-

Problem Analysis

Here's a summary of what we hope to achieve at the end of this workshop.

The purpose of this workshop is to create an NFT auction and get bidders to race to get the highest bid with a limited amount of time.

Let's breakdown some concepts that we need to understand before we ask critical questions.

-

Blockchain

A blockchain is a distributed ledger that records transactions in a series of blocks. The series of blocks are linked together using Merkel Tree where each block has a link to the previous block. What makes blockchain unique is that they use consensus which means each participant peer can confirm the validity of the blockchain.

-

BlockHeight

BlockHeight keeps count of all the number of blocks that have been mined since the beginning of the blockchain. Since each block can be mined at an approximate amount of time, the block height can be used to measure the auction time.

-

Auction

An auction is a public sale where the item being auctioned by the `auctioneer` goes to the highest `bidder` after a certain amount of time limit.

-

Decentralized App

Decentralized applications are immutable, operate autonomously, and are not tied to a single entity. They run on the blockchain using consensus algorithms to force all participants to agree.

1.

Objectives.

Let's run through the objectives :

1. We need to create a Decentralized application and publish it to a blockchain.
2. The decentralized application should be able to conduct a successful auction in consensus.

3. The decentralized application should be able to autonomously moderate the auction and ensure fairness and honesty amongst all the participants.

-

Expected Output

We expect the decentralized application to :

- Announce a winner at the end of the auction,
- Transfer the NFT to the winner.
- Transfer the highest bid to the auctioneer.

-

Expected Input.

We expect the decentralized application to handle :

- A NFT to be sent to the decentralized app.
- A NFT price
- An auction duration.
- A NFT bid.

-

Expected Processes for a Successful Auction.

In order to achieve a successful auction, the decentralized app needs a few processes to be done :

1. We need an auctioneer to create a new contract/DAPP.
2. We need an auctioneer to make the NFT being auctioned known to the contract.
3. Once the auctioneer sends the NFT to the contract, the auction is ready to start.
4. A bidder must `OPT-IN` the contract/DAPP and accept the NFT token.
5. For a bidder to make a successful bid, the bid must be placed on time and the amount should be larger than the last bid.
6. The auction should continue until the timeout hits.
7. If timeout hits, the NFT should be sent to the highest bidder and the highest bid should be sent to auctioneer.
8. The decentralized app should exit and self-destruct if it has no NFT and the contract balance is 0.

-

Analyzing the Scope of the Problem.

Based on what we have touched so far, we can now assess the possibilities and the limitations that will be encountered in the process of creating a successful auction. In order to get a clear understanding of the problem, let's break down our program into specifics by asking key questions :

1. What modules will we use to create and compile the program ?
2. Who will be involved in executing the contract ?
3. What tasks should be completed to make a successful auction ?
4. What data types will be used to hold program information ?
5. What functions can we use to help the actors participate ?
6. What algorithms can we use to run an honest auction ?

•

Problem Design

Here we will answer the questions we asked during the problem analysis.

The purpose of this workshop is to create an NFT auction and get bidders to race to get the highest bid with a limited amount of time.

Let's breakdown some questions we need to answer before we can design the DAPP.

•

▶

Which programming language will we use to create our DAPP ?

Reach is a domain specific language for building decentralized applications. The Reach Module is a `.rsh` file that contains the DAPP that can run on multiple blockchain platforms.

1. ▶

Reach Module

The Reach Module must begin with a `version type` as it's first line and stored in a `index.rsh` file.

> index.rsh

```
'reach 0.1';
```

> Reach Syntax is written in **JavaScript** syntax.

•

▶

Reach App.

The Reach App specifies the DAPP in it's entirety. It is the body of the DAPP.

Reach uses Module-level Identifiers such as export to identify the module to be compiled.

> index.rsh

```
export const main = Reach.App(() => {
    //DAPP body.
})
```

> All the functions we want to perform will go into the `main` function.

-

▶

Reach Participant.

A Participant is a logical actor who takes part in a DAPP and is associated with an address on the consensus network. A Reach participant is capable of persistently storing data on the local state.

> index.rsh

```
export const main = Reach.App(() => {
    //DAPP body.
    const Auctioneer = Participant('Auctioneer', {
        //Auctioneer body
    });
})
```

> All the functions that will be necessary for the `auctioneer` to perform will be put inside the `Auctioneer body`.

-

▶

Reach API.

A Reach API is group of Reach Participant who are racing to achieve the same goal in a DAPP.

> index.rsh

```
export const main = Reach.App(() => {
    //DAPP body.
    const Bidder = API('Bidder', {
        //Bidder interface.
    });
})
```

> A main difference between a `Reach Participant` and a `Reach API` is that a `Reach API` can be called from the actors` frontend.

> All the functions that will be necessary for the `bidder` to perform will be put inside the `Bidder Interface`.

-

▶

Which data types will we use in our DAPP to hold information ?

We can use reach Types as guidance to choose which types we can use to represent our data.

If we go back to our problem analysis, we can take a look at our expected input and our expected output and try to convert all that information to Reach Types.

