WORKSHOP

Build UTXO on Substrate

Nicole Zhu

@nczhu
nicole@parity.io

Amar Singh

@AmarRSingh
amar@parity.io

Dmitriy Kashitsyn

@0x7CFE
dmitriy@parity.io |

UTXO Workshop Setup

1. Clone: https://github.com/nczhu/utxo-workshop
2. `git checkout workshop`
3. cd root directory of utxo project
4. `./build.sh` and `cargo build --release` (This will take a while)

Don’t peek at the code yet!

parity

You will learn

- How to implement the UTXO model on Substrate
- How to secure UTXO transactions against attacks
- How to seed genesis block with UTXOs
- How to reward block validators in this environment
- How to customize transaction pool logic on Substrate
- Good coding patterns for working with Substrate & Rust

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- [Optional] Challenge 3: Build an extension

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- Challenge 3: Build an extension

Why build UTXO on Substrate

- UTXO vs accounts based model (Substrate)
- Buildable on top of Substrate
- Scalability Possibilities — Since it is possible to process multiple UTXOs at the same time, it enables parallel transactions and encourages scalability innovation.

Let's do this in Rust!

parity

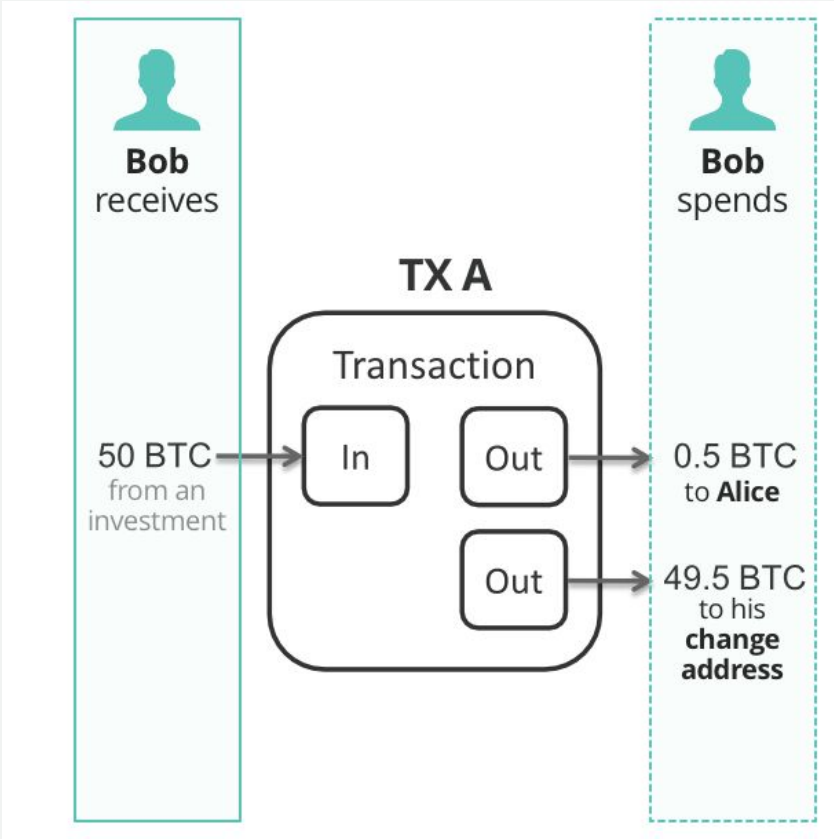
What is UTXO

- Foundation of the Bitcoin ecosystem
- Stands for “Unspent Transaction Output”
- Can only be spent as a whole; cannot be divided
- It’s like travelers’ checks

