Commuto

jimmyt

February 13, 2023

Abstract

This document introduces Commuto, a new type of exchange for traditional currencies and virtual currencies adopting the ERC20 token standard, built on the Ethereum Virtual Machine. The Commuto Protocol's name comes from the Latin word "commuto", meaning "I exchange" or "I barter."

Introduction

Commuto is a tool for private, permissionless, auditable, censorship resistant exchange of national currencies and tokens adopting the ERC20[**?**] standard on Ethereum Virtual Machine[**?**] compatible blockchains.

Commuto is primarily composed of two software components: a set of smart contracts deployed on an EVM-compatible blockchain, and a set of applications that can interact both with said on-chain contracts as well as other Commuto users. The smart contracts will be referred to henceforth as "Commuto Core" and said applications will be referred to as "Commuto Interfaces". An intention expressed by a Commuto user to buy or sell a particular ERC20 token in exchange for one out of a list of national currencies is known as an "offer". An exchange of a particular national currency and a specific amount of a particular ERC20 token between two users, a maker and a taker, is known as a "swap".

Because Commuto allows users to exchange national currency, certain pieces of private information (such as bank account details and addresses) must be privately exchanged between users. Additionally, users should be able to communicate with each other in a convenient manner, in order to resolve any problems that may arise during the swap process. While an EVM blockchain could theoretically be used to exchange such information by storing it (even if only temporarily) on-chain, the relatively high cost of storage on such blockchains makes this approach unfeasible. Additionally, many users are likely accustomed to instant messaging applications that deliver messages in less than a second. Additionally, the relatively longer time required to incorporate new transactions into such an EVM blockchain would act as a bottleneck to such a blockchain-based messaging system. This is yet another reason why EVM-blockchain-based communication between users is not practical. Therefore, Commuto Interfaces use Matrix[**?**] to reliably exchange information in a decentralized, censorship-resistant manner. Matrix is a network of nodes running software conforming to the Matrix Specification[**?**], which "defines a set of open APIs for decentralised communication, suitable for securely publishing, persisting and subscribing to data over a global open federation of servers with no single point of control." A Matrix Room is a directed acyclic graph of Events, the ordering of which is the chronological ordering of Events in the Room. Any node may maintain a copy of any such Room, and nodes may add new elements to Rooms. Events are simply JSON objects with zero or more "parent" events, which are chronological predecessors in the event graph.

Network nodes use state resolution algorithms and communicate with one another using federation algorithms (as defined in the specification) in order to maintain persistent, eventually-consistent synchronization of Room state across all nodes in the network. Therefore, no single node (referred to in the Matrix Specification and henceforth in this paper as a "homeserver") has control over any given Room. Thus, due to its open, flexible, decentralized, censorship-resistant nature, Commuto uses Matrix to exchange data between Commuto Interfaces.

Commuto Core

Commuto Core is composed of smart contracts that enable swaps between offer makers and offer takers, and also allow the resolution of disputes between makers and takers. (For example, a dispute may arise when a token buyer claims to have sent payment to the seller, but the seller claims that they have not received this payment.) We begin by considering the operations of the CommutoSwap smart contract, which enables the swapping process. Subsequently, we explain the functionality of the contracts allowing dispute resolution, governance, and the distribution of service fees collected by CommutoSwap. Smart contract code snippets are written in Solidity[**?**]. For readability, we include parameter names in function signatures, even though actual EVM smart contract function signatures do not include parameter names.

CommutoSwap

The CommutoSwap smart contract can be best understood by considering the way it is used step by step through the swap process, so we describe it in this context. Note that, as described in the introduction, the swap process includes the exchange of information via Matrix. However, because this section focuses on the operations of CommutoSwap, we temporarily omit the details of off-blockchain communication, and instead describe them in a later section. Throughout this document, we use the symbol ERC to refer to any ERC20 token, and a the symbol CUR to refer to any national currency. Sellers have ERC and want CUR, and buyers have CUR and want ERC.

Opening an Offer

An offer maker opens an offer by calling the `openOffer` function. This function has the following signature:

```
openOffer(bytes16 offerID, Offer newOffer)
```

where `offerID` is a Version-4 UUID[**?**], and `newOffer` is an `Offer` struct that is defined as follows:

Offer		
Property Type	Property Name	Description
bool	isCreated	Used by contract code to check for offer existence, will be set to true by `openOffer`.
bool	isTaken	Used by contract code to check whether an offer is taken, will be set to false by `openOffer`.
address	maker	The maker's address, will be sent to `msg.sender` by `openOffer`.

bytes	interfaceId	An array of bytes that should be placed in the "recipient" field of any message sent to the maker of this offer via Matrix.
address	stablecoin	The contract address of the token that the offer maker is offering to swap.
uint256	amountLowerBound	The lower bound on the range of token amounts that the maker is willing to swap.
uint256	amountUpperBound	The upper bound on the range of token amounts that the maker is willing to swap.
uint256	securityDepositAmount	The token amount to be used as a security deposit, which must not be less than ten percent of amountUpperBound.
uint256	serviceFeeRate	The percentage times 100 of the amount of swapped tokens that the maker and taker must pay as a service fee.
SwapDirection	direction	Indicates whether the maker is offering to buy tokens or sell tokens.
bytes[]	settlementMethods	An array of `bytes`, each one representing a method by which the maker is willing to send/receive payment for tokens.
uint256	protocolVersion	A number optionally describing which version of the Commuto Interface software was used to open this offer.

`SwapDirection` is an `enum` that is defined as follows:

```
enum SwapDirection {
    BUY
    SELL
}
```

A `SwapDirection.BUY` value indicates that the maker of the Offer is a buyer, as previously defined, and `SwapDirection.SELL` indicates that the maker of the Offer is a seller, as previously defined.

