Development of smart contract for auctions in Beaker

Group: G1

Lorenzo Benetollo
Andrea Flamini
Marco Giangolini

18/10/2022

International School on Algorand Smart Contracts

Outline of presentation

1. Motivation
2. Deposited Bidding auction (Beaker)
    a. high level description;
3. Committed Bidding auction (Beaker)
    a. high level description;
4. Implementation Challenges
    a. Understanding the Algorand ecosystem;
    b. Design the protocol for the bridge implementation;

Motivation

Estimated Costs per Auction type (€)				
	Notarized	**Deposited**	**Committed**	**Confidential**
Bitcoin	lowSugFee: 25, 30 € highSugFee: 31, 13 €	Not Supported	Not Supported	Not Supported
Algorand	0, 0034 €	0, 2278 €	1, 7560 €	Not Supported
Ethereum	lowSugFee: 3, 88 € highSugFee: 5, 60 €	lowSugFee: 13, 63 € highSugFee: 19, 67 €	lowSugFee: 35, 04 € highSugFee: 50, 55 €	Committed + k_6

Mogavero, Francesco, et al. "The Blockchain Quadrilemma: When Also Computational Effectiveness Matters." 2021 IEEE Symposium on Computers and Communications (ISCC). IEEE, 2021.

Different platform have different:

1. block finality;
2. scripting language for smart contracts;
3. costs in term of fees;
4. ….

Deposited Bidding auction (Beaker)

Actors and Workflow:

1. contract creator
    a. creates the auction (1)
    b. binds an asset to the created auction and starts the auction (2)
    c. closes the auction (anyone can close it) (5)
2. bidders
    a. send their public bids to the contract (3)
    b. get refund when necessary (4)

The asset is transferred to the auction winner or it is returned to the contract creator if there have been no bids.

Deposited Bidding auction (Beaker)

```python
@external(authorize = Authorize.only(owner))
def setup(self, payment: abi.PaymentTransaction, starting_price: abi.Uint64, nft: abi.Asset,
          start_offset: abi.Uint64, duration: abi.Uint64):
    payment = payment.get()
    return Seq(
        # Set global state
        self.highest_bid.set(starting_price.get()),
        self.nft_id.set(nft.asset_id()),
        self.auction_start.set(Global.latest_timestamp() + start_offset.get()),
        self.auction_end.set(Global.latest_timestamp() + start_offset.get() + duration.get()),
        Assert(
            And(
                Global.latest_timestamp() < self.auction_start.get(),
                self.auction_start.get() < self.auction_end.get(),
                payment.type_enum() == TxnType.Payment,
                payment.sender() == Txn.sender(),
                payment.receiver() == Global.current_application_address(),
            )
        ),
        self.do_opt_in(self.nft_id)
    )
```

```python
@external
def bid(self, payment: abi.PaymentTransaction, previous_bidder: abi.Account):
    payment = payment.get()
    auction_end = self.auction_end.get()
    highest_bidder = self.highest_bidder.get()
    highest_bid = self.highest_bid.get()

    return Seq(
        Assert(
            And(
                Global.latest_timestamp() < auction_end,
                payment.amount() > highest_bid,
                Txn.sender() == payment.sender()
            )
        ),
        # Return money to previous bidder
        If(highest_bidder != Bytes(""),
            Seq(Assert(highest_bidder == previous_bidder.address()), self.pay_bidder(highest_bidder, highest_bid))),
        # Set global state
        self.highest_bid.set(payment.amount()),
        self.highest_bidder.set(payment.sender())
    )
```

```python
# Asset opt-in for the smart contract
@internal(TealType.none)
def do_opt_in(self, asset_id):
    return self.do_axfer(self.address, asset_id, Int(0))


# Asset transfer to the smart contract
@internal(TealType.none)
def do_axfer(self, receiver, asset_id, amount):
    return InnerTxnBuilder.Execute(
        {
            TxnField.type_enum: TxnType.AssetTransfer,
            TxnField.xfer_asset: asset_id,
            TxnField.asset_amount: amount,
            TxnField.asset_receiver: receiver,
            TxnField.fee: MIN_FEE
        }
    )
```

```python
# Asset close out from the smart contract to the receiver
@internal(TealType.none)
def do_aclose(self, receiver, asset_id, amount):
    return InnerTxnBuilder.Execute(
        {
            TxnField.type_enum: TxnType.AssetTransfer,
            TxnField.xfer_asset: asset_id,
            TxnField.asset_amount: amount,
            TxnField.asset_receiver: receiver,
            TxnField.fee: MIN_FEE,
            TxnField.asset_close_to: receiver
        }
    )
```

Committed Bidding auction (Beaker)

The smart contract is programmed to manage two distinct phases:

1. Committing Phase, in which bidders send their commitment
2. Bidding Phase, in which bidders show their bid

The two phases are managed by means of timestamps.