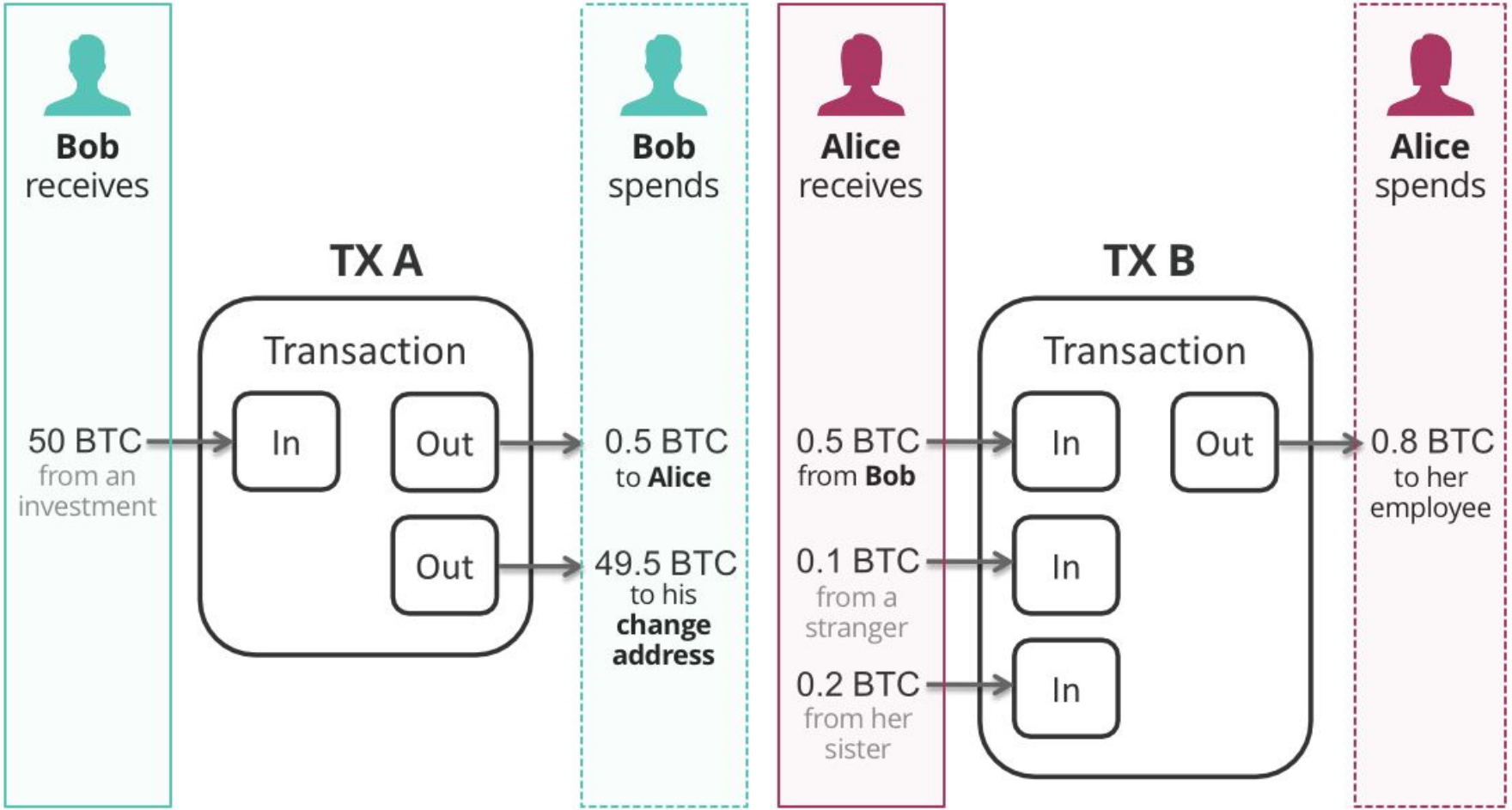
parity

UTXO Model Example

1. Bob owns a utxo of 50 BTC
2. Bob wants to give Alice 0.5 BTC
3. Bob creates a transaction:
    - Input: his 50 BTC utxo
    - Outputs:
        - 0.5 BTC utxo for Alice
        - 49.5 BTC utxo for himself
4. Now, two utxo exists in the chain:
    - Bob: 49.5 BTC
    - Alice: 0.5 BTC



