of the mediator are consistent, and the mediator is not in* `HALTED` *mode, then there exists a strategy for the honest operator to keep* `HALTED` *mode = false at the beginning of round $r+1$.*

Proof. (Sketch) The strategy depicted in Figure 16 consists in monitoring the deposits / withdrawals made by the clients as well as the open challenges. Each time a client makes a deposit (transition $S1 \to S3$) the operator will update the corresponding balance (transition $S3 \to S1$). If a client initiates a withdrawal (transition $S1 \to S4$), the withdrawal will be cancelled if illegitimate or the corresponding balance will be updated (transition $S4 \to S1$). If the client sends an approval (transition $S1 \to S5$) to the operator and the operator matches the corresponding order, the operator will update the balances (transition $S5 \to S1$) of the client accordingly and also send back the corresponding fill. Moreover the (signed) approval will enable the operator to successfully close a a dispute open by the client even though this dispute contains the fill. This way an honest operator is always able to come back to state $S1$ where the balances it handles and the balance managed by the mediator / clients are consistent. Moreover the operator can always reach state $S1$ within one quarter and thus will be able to successfully close disputes when needed (transitions $S1 \leftrightarrow S2$).

□

A.2.6 A client cannot extract more than their balance.

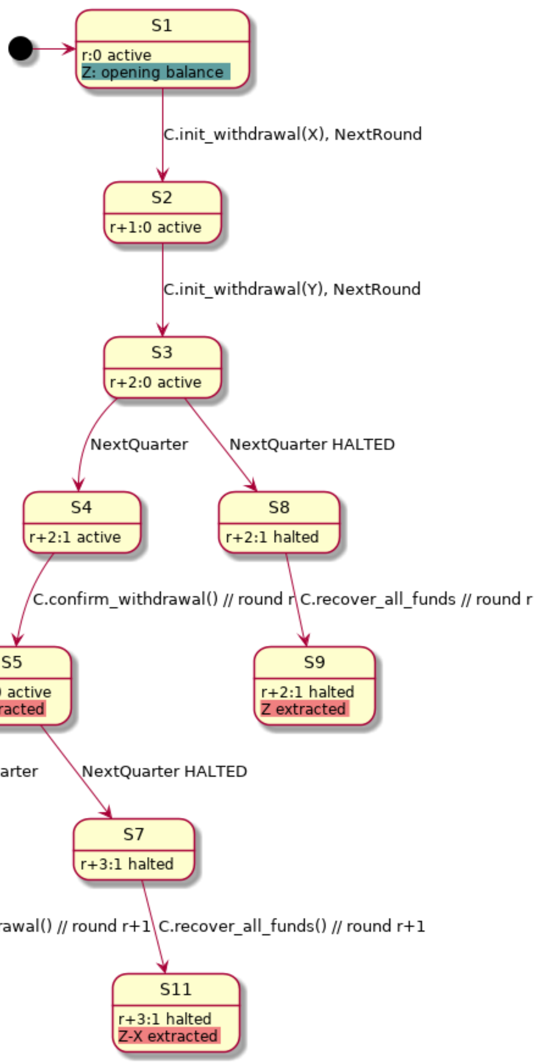

Figure 17: **No more than the client's balance can be moved outside the mediator.**

There are two ways for a client to "extract" tokens from the mediator: making a successful call to `mediator.confirm_withdrawal` or `mediator.recover_all_funds`[13]. As shown in Section A.1.7, a malicious client may abuse these functions in order to extract more tokens than they own according to their balance.

In the following we show that for a confirmed round r a malicious client who may collude with the operator is not able to extract more than their balance `balance`(r) from round $r + 2$ on.

This property is important as it ensures that when balances are computed correctly (see property 2) there will be always enough tokens in order to allow honest clients to recover their funds.

Property 5. *Let r be a confirmed round and* `balance`(r) *the corresponding opening balance for a potentially malicious client colluding with the operator. Assume that the client does not make any*

[13]For the sake of clarity we omit the edge case `mediator.recover_on_chain_funds_only`.

deposit during rounds $r, r+1, ...$ then we have that

$$\texttt{balance}(r) \geq \texttt{withdrawn}(r+2) + \texttt{withdrawn}(r+3) + \texttt{recovered}(r+2) + \texttt{recovered}(r+3)$$

where $\texttt{withdrawn}(r)$ *and* $\texttt{recovered}(r)$ *are the amounts successfully withdrawn (i.e. with confirmation) resp. recovered by the client during round r.*