Committed Bidding auction (Beaker)

Actors and Workflow:

1. contract creator
   a. creates the auction (1)
   b. binds an asset to the created auction and starts the auction (2)
   c. closes the auction (anyone can close it) (6)

2. bidders
   a. send their committed bids to the contract using SHA-256 and pay a deposit (3)
   b. sends a transaction to the contract which opens the commitment (4)
   c. get refunded when necessary (5)

The asset is transferred to the auction winner or it is returned to the contract creator if there have been no bids.

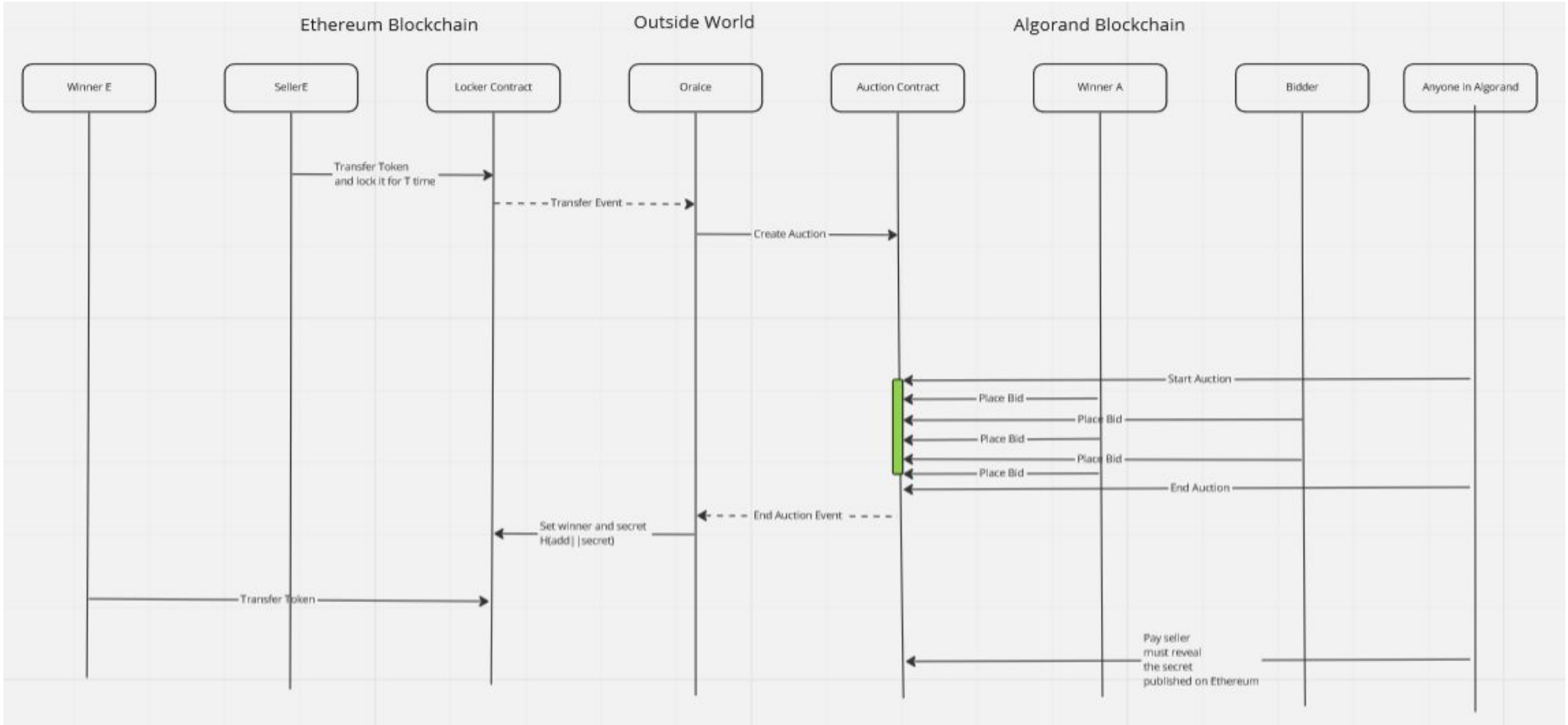
Committed Bidding auction (Beaker)

```python
# Local Bytes (1)
# to enable the contract writing the commitment (bid hashed value) into local state -> TealType.bytes (hashed amoun
# Enabling max_keys = 1, only one commitment is allowed
commitment: Final[DynamicAccountStateValue] = DynamicAccountStateValue(
    stack_type = TealType.bytes,
    max_keys = 1
)

# Local Ints (1)
# to enable the contract writing the bid (revealed) into local state -> TealType.uint64 (amount)
# max_keys(open_commitment) == max_keys(commitment) to map commit and corrispondent bid
open_commitment: Final[DynamicAccountStateValue] = DynamicAccountStateValue(
    stack_type = TealType.uint64,
    max_keys = 1
)

@external
def bid(self, payment: abi.PaymentTransaction, highest_bidder: abi.Account, k: abi.Uint8):
    payment = payment.get()
    return Seq(
        Assert(And(
            # Checking (current time) >= (commit end) is fundamental. Every user can do just one commit (max_keys = 1)
            # in the commitment phase. Every other commitment (if submitted) from the same user is going to be overwritt
            # during the commit phase (hence, before bids are revealed).
            Global.latest_timestamp() < self.auction_end.get(),
            Global.latest_timestamp() >= self.commit_end.get()
        ), comment="timestamp"),
        Assert(And(
            payment.type_enum() == TxnType.Payment,
            payment.sender() == Txn.sender(),
            payment.receiver() == Global.current_application_address(),
            Sha256(Itob(payment.amount())) == self.commitment[k][payment.sender()]
        ), comment="payment"),

        Log(Sha256(Itob(payment.amount()))),
        self.commitment[k][payment.sender()].delete(),
        self.open_commitment[k][payment.sender()].set(payment.amount()),

        If(payment.amount() > self.highest_bid.get())                                   # current bid > previous bid
        .Then(
            Seq(
                self.pay_bidder(Txn.sender(), self.deposit.get()),                      # give back the deposit (commit and the bid are valid)
                If(self.highest_bidder != Bytes(""))                                    # there was a previous highest bidder