Source: https://freedomnode.com/



Source: https://freedomnode.com/

parity

UTXO Transaction

UTXO Input {

prev_output: *hash of trx + index*

unlock_script: *Signed prev_output*

sequence_num: *a number*

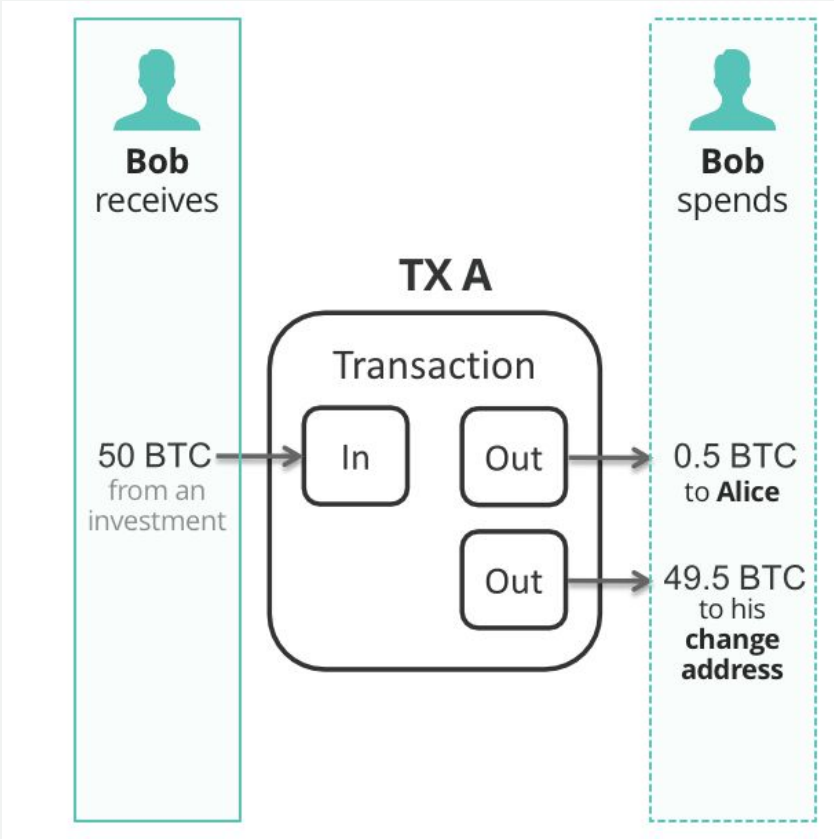
}

UTXO Output {

value: *utxo value*

locking_script: *key of next owner*

}



Source: https://freedomnode.com/

UTXO Transaction Validation

- Every transaction must prove that the sum of its inputs are greater than the sum of its outputs.
- Every referenced input must be valid and not yet spent.
- The transaction must have a signature matching the owner of the input for every input.

Hint: You might want to note these rules for your upcoming exercise!

Questions?

parity

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- Challenge 3: Build an extension

Demo

1. Create an account Nicole from seed
2. Give Nicole some balance to be able to send extrinsics
3. Check that genesis block grants UTXO(s) to Nicole
4. Have Nicole spend that UTXO on herself, then “discard” the rest
5. Check that the UTXO transaction was successful and included in the respective block

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- Challenge 3: Build an extension

The following implementation is not production ready. Use at your own discretion!

parity

UTXO Workshop Setup

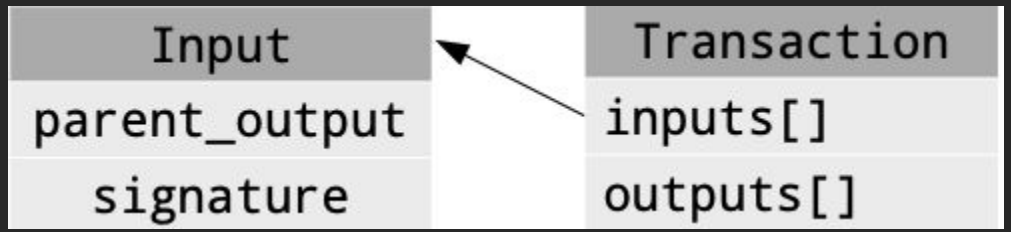
1. Clone: https://github.com/nczhu/utxo-workshop
2. git checkout workshop

Don’t peek at the code yet!