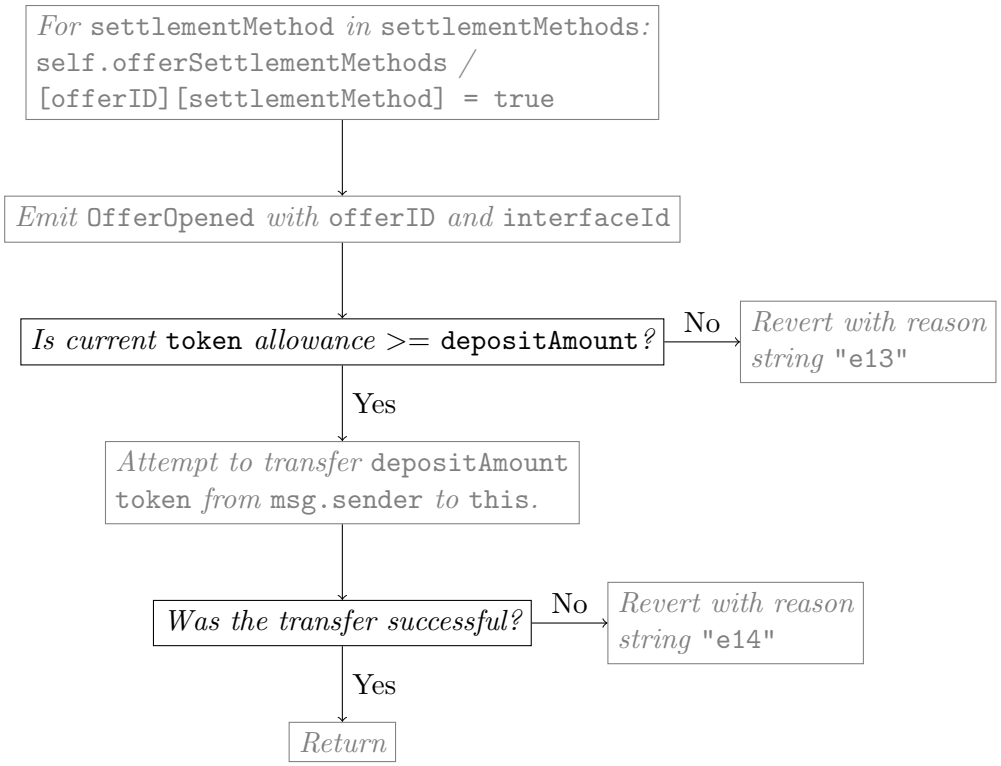
When called, `openOffer` ensures that an Offer with the specified ID does not already exist, validates all the data that it has been passed, maps the new Offer's ID to the Offer itself in `this.offers`, and maps the Offer's ID to its settlement methods, and then those settlement methods to `true` values in `this.offerSettlementMethods`. `this.offers` is a property of the CommutoSwap contract, and is a mapping from `bytes16` values to `Offer`s. `this.offerSettlementMethods` is a nested mapping. The outer mapping maps `bytes16` to inner mappings, and an inner mapping maps `bytes` to `bool` values. Then this emits an `OfferOpened` Event with the ID and interface ID of the offer. The `OfferOpened` Event has the following signature:

```
OfferOpened(bytes16 offerID, bytes interfaceId)
```

Then this transfers to itself from the maker an amount of the specified token equal to the maker's security deposit amount plus the percentage of the upper bound on the amount of tokens to exchange specified by `this.serviceFeeRate`. `this.serviceFeeRate` is a property of the CommutoSwap

contract of type `uint256`, expressed as a percentage times one hundred, of the amount of tokens exchanged between a buyer and seller, that both the buyer and seller must pay as a service fee. This process is shown in Figure 1.1.

Continued on next page.



Taking an Offer

An offer taker takes an existing offer by calling the `takeOffer` function. This function has the following signature:

```
takeOffer(bytes16 offerID, Swap newSwap)
```

where `offerID` is a Version-4 UUID and newSwap is a `Swap` struct that is defined as follows:

Swap		
Property Type	Property Name	Description
bool	isCreated	Used by contract code to check for swap existence, will be set to true by `takeOffer`.
bool	requiresFill	If the maker of this swap is the token seller, this indicates whether the seller has transferred to CommutoSwap the amount of tokens that they are selling. If the maker of this swap is not the token seller, this is not used.
address	maker	Identical to `Offer.maker` of the offer being taken.
bytes	makerInterfaceId	Identical to `Offer.interfaceId` of the offer being taken.
address	taker	The taker's address, will be sent to `msg.sender` by `takeOffer`.

bytes	takerInterfaceId	An array of bytes that should be placed in the "recipient" field of any message sent to the taker of this offer via Matrix.
address	stablecoin	Identical to `Offer.stablecoin` of the offer being taken.
uint256	amountLowerBound	Identical to `Offer.amountLowerBound` of the offer being taken.
uint256	amountUpperBound	Identical to `Offer.amountUpperBound` of the offer being taken.
uint256	securityDepositAmount	Identical to `Offer.securityDepositAmount` of the offer being taken.
uint256	takenSwapAmount	The exact amount of tokens that will be exchanged between the maker and the taker. Must be such that `amountLowerBound` $\leq$ takenSwapAmount $\leq$ `amountUpperBound`.
uint256	serviceFeeAmount	Used by contract code, will be set to `Offer.serviceFeeRate` of the offer being taken $\times$ `takenSwapAmount` $\div$ `10000`.
uint256	serviceFeeRate	Identical to `Offer.serviceFeeRate` of the offer being taken.
SwapDirection	direction	Identical to `Offer.direction` of the offer being taken.
bytes	settlementMethod	A `bytes` value representing the method by which the buyer will send payment for the exchanged tokens. Must be equal to one of the elements in the `Offer.settlementMethods` array of the offer being taken.
uint256	protocolVersion	Identical to `Offer.protocolVersion` of the offer being taken.
bool	isPaymentSent	Indicates whether the buyer has sent payment for tokens purchased. Will be set to false by `takeOffer`.
bool	isPaymentReceived	Indicates whether the seller has received payment for tokens sold. Will be set to false by `takeOffer`.
bool	hasBuyerClosed	Indicates whether the buyer has closed the swap to reclaim their security deposit (and unused service fee amount, if they are the maker). Will be set to false by `takeOffer`.
bool	hasSellerClosed	Indicates whether the seller has closed the swap to reclaim their security deposit (and unused service fee amount, if they are the maker). Will be set to false by `takeOffer`.

DisputeRaiser	disputeRaiser	Indicates whether the buyer, seller, or neither has raised a dispute during the swap process. Will be set to `NONE` by `takeOffer`.