                .Then(
                    self.pay_bidder(self.highest_bidder.get(), self.highest_bid.get())  # give back to the previous bidder bid + deposit
                ),
                # Set global state
                self.highest_bidder.set(Txn.sender()),
                self.highest_bid.set(payment.amount()),
                Approve()
            )
        ).Else(                                                                         # current bid <= previous bid
            self.pay_bidder(Txn.sender(), self.deposit.get() + payment.amount())        # give back bid + deposit to the current bidder (temporal order rule)
        )
    )
```

```python
@opt_in
def opt_in(self):                                   # opt-in for commitment
    return self.initialize_account_state()

@external
def nft_opt_in(self, nft: abi.Asset):               # opt-in the nft into the smart contract
    return self.do_opt_in(nft.asset_id())

    @external
     def end_auction(self, highest_bidder: abi.Account, nft: abi.Asset):
    def end_auction(self, nft: abi.Asset):
        auction_end = self.auction_end.get()
        highest_bid = self.highest_bid.get()
        owner = self.owner.get()
        highest_bidder = self.highest_bidder.get()

        return Seq(
            Assert(Global.latest_timestamp() > auction_end),
            If(self.highest_bidder == Global.zero_address())
            .Then(
                Seq(
                    self.do_aclose(owner, self.nft_id, Int(1)),
                    self.pay_owner(owner, highest_bid)
                )
            )
            .Else(
                Seq(
                    self.do_aclose(highest_bidder, self.nft_id, Int(1)),
                    self.pay_owner(owner, highest_bid)
                )
            )
        )
```

Implementation challenges

1. Understanding the Algorand ecosystem
    a. transaction (payment, asset creation, app creation)
    b. transaction fees (fee, min_fee, flat_fee)
    c. SDK (Python and JavaScript)
    d. Beaker framework (lack of documentation)
    e. Algorand Sandbox
2. Design the protocol for the bridge implementation
    a. Secure protocol design
    b. Oracle implementation: reading the Algorand blockchain state



We remove responsibility from the oracle performing a safe atomic swap. **The Oracle only moves $H(account_E||secret)$ from Algorand to Ethereum**

1. We need the secret to allow the seller redeem the amount of money bidded by the winner.
2. We need $account_E$ to be sure that only the right user can get possession of the asset. Only the secret is not enough because before being included in the blockchain a transaction is published in the network!

Future works

1. fix some vulnerabilities of the implemented smart contracts
2. implementation of the Oracle