parity

Transaction Input Struct

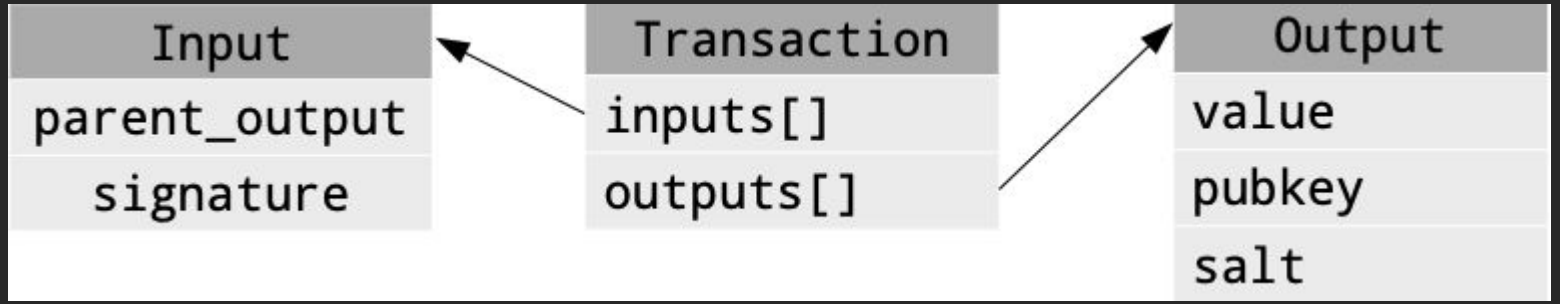
```rust
/// Single transaction input that refers to one UTXO
pub struct TransactionInput {
    /// Reference to an UTXO to be spent
    pub parent_output: H256,

    /// Proof that transaction owner is authorized to spend referred UTXO
    pub signature: Signature,
}
```



parity

Transaction Output Struct

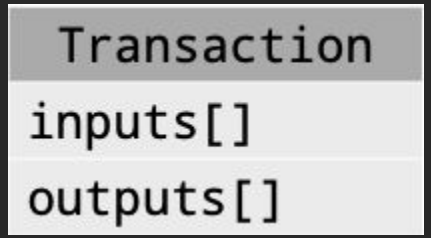
```
/// Single transaction output to create upon transaction dispatch
pub struct TransactionOutput {
    pub value: Value,
    /// Public key associated with this output. In order to spend this output owner must provide a
proof by hashing whole `TransactionOutput` and signing it with a corresponding private key.
    pub pubkey: H256,
    /// Unique (potentially random) value used to distinguish this particular output from others
addressed to the same public key with the same value. Prevents potential replay attacks.
    pub salt: u64,
}
```



parity

Transaction Struct

```rust
/// Single transaction to be dispatched
pub struct Transaction {
    /// UTXOs to be used as inputs for current transaction
    pub inputs: Vec<TransactionInput>,

    /// UTXOs to be created as a result of current transaction dispatch
    pub outputs: Vec<TransactionOutput>,
}
```