Proof. (Sketch) In order to show that the client cannot extract more than their balance let us examine the state diagram in Figure 17. At the beginning of round r the client's opening balance is of Z tokens (state $S1$). As round r is confirmed, this means that the contract will remain active at least until the beginning of round $r+2$. During round r and $r+1$ the client may initiates withdrawals of amounts X, Y respectively where $X \leq Z$ and $Y \leq Z$. Given that only one withdrawal can be active at a time we have that either $X = 0$ or $Y = 0$[14]. At the beginning of round $r+2$ the mediator may enter in `HALTED` mode (state $S8$). In this case the client can recover their funds from round r (state $S9$). **Yet it is important to stress that the client cannot indeed confirm the withdrawal made during round r nor $r+1$ as the timeline is frozen to round $r+2$, quarter** 0. In the other case where the mediator is still active (state $S4$) and the client can confirm the withdrawal corresponding to round r (state $S5$). From this state the mediator may enter in `HALTED` mode which will make the client recover their funds from round $r+1$ (state $S11$). Otherwise the client may confirm the withdrawal made during round $r+1$ (state $S10$).

We can observe that at the end of all the executions (states with red background) the amount extracted by the client is lower or equal to Z:

State	Amount extracted
S5	$X \leq Z$
S9	$Z \leq Z$
S10	$X + Y \leq Z$
S11	$X + (Z - X) = Z \leq Z$

□

B Proofs of liabilities tree

B.1 Security of proof of liabilities

In this section we expand on the security of the Merkle tree datastructure introduced in Section 2.1.2.7. More precisely we show the following: If an operator $\mathcal{O}$ provides a set of valid proofs τ_i for every honest participant P_i, then, assuming the hash function `H` used in the tree is collision-resistant, we can guarantee that the sum of balances for each honest participant is lower or equal to the global sum placed at the root.

Proposition 1. *Let $\kappa \in \mathbb{N}$ be the security parameter, $d \in \mathbb{N}$ some constant, and $||$ be the concatenation operator on binary strings. Let $\mathcal{H}$ be a collision-resistant hash function (CRHF) family and $\texttt{H} : \{0,1\}^{\kappa^d} \rightarrow \{0,1\}^{\kappa} \in \mathcal{H}$ be a randomly sampled hash function. Consider a balanced binary Merkle tree with 2^l leaves for some $l \in \mathbb{N}$ and root value $\texttt{r}$ such that:*

- *Every leaf value L is of the form $\texttt{val}_L = (h_L, \texttt{balance}(L))$ where $h_L \in \{0,1\}^{\kappa}$ is a hash value and $\texttt{balance}(L) \in \mathbb{N}$ is positive integer.*

[14]Note that property 5 would hold even if allowing one withdrawal per round.

- *For every internal node N with left node L and right node R,*

$$\texttt{val}_N = (\texttt{H}(\texttt{val}_L || \texttt{val}_R), \texttt{balance}(L) + \texttt{balance}(R))$$

Then, for any probabilistic-polynomial-time adversary $\mathcal{A}$:

$$\Pr\left[\texttt{r}, \tau_{i_1}, \tau_{i_2}, \cdots, \tau_{i_k} \leftarrow \mathcal{A}(\texttt{H}) : \begin{array}{c} \forall j \in [1, \cdots, k] : \texttt{validate}(\tau_{i_j}, \texttt{r}) = \mathit{true} \wedge \\ \sum_j \texttt{balance}(\tau_{i_j}) > \texttt{balance}(\texttt{r}) \end{array}\right] = \texttt{negl}(\kappa)$$

Proof. Given that the verification procedure is successful, we observe that for every node N, $\texttt{balance}(N) = \sum_{L \in \texttt{leaves}(N)} \texttt{balance}_L$ where $\texttt{leaves}(L)$ is the set of leaves of the tree rooted in N. Let $\mathcal{A}$ be the adversary that produces $\tau_{i_1}, \tau_{i_2}, \cdots, \tau_{i_k}$ such that

$$\forall j \in [1, \cdots, k] : \texttt{validate}(\tau_{i_j}, \texttt{r}) = \mathit{true}$$

where $\texttt{r}$ is the root of the tree T and $\sum_j \texttt{balance}(\tau_{i_j}) > \texttt{balance}(\texttt{r})$.