-

▶

Handling Output Data

Let's take a look at the `Reach Types` we will use to represent our output data.

Announcing a winner at the end of the auction.

- In order to announce a winner, we will need the participant to learn new information :

    1. The winning bid.

    2. The Winner.

- How can we represent these two pieces of information in a DAPP ?

    1. We can represent the winning bid with a UInt type.

    2. We can represent the winner with an Address type.

-

▶

Handling Input Data

Let's take a look at the `Reach Types` we will use to represent our input data.

Adding the NFT for the auction.

- In order to add a NFT to the contract, we will need :

1. The NFT ID.
2. The NFT price / starting bid.
3. The auction duration.

- How can we represent this information in a DAPP ?

   1. We can use a Token type to represent the NFT ID.
   2. Since the price is a number, we can use a UInt type to represent the price.
   3. We can use a UInt type to represent the auction duration, which will represent block height as opposed to actual time.

-

▶

Which functions will we use to manipulate the data ?

We can use Reach Functions type as guidance to choose which types we can use to represent our data.

Reach Functions type will be useful to arranging the input and output data more efficiently.

-

▶

Output Functions.

Output functions that will be used to notify our frontend.

-

Announcing a winner at the end of the auction.

- In order to announce a winner, we will need the participant to learn new information :

   1. The winning bid.
   2. The Winner.

- We have already established how we can represent the data, now let's see how we can send this information to the frontend.

```
//showOutcone function.
showOutcome: Fun([Address, UInt], Null),
```

`showOutcome` is a function that sends `[Address, UInt]` which are the `[winner, winning bid]` to the frontend and does not expect a return value.

-

Transferring the NFT to the winner.

- After the auction is complete, we will need to transfer the NFT from the contract to the winner.
- Reach offers a Transfer function, a consensus step that calls the contract to pay a token to the given address.

```
transfer(`UInt`,`Token`).to(`Address`);
```

`transfer` takes a `amount`(`UInt`), a `Token` and transfers the amount to an `Address` once a condition is met.

-

Transferring the highest bid to the auctioneer.

- We will also need to transfer the highest bid to the auctioneer once the auction ends.
- Reach offers a Transfer function, a consensus step that calls the contract to pay a token to the given address.

```
transfer(`UInt`).to(`Address`);
```

`transfer` takes a `amount`(`UInt`) and transfers the amount to an `Address` once a condition is met.

-

▶

Input Functions.

Input functions will be used to notify our frontend what the backend expects as well as calling backend functions from the frontend.

-

Receiving the NFT to be auctioned from the frontend.

Since it is the auctioneers' responsibility to add the NFT to the contract, we will make sure that only the Auctioneer who is capable of setting the NFT.

Any time a participant sends information to the backend contract, we can use an `interact` interface to get that information.

Here's the information that we will need from the auctioneer :

1. The NFT ID.

2. The NFT price / starting bid.
3. The auction duration.

- We have already established how we can represent the data, now let's see how we can get this information to the frontend.

```
//getSale function
    getSale: Fun([],[Token, UInt, UInt]),
```

`getSale` function expects the [Token, UInt, UInt]/([nftId, price, auctionTime]) from the frontend.

Reach also provides an Object type that can be used to nest other types.

```
Object({
    nftId: Token,
    minBid: UInt,
    lenInBlocks: UInt,
})
```

Let's add this to the function :

```
getSale: Fun([], Object({
    nftId: Token,
    minBid: UInt,
    lenInBlocks: UInt,
}))
```

-

Allowing a bidder to place a bid.

- Bidders will also be expected to place bid i.e. Call a bid function from the frontend.

```
bid: Fun([UInt], Null),
```

`bid` expects a number from the frontend which a Bidder address will be attached to during the auction.

-

Alerting when the auction is ready.

- We can also alert the Auctioneer when the auction is ready to start.

```
auctionReady: Fun([], Null),
```

`auctionReady` notifies the Auctioneer frontend when the auction is ready.

-

▶

Which tasks should we run to make a successful auction ?

Introduction to Reach Steps

Here, we will be introducing new concepts that will be useful to understand how Reach works.

Reach can be in two states :

- Local step
- Consensus step

Most DAPP have a creator, an actor, a wager and a condition. Before a contract becomes autonomous, the creator publishes the criteria for the wager and the condition. Once the creator does this, they have no control of the outcome, nor can they pause the contract once it initiates. The bidder can observe the contract on the blockchain and decide whether to opt-in.

Local steps are steps that are run locally by a single actor while consensus steps are run on the blockchain in consensus. To improve anonymity and security, local steps exists to ensure that each actor is unaware of what any other actor is up to.

If they choose to make the information known, they must enter a consensus step and make the information know on the blockchain. Consensus steps also ensure that the core logic and condition of the contract, run on the blockchain, where all active actors can see what is going on.

Let's take a look at the tasks that we will need to execute in order to make a successful auction :

-

▶

Adding Actors

We have already selected how we will represent our data, we have also established functions that can be used to get the necessary data, now the final step is to add the functions into classes that can perform logic and store states. In Reach, they are called Participants.