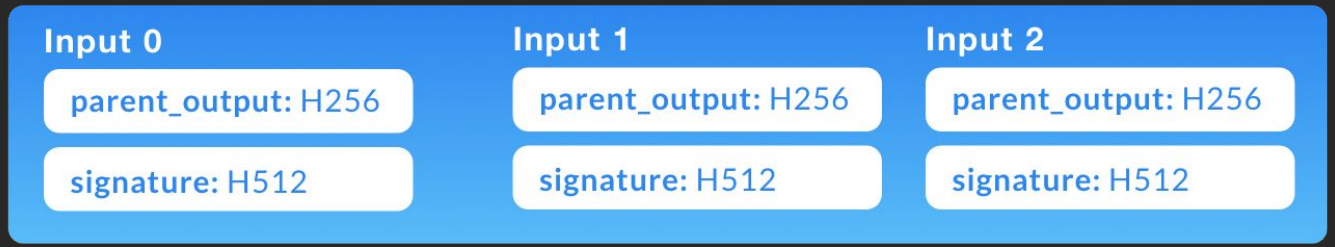


parity

UTXO Transaction on Substrate

Transaction Inputs

Vec<TransactionInput>



Transaction Outputs

Vec<TransactionInput>

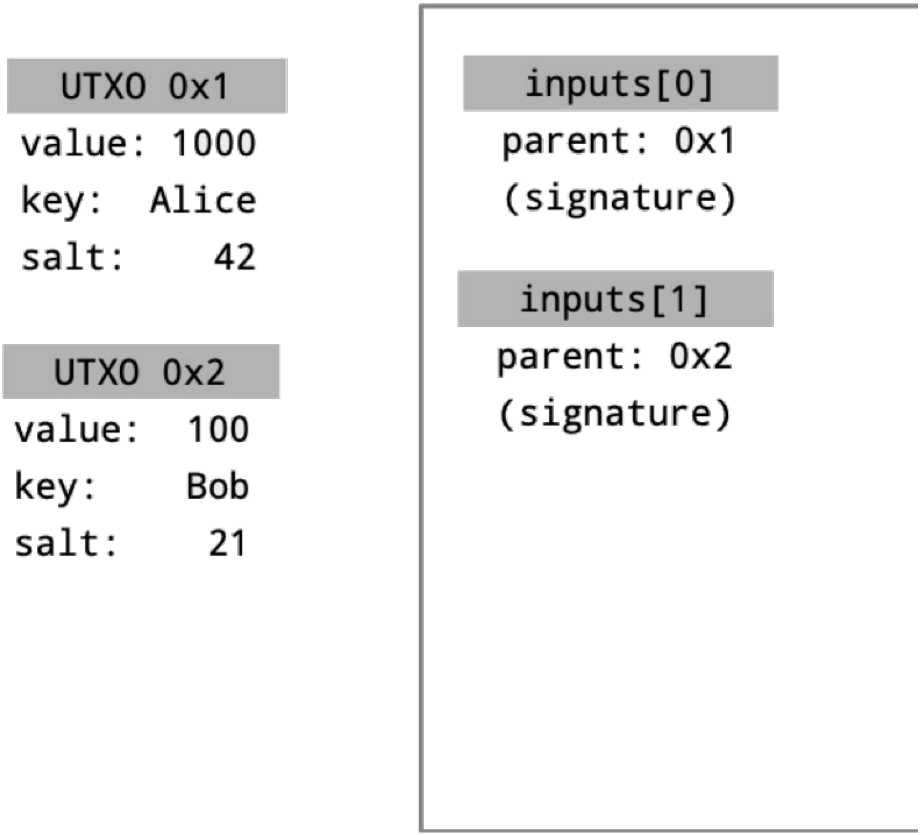
parity

Transaction Example

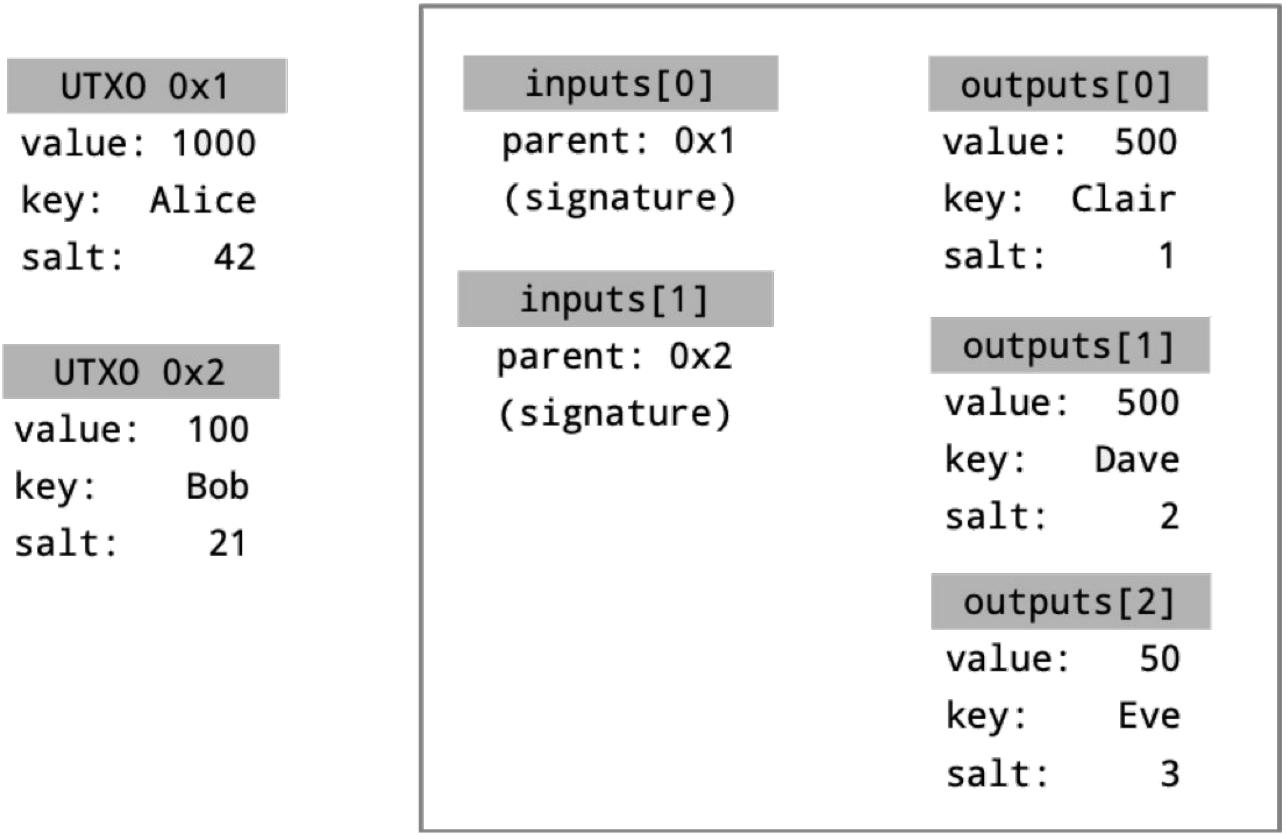
```
UTXO 0x1
value: 1000
key:   Alice
salt:    42
```

```
UTXO 0x2
value:  100
key:    Bob
salt:    21
```