`DisputeRaiser` is an `enum` that is defined as follows:

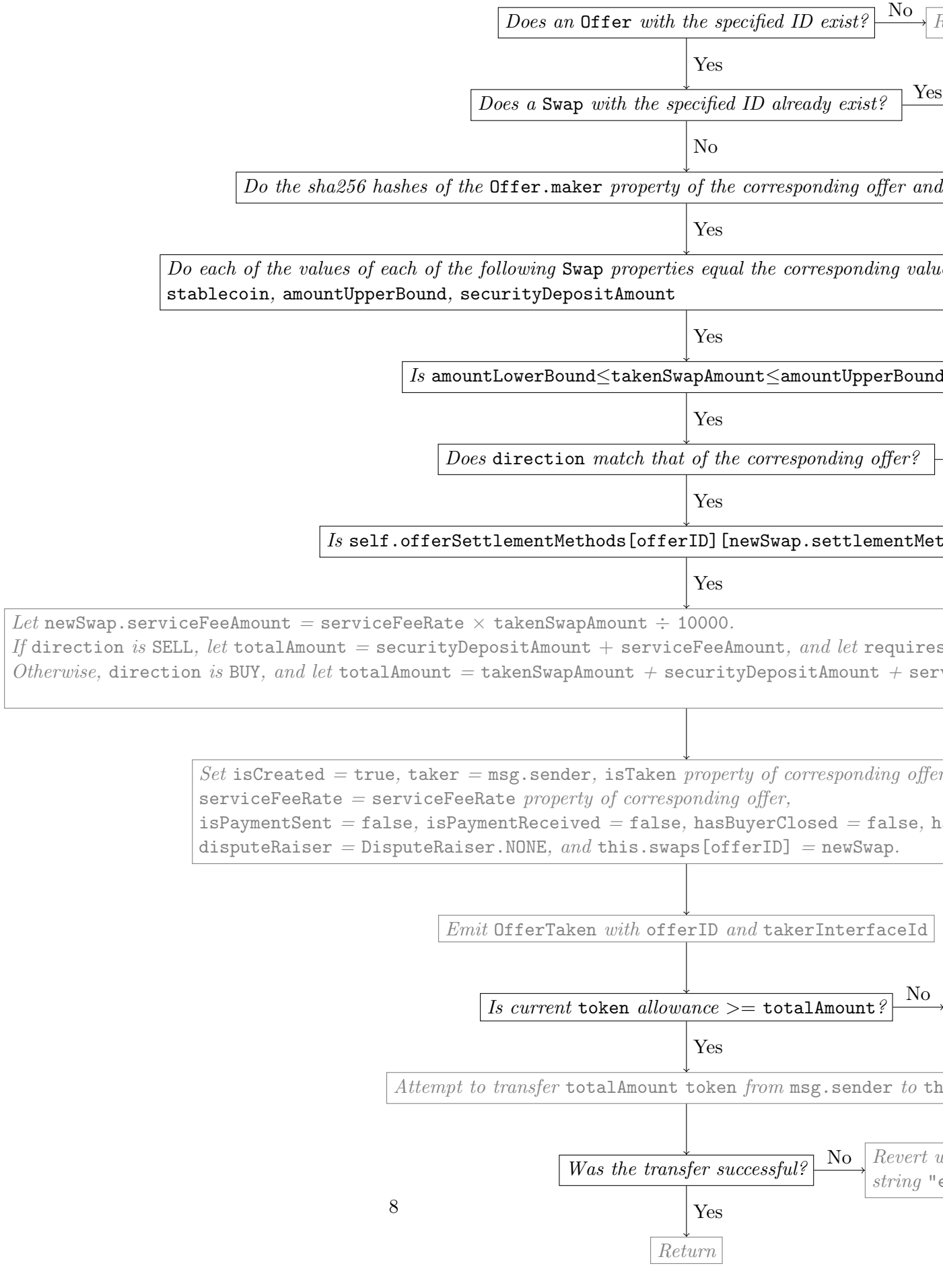
```
enum DisputeRaiser  {
    NONE,
    MAKER,
    TAKER
}
```

The meaning of these values and the use of the `Swap.disputeRaiser` property will be described later in this paper.

When called, `takeOffer` ensures that an Offer with the specified ID already exists and is not taken, and validates all the data that it has been passed to ensure that it corresponds with the Offer being taken. Then this sets a flag on the Offer being taken to indicate this, and adds the new swap to `this.swaps`. `this.swaps` is a property of the CommutoSwap contract, and is a mapping from `bytes16` values to `Swap`s. Then this emits an `OfferTaken` event with the ID of the Offer and the interface ID of the taker. The `OfferTaken` Event has the following signature:

```
OfferTaken(bytes16 offerID, bytes takerInterfaceId)
```

Then this transfers to itself from the taker an amount of the specified token. If the taker is the buyer, this amount is equal to the security deposit plus the exact service fee. If the taker is the seller, this amount is equal to the security deposit plus the exact service fee plus the amount to be sold. This process is shown in Figure 1.2.


Does an Offer with the specified ID exist?
No
Yes
Does a Swap with the specified ID already exist?
Yes
No
Do the sha256 hashes of the Offer.maker property of the corresponding offer and
Yes
Do each of the values of each of the following Swap properties equal the corresponding valu
stablecoin, amountUpperBound, securityDepositAmount
Yes
Is amountLowerBound≤takenSwapAmount≤amountUpperBound
Yes
Does direction match that of the corresponding offer?
Yes
Is self.offerSettlementMethods[offerID][newSwap.settlementMet
Yes
Let newSwap.serviceFeeAmount = serviceFeeRate × takenSwapAmount ÷ 10000.
If direction is SELL, let totalAmount = securityDepositAmount + serviceFeeAmount, and let requires
Otherwise, direction is BUY, and let totalAmount = takenSwapAmount + securityDepositAmount + serv
Set isCreated = true, taker = msg.sender, isTaken property of corresponding offer
serviceFeeRate = serviceFeeRate property of corresponding offer,
isPaymentSent = false, isPaymentReceived = false, hasBuyerClosed = false, h
disputeRaiser = DisputeRaiser.NONE, and this.swaps[offerID] = newSwap.
Emit OfferTaken with offerID and takerInterfaceId
Is current token allowance >= totalAmount?
No
Yes
Attempt to transfer totalAmount token from msg.sender to th
Was the transfer successful?
No
Revert u
string "e
Yes
Return