-

Adding an Auctioneer Pariticipant

- In the functions section, we saw how we could use input and output functions to collect data, now let's add the necessary data to our auctioneer participant.

```
const Auctioneer = Participant('Auctioneer', {
    //getSale function.
    getSale: Fun([], Object({
        nftId: Token,
        minBid: UInt,
        lenInBlocks: UInt,
    })),
    //auctionReady function.
    auctionReady: Fun([], Null),

    //seeBid function.
    seeBid: Fun([Address, UInt], Null),

    //showOutcome function.
    showOutcome: Fun([Address, UInt], Null),
});
```

- Here, we create an Auctioneer participant with the name `Auctioneer` and the data we need to run the auction.
- We used the `getSale` function to get the NFT data from the frontend.
- We used the `auctionReady` function to notify the Auctioneer when the auction is ready.
- We used the `seeBid` function to notify the Auctioneer when a bidder has placed a bid.
- We used the `showOutcome` function to notify the Auctioneer when the auction is over and who the winner is.

-

Adding a Bidder Participant.

- For the bidder, we will also use a participant class. However, unlike the Auctioneer who is the deployer of the contract, we expect multiple bidders to be added to the contract.
- Reach provides a way of representing multiple participants in the Reach API class.
- You can think of the API as a representation of multiple participants expected to race to a certain goal.
- For our instance, we expect the Bidders to be able to place a bid.

```
const Bidder = API('Bidder', {
    //Bidder interface.
    bid: Fun([UInt], Tuple(UInt,Address, UInt)),
});
```

- We have a Bidder interface that can be used to represent multiple bidders.

- Each Bidder will have a bid function that they can use to place a bid.

> One of the advantages of Reach API is that functions can be called from the frontend.

-

▶

Initializing the contract.

What comes after creating the actors/participants.

There are other Reach interfaces we have not touched on such as Views and Events. We have only talked about the API and the Participant.

These interfaces represent what functions and classes should be replicated by the frontend and should come before the `init()` statement.

```
const newParticipant = Participant(participantName, participantInteractInterface)

const newAPI = API(APIName, APIInteractInterface)

const newView = View(ViewName, ViewInteractInterface)

const newEvent = Events(EventName, EventInteractInterface)

init()

// Consensus step or local step.
```

`init()` symbolizes the beginning of the DApp to be compiled. In other words, anything that follows the `init` statement is either a local step or a consensus step.

-

▶

Using Local Steps.

What comes after the `init()` statement.

-

Local Private Step.

Once the `init()` statement has been called, the DApp is automatically in a [local private](https://docs.reach.sh/model/#p_33) step. This means, that any information being accessed is only available on the local machine of the participant.

-

Local Public Step.

Local private is not that useful if we have information : such as the NFT data, that we need other actors to access. So how do we move from local private to local public ?

- We use Reach declassify to do this.

> Reach declassify can pass information from the frontend to the backend. Let's test this with the `Auctioneer` participants' `getSale` function to get the NFT information from the frontend.

```
//declassify function.
Auctioneer.only(() => {
    const {nftId, minBid, lenInBlocks} = declassify(interact.getSale());
});
```

- `Auctioneer.[only]()` function makes sure that only the `Auctioneer` i.e the creator of the contract, can access this function.
- `interact` is a function used to get information from the frontend.
- `declassify` makes the information public.

> Here, we are interacting with the `Auctioneer` frontend and `await`ing the result (const {nftId, minBid, lenInBlocks}).

The information is public but it's still local. Let's move to a consensus step to publicize the information on the contract.

-

▶

Using Consensus Steps.

Publishing information onto the contract.

In order to move from a local step to a consensus step, Reach provides a few mechanisms that can help us do exactly that.

1.

Publish

Consensus step shares information on the blockchain, we can use Reach Publish to share NFT information with the contract.

```
Auctioneer.publish(nftId, minBid, lenInBlocks);
```

- `Auctioneer.publish()` function makes sure that only the `Auctioneer` i.e the creator of the contract, can publish this information onto the contract.

> Using Commit()

Once we are in a consensus step, we can fall back to a local step by using Commit().

Why would we do this ?

- Security reasons

  We use commit to ensure that we are back to a `local private` state before we can conduct sensitive functions such as making payments to the contract.

```
commit();
```

-

Pay

Now that we are back to a `local private` step, we can transfer the NFT from the Auctioneer to the contract.

```
Auctioneer.pay([[1, nftId]])
```

- `Auctioneer.pay()` function makes sure that only the `Auctioneer` i.e the creator of the contract, can pay.
- We are sending one NFT Token to the contract to be auctioned.

> We send [1] NFT because an NFT should be unique. Instead of directly sending the `UInt 1`, we can put it in a variable so that we can store the information.

```
const amt = 1;
```

Then, Pay becomes :

```
Auctioneer.pay([[amt, nftId]])
```

Now the DApp has necessary information to run an auction. What's left is the auction logic. But before we do that, let's notify the Auctioneer that the auction is ready.

```
Auctioneer.interact.auctionReady();
```