Transaction Example



Transaction Example



UTXO is stored in UnspentOutputs

```
UnspentOutputs build(|config: &GenesisConfig<T>| {
        config.initial_utxo
            .iter()
            .cloned()
            .map(|u| (BlakeTwo256::hash_of(&u), u))
            .collect::<Vec<_>>()
    }): map H256 => Option<TransactionOutput>;
```

Initialize UnspentOutputs for Genesis block

In Chain_spec.rs:

```
    utxo: Some(UtxoConfig {
           initial_utxo: vec![
                utxo::TransactionOutput {
                    value: utxo::Value::max_value(),
                    pubkey: H256::zero(),
                    salt: 0,
                }
            ],
           ..Default::default()
     }),
```

parity

Questions?

parity

Implement transaction logic with utxo::execute( )

```
/// Dispatch a single transaction and update UTXO set accordingly
        pub fn execute(origin, transaction: Transaction) -> Result {
            ensure_inherent(origin)?;

            // Verify the transaction
            let leftover = match Self::check_transaction(&transaction)? {
                CheckInfo::Totals{input, output} => input - output,
                CheckInfo::MissingInputs(_) => return Err("Invalid transaction inputs")
            };

            // Update unspent outputs
            Self::update_storage(&transaction, leftover)?;
            ...
            Ok(())
        }
```

parity

Update storage updates UnspentOutputs

```
/// Update storage to reflect changes made by transaction
   fn update_storage(transaction: &Transaction, leftover: Value) -> Result {
      ...
       // Storing updated leftover value
       for input in &transaction.inputs {
          <UnspentOutputs<T>>::remove(input.parent_output);
       }

       // Add new UTXO to be used by future transactions
       for output in &transaction.outputs {
           let hash = BlakeTwo256::hash_of(output);
          <UnspentOutputs<T>>::insert(hash, output);
       }
       Ok(())
   }
```

parity

Note: Check_transaction returns a CheckInfo enum

```
/// Information collected during transaction verification
pub enum CheckInfo<'a> {
    /// Combined value of all inputs and outputs
    Totals { input: Value, output: Value },

    /// Some referred UTXOs were missing
    MissingInputs(Vec<&'a H256>),
}
```

parity

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- Challenge 3: Build an extension

UTXO validates transactions as follows:

- Check signatures
- Check all inputs are unspent
- Check input == output value
- Set Input to “spent”

parity

Challenge 1: Check Transactions

1. Go to the workshop branch
2. Extend `check_transaction()` to make the malicious tests correctly fail.

`cargo test -p utxo-runtime -- --nocapture`

```
running 7 tests
test utxo::tests::attack_by_double_generating_output ... ok
test utxo::tests::attack_by_double_counting_input ... ok
test utxo::tests::attack_by_over_spending ... ok
test utxo::tests::attack_with_empty_transactions ... ok
test utxo::tests::attack_by_permanently_sinking_outputs ... ok
test utxo::tests::valid_transaction ... ok
test utxo::tests::attack_with_invalid_signature ... ok

test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Transaction Checks

- ❑ Inputs and outputs are not empty
- ❑ All inputs match to existing, unspent and unlocked outputs
- ❑ Each input is used exactly once
- ❑ Each output is defined exactly once and has nonzero value
- ❑ Total output value must not exceed total input value
- ❑ New outputs do not collide with existing ones
- ❑ Sum of input and output values does not overflow
- ❑ Provided signatures are valid

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- Challenge 3: Build an extension

Distribute leftover utxo among Authorities

- Leftover is calculated from transactions at the end of every block with `on_finalize`
- Leftover is evenly distributed among authorities with `spend_dust`

UTXO locking

- Bitcoin nodes process transactions but do not share the same state db. You cannot stake or lock a token.
- In Substrate, you can easily implement locking logic to give your tokens more utility.