Filling a Swap

Consider the amount of tokens transferred from a user to CommutoSwap when the user is opening an offer to sell tokens. Note that this amount is the sum of the security deposit and the maximum service fee amount; The actual amount of tokens that the user wishes to sell is not transferred when opening such an offer. Therefore, when another user takes an offer to sell tokens, the amount of tokens being sold must be transferred from the offer maker to CommutoSwap. (Note that this is not necessary if the maker is offering to buy tokens, and thus this step is skilled in such cases.) The maker and seller accomplishes this by calling the `fillSwap` function. This function has the following signature:

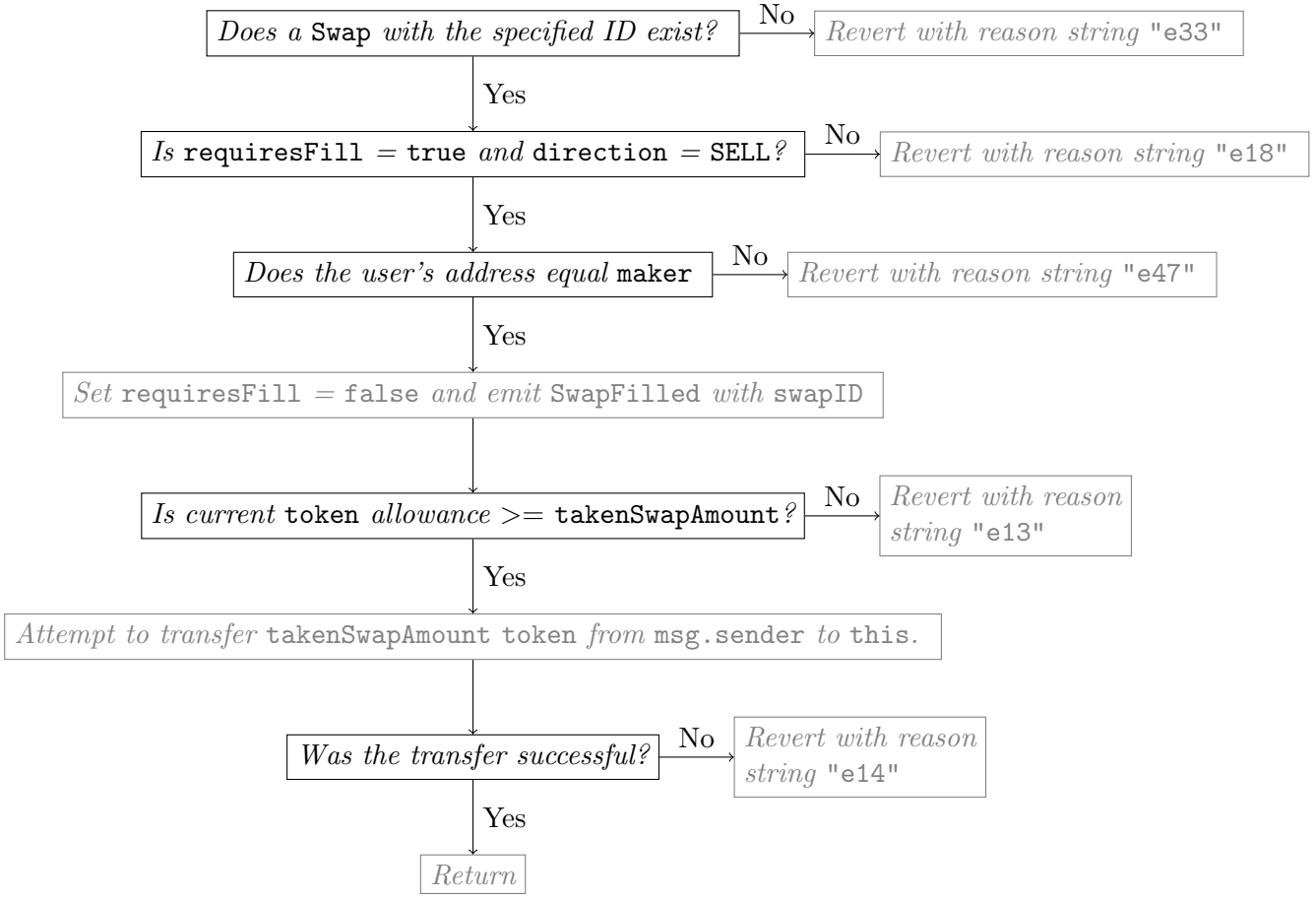
```
fillSwap(bytes16 swapID)
```

where `swapID` is the ID of the swap to be filled.

When called, `fillSwap` ensures that the caller is the maker and seller of the specified Swap, and that the Swap has not yet been filled. Then this sets a flag on the specified Swap indicating that it has been filled, and then emits a `SwapFilled` Event with the ID of the Swap. The `SwapFilled` Event has the following signature:

```
SwapFilled(bytes16 swapID)
```

Finally, this transfers to itself from the maker the exact amount of tokens that the taker is buying. This process is shown in Figure 1.3.