By construction, each proof τ_{i_j} contains a leaf value and sibling nodes that enable a verifier to reconstruct the root value $\texttt{r}$. First assume that there are 2 different $\tau_{i_{j_1}}$, $\tau_{i_{j_2}}$ such that the corresponding leaves are in the same position (see Figure 18) then this means that the adversary has found a collision for $\texttt{H}$. Thus from now on we assume that all proofs τ_{i_j} correspond to different leaves. Now assume that for two proofs $\tau_{i_{j_1}}$ $\tau_{i_{j_2}}$, the corresponding subtrees have some nodes in the same position which internal value is different. Then again, the adversary has found a collision for $\texttt{H}$ (see Figure 19). We thus can build a tree T' by merging all the subtrees corresponding to the proofs (see Figure 20). Let $\texttt{r}'$ be the root of the tree T', and L_τ be the set of indices of leaves that correspond to some proof τ_{i_j}. The we have that:

$$\begin{array}{rcl} \texttt{balance}(\texttt{r}') & = & \sum_{L \in \texttt{leaves}(\texttt{r}')} \texttt{balance}(L) \\ & = & \sum_{l \in L_\tau} \texttt{balance}(l) + \sum_{l \notin L_\tau} \texttt{balance}(l) \\ & = & \sum_j \texttt{balance}(\tau_{i_j}) + \sum_{l \notin L_\tau} \texttt{balance}(l) \end{array}$$

If $\texttt{balance}(\texttt{r}') < \sum_j \texttt{balance}(\tau_{i_j})$ then we must have that $\sum_{l \notin L_\tau} \texttt{balance}(l) < 0$ and hence there must be a node N^* (a leaf or internal) for one of the proofs τ_{i_j} such that $\texttt{balance}(N^*) < 0$. As the verification procedure does not allow negative amounts for any node we obtain the desired result.

□

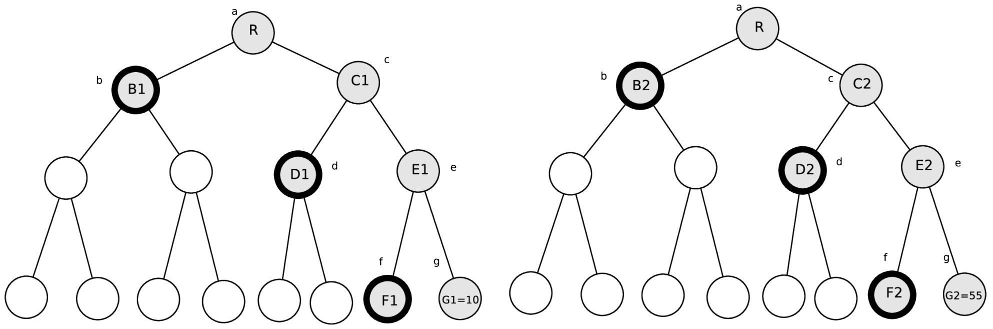

Figure 18: **It is impossible to build two different proofs $\tau_{i_{j_1}}$, $\tau_{i_{j_2}}$ so that the nodes are in the same position.**
Assume $\mathcal{A}$ can build $\tau_{i_{j_1}} \neq \tau_{i_{j_2}}$ such that both proofs (nodes with thick lines) have their nodes in the same position (i.e., nodes a, b, c, d, e, f, g) then a collision for H can be found as follows. If $(B1, C1) \neq (B2, C2)$ then $\mathtt{H}(B1||C1||v) = \mathtt{H}(B2||C2||v)$ and we have a collision (v is some value internal to the node, like the sum of balances.). If $(B1, C1) = (B2, C2)$ then we go deeper into the tree. If $(D1, E1) \neq (D2, E2)$, then $\mathtt{H}(D1||E1||v) = \mathtt{H}(D2||E2||v')$ is a collision. If $(D1, E1) = (D2, E2)$ we go deeper into the tree. Finally we know that $(F1, G1) \neq (F2, G2)$ and thus $\mathtt{H}(F1||G1||v) = \mathtt{H}(F2||G2||v')$ is a collision.

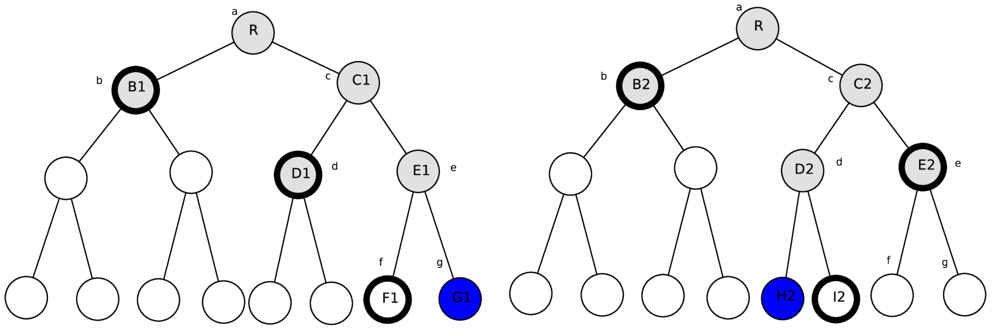

Figure 19: **It is impossible to build two proofs $\tau_{i_{j_1}}$, $\tau_{i_{j_2}}$ so that the corresponding subtrees have nodes of different value in the same position.**
In this picture, $\mathcal{A}$ can build $\tau_{i_{j_1}}$ (on the left) and $\tau_{i_{j_2}}$ (on the right) such that the corresponding subtrees shares some nodes in similar position (nodes with grey background). Then if a node in the left subtree is in the same position as a node in the right subtree, it must have the same internal value. Otherwise through a reasoning similar as in Figure 18, we can find a collision for H.