```
LockedOutputs: map H256 => Option<LockStatus<T::BlockNumber>>;

pub enum LockStatus<BlockNumber> {
    Locked,
    LockedUntil(BlockNumber),
}
```

parity

Questions?

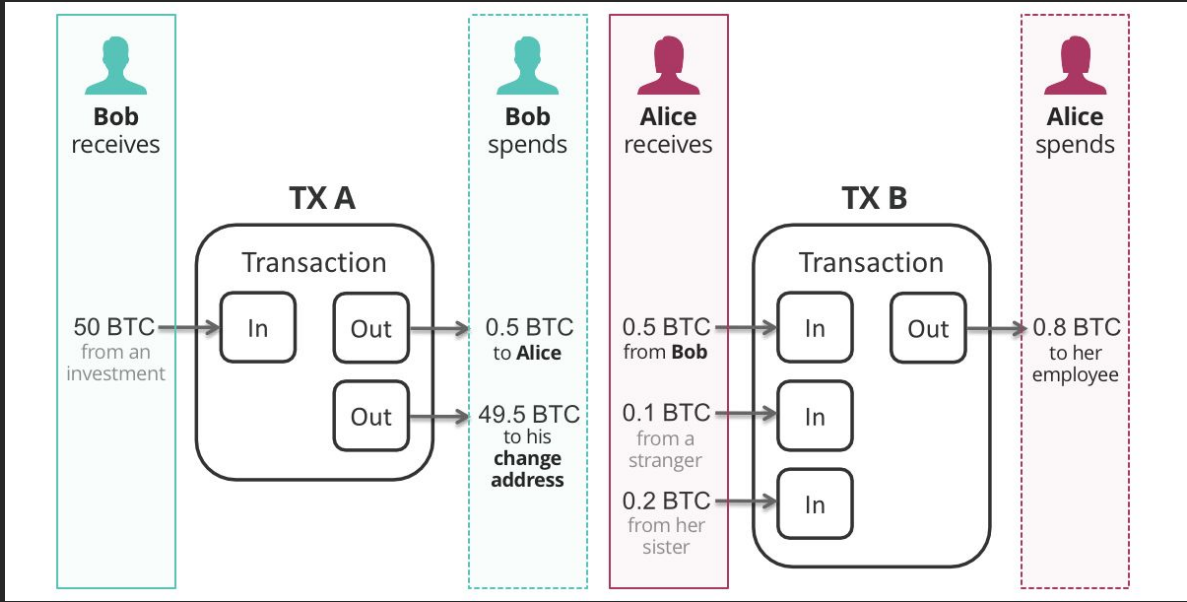
parity

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- Challenge 3: Build an extension

Challenge 2: Transaction ordering

Consider the Scenario where transaction B depends on transaction A, but B arrives to the pool before A



parity

TaggedTransactionQueue Trait

```
fn validate_transaction(
    &self,
    at: &BlockId<Block>,
    tx: <Block as BlockT>::Extrinsic
) -> Result<TransactionValidity, Error>
```

parity

TransactionValidity

```
pub enum TransactionValidity {
    Invalid(i8),
    Valid {
        priority: TransactionPriority,
        requires: Vec<TransactionTag>,
        provides: Vec<TransactionTag>,
        longevity: TransactionLongevity,
    },
    Unknown(i8),
}
```

parity

Transaction Types

- Q: What should you use as the require tag?
- Q: What do you do if a transaction is:
  - Missing outputs?
  - Erroring?
  - valid?

Let’s implement!

Agenda

- Overview of UTXO Model
- Demo of what you'll build
- Architecting UTXO on Substrate
- Challenge 1: Thwart malicious transactions
- Leftover and UTXO locking mechanisms
- Challenge 2: Design transaction pool logic
- Challenge 3: Build an extension

Challenge 3: Build an extension

Potential extensions:

- Implement transaction longevity
- Implement coinbase transactions

We’re hiring!

parity.io/jobs

parity

Parity updates and events

parity.io/newsletter

parity

Questions?

john@parity.io
@johndoe

parity