Sending Payment

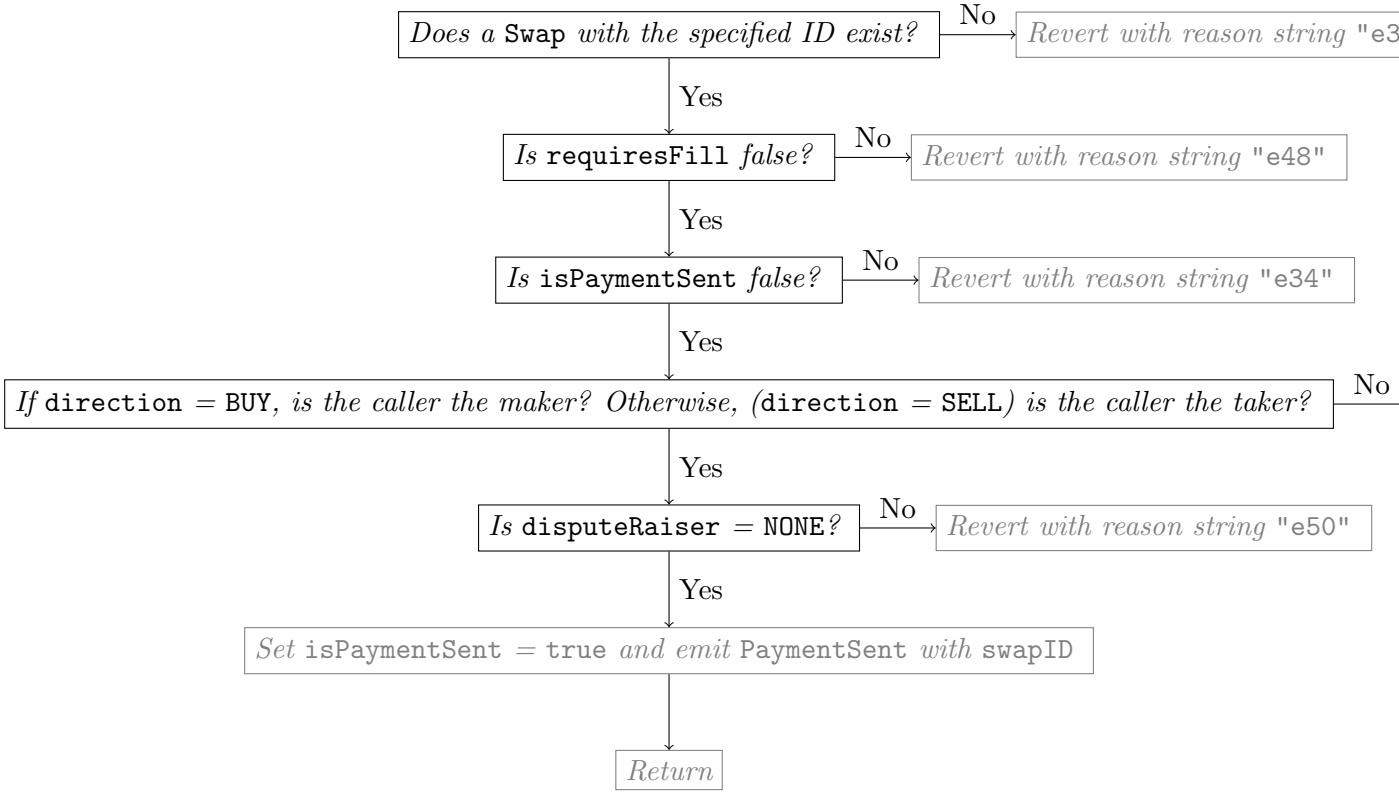
Now the token seller and buyer must exchange whatever information is necessary for the buyer to send payment to the seller. This will be accomplished via Matrix, and the precise details of this information exchanging process are described later in this document. Once said information is exchanged, the token buyer must send payment to the seller using the provided details. Then the buyer notifies the seller that this has been done by calling the `reportPaymentSent` function. This function has the following signature:

```
reportPaymentSent(bytes16 swapID)
```

When called, `reportPaymentSent` ensures that the the Swap does not need to be filled by calling `fillSwap`, that `reportPaymentSent` has not yet been called for the specified Swap, that the caller is the buyer, and that the specified Swap is not disputed (Disputed Swaps are described later in this document.) Then this sets a flag indicating that `reportPaymentSent` has been called and emits a `PaymentSent` event with the ID of the Swap. The `PaymentSent` Event has the following signature:

```
PaymentSent(bytes16 swapID)
```

This process is shown in Figure 1.4.



Receiving Payment

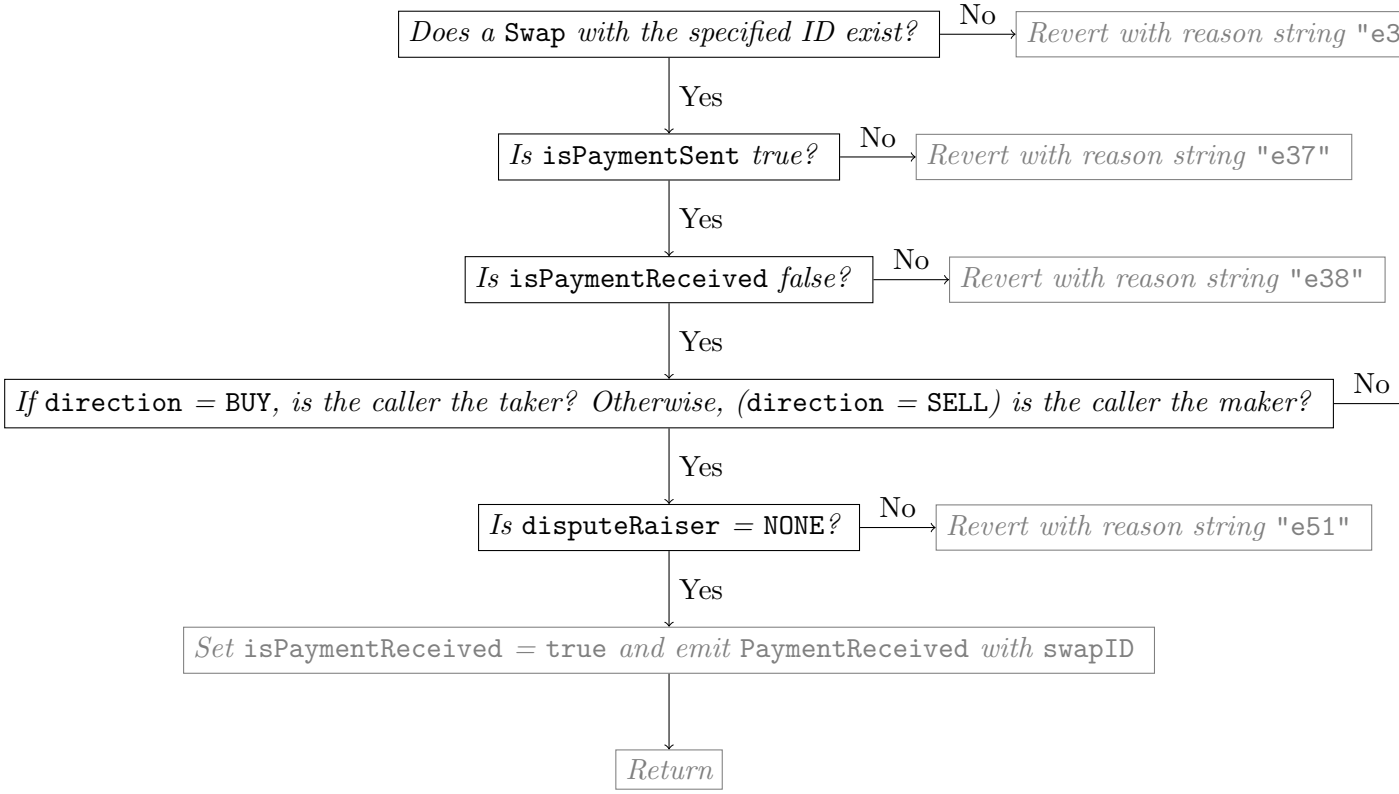
Once the token seller receives the payment that has been sent by the buyer, the seller must notify the buyer by calling the `reportPaymentReceived` function. This function has the following signature:

```
reportPaymentReceived(bytes16 swapID)
```

When called, `reportPaymentReceived` ensures that the buyer in the specified Swap has already called `reportPaymentSent`, that `reportPaymentReceived` has not yet been called for the specified Swap, that the caller is the seller, and that the specified Swap is not disputed. Then this sets a flag indicating that `reportPaymentReceived` has been called and emits a `PaymentReceived` Event with the ID of the specified Swap. The `PaymentReceived` Event has the following signature:

```
PaymentReceived(bytes16 swapID)
```

This process is shown in Figure 1.5.