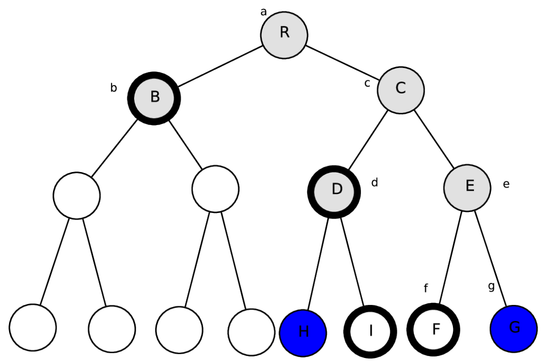

Figure 20: **Merging proofs and their corresponding subtrees.**
In this picture we see how to merge two subtrees corresponding to proofs $\tau_{i_{j_1}}$ and $\tau_{i_{j_2}}$ of Figure 19. As shown previously the nodes that are in the same position for both subtrees must have the same value and thus we have $B1 = B2 = B$, $C1 = C2 = C$, $D1 = D2 = D$ and $E1 = E2 = E$. Nodes with blue background correspond to the leaves of some participant.

B.2 Avoiding 2nd-preimage attacks using padding techniques

In the previous section we assumed that the tree is of height l, and is complete, which means it has 2^l leaves. The procedure that validates the presence of a leaf in the Merkle tree (using a list of siblings) must somehow take into consideration the structure of the tree. The reason is that if we only check that from the list of siblings and the leaf we can recover the root value we may still be victim of a second preimage attack. A basic example happens when the validation procedure does not differentiate between leaves and internal nodes [15] In this case the attacker only uses a subtree of the original tree. However more sophisticated attacks[16] exist where two trees T_1 and T_2 such that $T_1 \subsetneq T_2$ nor $T_2 \subsetneq T_1$ yield the same value at the root.

While these attacks are not always a problem in practice they are definitely an obstacle when we want to prove the security of our construction. Hence we need to provide a way to "lock" the structure of the tree we use to compute the commitment.

We propose the following padding. The new root value $\mathtt{r}'$ of the tree will be computed as follows: $\mathtt{r}' = \mathtt{H}(\mathtt{r}||H||W)$ where $\mathtt{r}$ is the original root, H is the height of the tree and W is the binary representation of the position of the right most leaf which can be obtained by concatenating the bits 0 (left) or 1 (right) from the root to the leaf. See an example in Figure 21. The validation procedure will have to check that

- The height of the proof is equal to H
- The binary representation of the leaf corresponding to the proof is no bigger than W.

[15]See `https://flawed.net.nz/2018/02/21/attacking-merkle-trees-with-a-second-preimage-attack/`.
[16]See [1], last paragraph of page 74 and page 75.

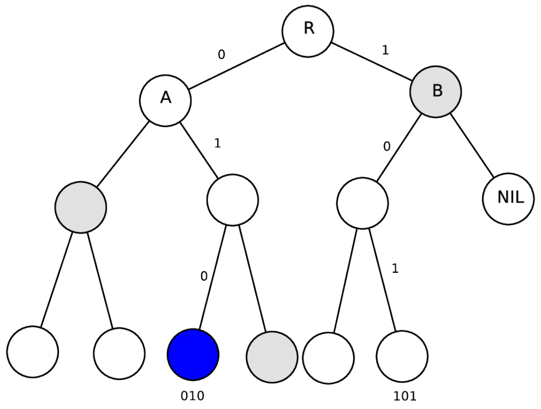

Figure 21: **Merkle tree with padding.**
In this figure the tree is of height 3 and thus can have at most $8 = 2^3$ leaves. The most right leaf is assigned the binary string 101. The blue leaf's position corresponds to the binary string $010 \leq 101$ which can be deduced from the proof τ (grey nodes): Start from the root: if τ is a left (resp. right) child then concatenate 1 (resp. 0). Go to the next level and apply the same procedure until to bottom node of the proof (sibling of the leaf). The root value of the tree is $R = \mathtt{H}(A||B||3||101)$.