`PaymentSent` is an Event with the following signature:

```
PaymentReceived(bytes16 swapID)
```

Closing the Swap

Now that the token seller has received payment from the buyer, both the buyer and the seller must close the swap by calling the `closeSwap` function. This function has the following signature:

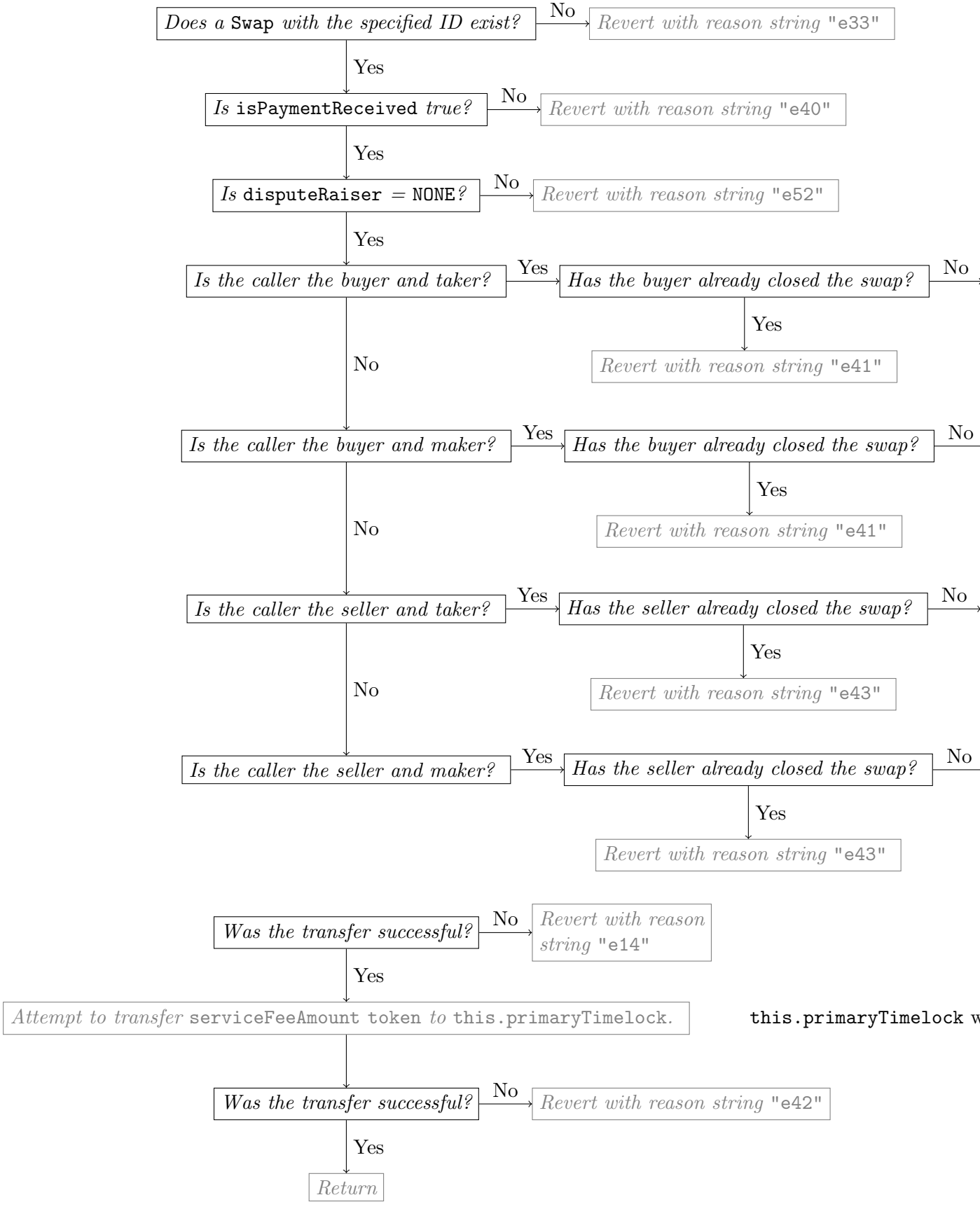
```
closeSwap(bytes16 swapID)
```

When called, `closeSwap` ensures that `reportPaymentReceived` has been called for the swap, and that the specified Swap is not disputed. It transfer's the caller's service fee to `this.primaryTimelock`,

which is a property of the CommutoSwap contract of type `address`, to which all service fees are sent. This property is also used by administration and governance-related code, which is described later in this document. If the caller is the buyer and taker, this will ensure that they have not already called `closeSwap` for the Swap, and then will refund their security deposit and will transfer the to them the tokens they bought. If the caller is the buyer and maker, this will ensure that they have not already called `closeSwap` for the Swap, and then will refund their security deposit and transfer to them the tokens they bought and the remainder of the maximum service fee (if any) that they transferred to CommutoSwap when opening the offer. If the caller is the seller and taker, this will ensure that they have not already called `closeSwap` for the swap, and then will refund their security deposit. If the caller is the seller and maker, this will ensure that they have not already called `closeSwap` for the swap, and then will refund their security deposit and will transfer to them the remainder of the maximum service fee (if any) that they transferred to CommutoSwap when opening the offer. If the caller is the buyer, this will then emit a BuyerClosed Event. If the caller is the seller, this will then emit a SellerClosed Event. Both specify the Swap ID. These Events have the following signature:

```
BuyerClosed(bytes16 swapID), SellerClosed(bytes16 swapID)
```

Once buyer and seller have both requested Swap closure, the Swap is completely closed. This process is shown in Figure 1.6.


Does a Swap with the specified ID exist?
No
Revert with reason string "e33"
Yes
Is isPaymentReceived true?
No
Revert with reason string "e40"
Yes
Is disputeRaiser = NONE?
No
Revert with reason string "e52"
Yes
Is the caller the buyer and taker?
Yes
Has the buyer already closed the swap?
No
Yes
Revert with reason string "e41"
No
Is the caller the buyer and maker?
Yes
Has the buyer already closed the swap?
No
Yes
Revert with reason string "e41"
No
Is the caller the seller and taker?
Yes
Has the seller already closed the swap?
No
Yes
Revert with reason string "e43"
No
Is the caller the seller and maker?
Yes
Has the seller already closed the swap?
No
Yes
Revert with reason string "e43"
Was the transfer successful?
No
Revert with reason string "e14"
Yes
Attempt to transfer serviceFeeAmount token to this.primaryTimelock.
this.primaryTimelock w
Was the transfer successful?
No
Revert with reason string "e42"
Yes
Return

Editing an Offer

A maker of an Offer may wish to change the settlement methods by which they are willing to send/receive payment for that particular Offer. If the Offer has not yet been taken, they may do this by calling the `editOffer` function. This function has the following signature:

```
editOffer(bytes16 offerID, Offer editedOffer)
```

where `offerID` is ID of the Offer to be edited and `editedOffer` is an `Offer` struct as previously defined, the `settlementMethods` property of which contains the maker's new settlement methods for the Offer, which will completely replace the Offer's current on-chain settlement methods.

When called, `editOffer` ensures that an Offer with the specified ID exists and is not taken, and that the caller is the maker of the specified Offer. Then, within the `bytes -> bool` mapping that maps supported settlement methods to `true` (which is the inner mapping to which the nested mapping `this.offerSettlementMethods` maps `offerID`) this maps all currently supported settlement methods to `false`. Then, within the same mapping, this maps all settlement methods specified in `editedOffer.settlementMethods` to `true`. Finally, this updates the value of the Offer's `settlementMethods` with the contents of `editedOffer.settlementMethods`, and emits an `OfferEdited` event, specifying the Offer ID. The `OfferEdited` Event has the following signature:

```
OfferEdited(bytes16 offerID)
```

Cancelling an Offer

A maker of an Offer may wish to cancel that particular Offer. If the Offer has not yet been taken, they may do this by calling the `cancelOffer` function. This function has the following signature:

```
cancelOffer(bytes16 offerID)
```

where `offerID` is the ID of the Offer to be canceled.

When called, `cancelOffer` ensures that an Offer with the specified ID exists and is not taken, and that the caller is the maker of the specified Offer. Then, this deletes the specified Offer from `this.offers`, and within the `bytes -> bool` mapping that maps supported settlement methods for this Offer to `true` (which is the inner mapping to which the nested mapping `this.offerSettlementMethods` maps `offerID`) this maps all the Offer's supported settlement methods to `false`. Finally, this emits an `OfferCanceled` event, specifying the Offer ID, and returns to the maker an amount of the specified token for the Offer being canceled equal to the sum of the security deposit and maximum service fee amount. The `OfferCanceled` Event has the following signature:

```
OfferCanceled(bytes16 offerID)
```

Dispute Process

In the process described above, both the buyer and seller have opportunities to misbehave and attempt to obtain payment from the other party without sending payment themselves. Additionally, other issues may arise during the process, such as the buyer becoming unable to send payment,

or the seller becoming unable to receive payment. In the event of such issues, users can raise a dispute to request intervention from dispute agents, in order to claim whatever tokens or payment are rightly theirs.

Dispute agents are unbiased, bonded 3rd parties who hear arguments from both the buyer and seller concerning what went wrong during the swap, and they should determine who should receive the tokens still held by CommutoSwap for the swap. CommutoSwap shall have a property named `activeDisputeAgents` of type `address[]`, which contains the address of every current/active dispute agent, and a property named `disputeAgents` that is a mapping from `address` to `bool` values. Every `address` in `activeDisputeAgents` should be mapped to `true` in `disputeAgents`, and all other `address`es should be mapped to `false`. CommutoSwap shall have a property named `disputes` that is a mapping from `bytes16` values to `Dispute` structs. This is used to map the IDs of disputed Swaps to the corresponding `Dispute` struct. The `Dispute` struct is defined as follows:

Dispute		
Property Type	Property Name	Description
uint	disputeRaisedBlockNum	The block number in which the dispute was raised.
address	disputeAgent0	The address of the first dispute agent selected by the dispute raiser.
address	disputeAgent1	The address of the second dispute agent selected by the dispute raiser.
address	disputeAgent2	The address of the third dispute agent selected by the dispute raiser.
bool	hasDA0Proposed	Indicates whether the first dispute agent has submitted a resolution proposal.
uint256	dA0MakerPayout	The amount of tokens that the first dispute agent believes the maker should receive.
uint256	dA0TakerPayout	The amount of tokens that the first dispute agent believes the taker should receive.
uint256	dA0ConfiscationPayout	The amount of tokens that the first dispute agent believes should be confiscated and sent to the primary timelock.
bool	hasDA1Proposed	Indicates whether the second dispute agent has submitted a resolution proposal.
uint256	dA1MakerPayout	The amount of tokens that the second dispute agent believes the maker should receive.
uint256	dA1TakerPayout	The amount of tokens that the second dispute agent believes the taker should receive.

uint256	dA1ConfiscationPayout	The amount of tokens that the second dispute agent believes should be confiscated and sent to the primary timelock.
bool	hasDA2Proposed	Indicates whether the third dispute agent has submitted a esolution proposal.
uint256	dA2MakerPayout	The amount of tokens that the third dispute agent believes the maker should receive.
uint256	dA2TakerPayout	The amount of tokens that the third dispute agent believes the taker should receive.
uint256	dA2ConfiscationPayout	The amount of tokens that the third dispute agent believes should be confiscated and sent to the primary timelock.
MatchingProposalPair	matchingProposals	Indicates whether two resolution proposals are identical to each other, and if so, which two they are.
DisputeReaction	makerReaction	The maker's reaction to the matching pair or triplet of reaction proposals, if any.
DisputeReaction	takerReaction	The taker's reaction to the matching pair or triplet of reaction proposals, if any.
DisputeState	state	Indicates the current state of the dispute.
bool	hasMakerPaidOut	Indicates whether the maker has closed the disputed swap to receive tokens in the amount specified by the accepted resolution proposals. Will definitely be `false` if identical resolutions have not yet been proposed.
bool	hasTakerPaidOut	Indicates whether the taker has closed the disputed swap to receive tokens in the amount specified by the accepted resolution proposals. Will definitely be `false` if identical resolutions have not yet been proposed.
uint256	totalWithoutSpentServiceFees	The total amount of tokens that the maker and taker have transferred to CommutoSwap for the Swap corresponding to this Dispute.

`MatchingProposalPair` is an `enum` that is defined as follows:

```
enum MatchingProposalPair {
    NO_MATCH,
    ZERO_AND_ONE,
    ZERO_AND_TWO,
    ONE_AND_TWO
}
```

`DisputeReaction` is an `enum` that is defined as follows:

```
enum DisputeReaction {
    NO_REACTION,
    ACCEPTED,
    REJECTED
}
```

`DisputeState` is an `enum` that is defined as follows:

```
enum DisputeState {
    OPEN,
    PAID_OUT,
    ESCALATED,
    ESCALATED_PAID_OUT
}
```

In the following description, a "swapper" is defined as either the maker of an Offer that has been taken, or the taker of an Offer.

Raising a Dispute

A swapper raises a dispute by calling the `raiseDispute` function. This function has the following signature:

```
raiseDispute(
    bytes16 swapID,
    address disputeAgent0,
    address disputeAgent1,
    address disputeAgent2
)
```

where `swapID` is the ID of the Swap to be disputed, and the remaining three arguments are unique `address`es that exist in CommutoSwap's `activeDisputeAgents` property.

When called, `raiseDispute` sets the `disputeRaiser` property of the Swap with the specified ID to indicate whether the caller is the maker or the taker of the swap. Then, in the `Dispute` to which `swapID` is mapped in CommutoSwap's `disputes` property, this sets the `disputeRaisedBlockNum` property to the current block number, and the dispute agent addresses to those specified by the caller. Finally, this emits a `DisputeRaised` event, specifying the Swap's ID and the three addresses supplied by the caller. The `DisputeRaised` event has the following signature:

```
DisputeRaised(
    bytes16 swapID,
    address disputeAgent0,
    address disputeAgent1,
    address disputeAgent2
)
```

Proposing a Resolution

Using a Commuto Interface, both the maker and the taker communicate with the dispute agents and explain the circumstances of the dispute (which may involve problems sending or receiving payment, an unresponsive peer, or other reason.) Similarly, the dispute agents communicate with each other to determine a resolution. The dispute agents have one week to determine such a resolution. They do this by indicating the amount of tokens currently being held by CommutoSwap for the disputed Swap in question that the buyer and seller should receive. Then each dispute agent calls the `proposeResolution` function, passing those amounts. This function has the following signature:

```
proposeResolution(
    bytes16 swapID,
    uint256 makerPayout,
    uint256 takerPayout,
    uint256 confiscationPayout
)
```

where `swapID` is the ID of the disputed Swap for which this resolution is being proposed, `makerPayout` is the amount of tokens that the caller believes the maker should receive, `takerPayout` is the amount of tokens that the caller believes the taker should receive, and `confiscationPayout` is the amount of tokens that the caller believes should be confiscated and transferred to `this.primaryTimelock`.

When called, `proposeResolution` ensures that a Swap with the specified ID exists, and that the corresponding Dispute has been escalated. (The meaning of an escalated Dispute will be explained later in this document.) Then, this calculates the total amount of tokens deposited by the maker and taker, minus their service fees, and ensures that this amount equals the sum of `makerPayout`, `takerPayout` and `confiscationPayout`. Additionally, it ensures that neither the maker nor the taker have reacted to any resolution proposed for this Swap by ensuring that the `makerReaction` and `takerReaction` properties of the corresponding `Dispute` both equal `DisputeReaction.NO_REACTION`. Finally, depending on whether the caller was the first, second or third dispute agent specified by the dispute raiser, this sets the proper "maker payout", "taker payout", "confiscation payout" and "has proposed" properties of the corresponding Dispute. If the caller is not one of the three dispute agents specified by the dispute raiser, the call fails at this point. Finally, this emits a `ResolutionProposed` event, specifying the Swap's ID and the address of the caller. The `ResolutionProposed` event has the following signature:

```
ResolutionProposed(
    bytes16 swapID,
    address disputeAgent
)
```

Reacting to a Resolution Proposal

Once at least two dispute agents have submitted identical resolution proposals, both the maker and the taker of the disputed Swap should indicate whether they accept or reject the proposed resolution by calling `reactToResolutionProposal`, specifying their reaction. This function has the following signature:

```
reactToResolutionProposal(
    bytes16 swapID,
    DisputeReaction reaction
)
```

where `swapID` is the ID of the disputed Swap for which the caller is reacting to a resolution proposal, and `reaction` is the caller's reaction. If the caller is the maker of the Swap, this ensures that they have not already reacted, and then sets the `makerReaction` property of the corresponding `Dispute` to `reaction`. If the caller is the taker of the Swap, this ensures that they have not already reacted, and then sets the `takerReaction` property of the corresponding `Dispute` to `reaction`. If the caller is neither the maker nor the taker, the call reverts at this point. If a pair of matching resolution proposals has not yet been found, this searches for them: If both the first and second dispute agents have submitted identical resolution proposals, this sets the `matchingResolutionProposals` property of the corresponding Dispute to `MatchingProposalPair.ZERO_AND_ONE`. Otherwise, if the second and third dispute agents have submitted identical resolution proposals, this sets the `matchingResolutionProposals` property of the corresponding Dispute to `MatchingProposalPair.ONE_AND_TWO`. Otherwise, if the first and third dispute agents have submitted identical resolution proposals, this sets the `matchingResolutionProposals` property of the corresponding Dispute to `MatchingProposalPair.ZERO_AND_TWO`. Otherwise, two matching resolution proposals have not been submitted, and this call reverts at this point. Finally, this emits an `ReactionSubmitted`, specifying the Swap ID, the address of the caller, and the caller's reaction. The `ReactionSubmitted` Event has the following signature:

```
ReactionSubmitted(bytes16 swapID, address addr, DisputeReaction reaction)
```