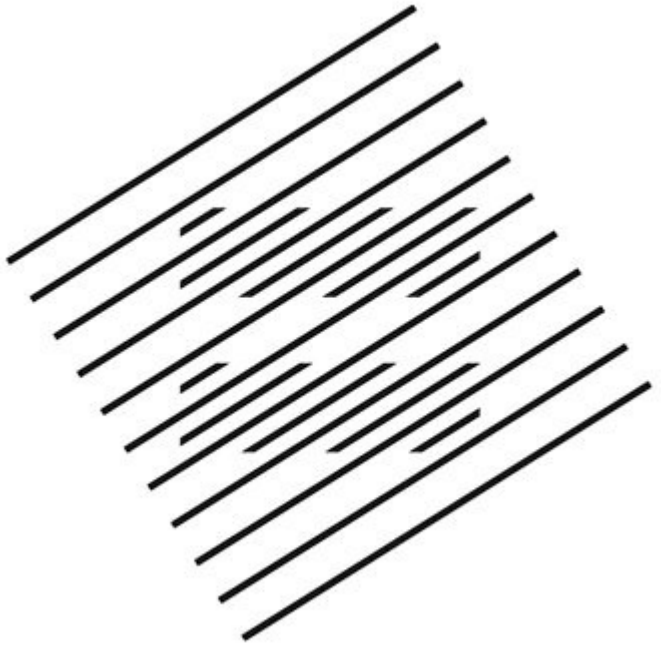
Dmitry Kashitsyn aka 0x7CFE

PoC UTXO chain on Parity Substrate



Polkadot Meetup
Moscow
2019

Disclaimer: this implementation is NOT

- Bitcoin compatible
- Production ready
- Formally verified
- DoS tolerant

Concepts and stuff that were implemented:

Concepts and stuff that were implemented:

- UTXO

Concepts and stuff that were implemented:

- UTXO
- Transaction validity checks

Concepts and stuff that were implemented:

- UTXO
- Transaction validity checks
- ed25519 key per UTXO

Concepts and stuff that were implemented:

- UTXO
- Transaction validity checks
- ed25519 key per UTXO
- "Premined" UTXO set to be sealed in Genesis

Concepts and stuff that were implemented:

- UTXO
- Transaction validity checks
- ed25519 key per UTXO
- "Premined" UTXO set to be sealed in Genesis
- Transaction fee affects priority

Concepts and stuff that were implemented:

- UTXO
- Transaction validity checks
- ed25519 key per UTXO
- "Premined" UTXO set to be sealed in Genesis
- Transaction fee affects priority
- Fees are redistributed among validators

Concepts and stuff that were implemented:

- UTXO
- Transaction validity checks
- ed25519 key per UTXO
- "Premined" UTXO set to be sealed in Genesis
- Transaction fee affects priority
- Fees are redistributed among validators
- Transaction ordering based on tags

Concepts and stuff that were implemented:

- UTXO
- Transaction validity checks
- ed25519 key per UTXO
- "Premined" UTXO set to be sealed in Genesis
- Transaction fee affects priority
- Fees are redistributed among validators
- Transaction ordering based on tags
- UTXO locking (for staking, etc.)

What is UTXO?

What is UTXO?

- Foundation of the Bitcoin ecosystem

What is UTXO?

- Foundation of the Bitcoin ecosystem
- Stands for “unspent transaction output”

What is UTXO?

- Foundation of the Bitcoin ecosystem
- Stands for “unspent transaction output”
- May be spent only as a whole,
  i.e. could not be divided

What is UTXO?

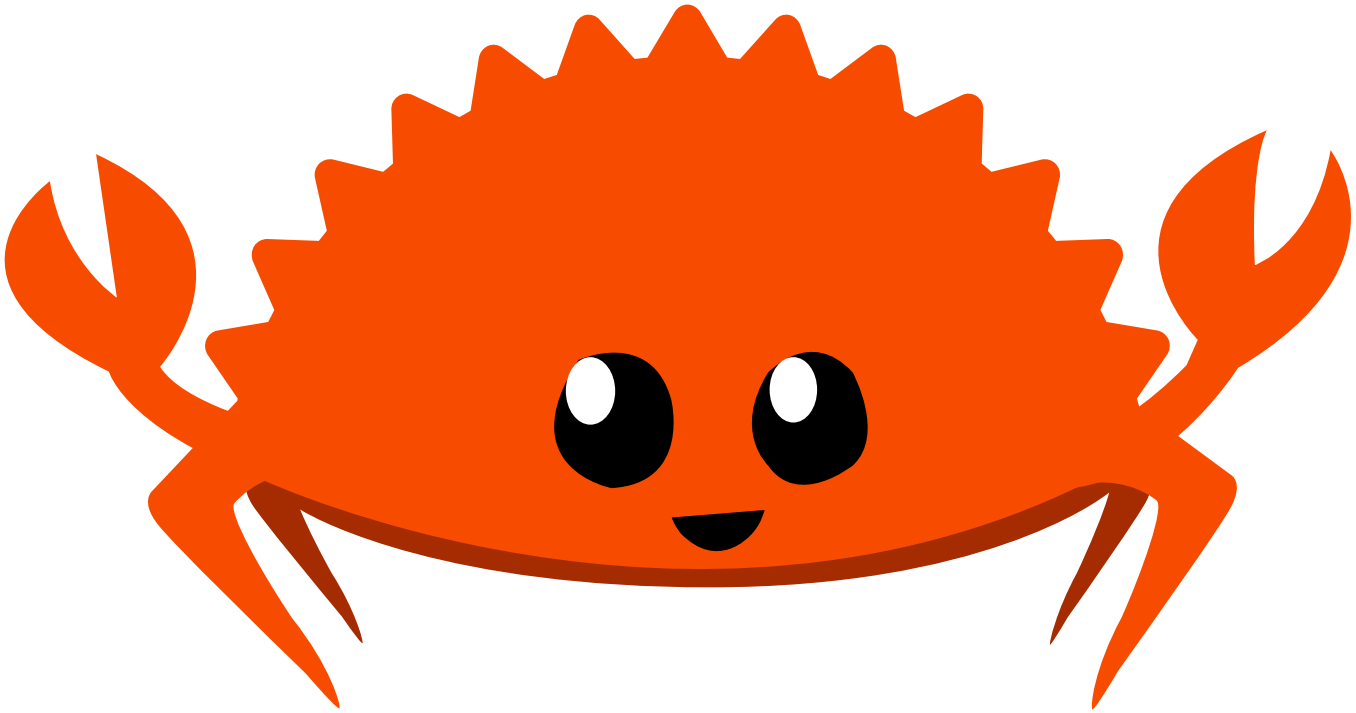
- Foundation of the Bitcoin ecosystem
- Stands for “unspent transaction output”
- May be spent only as a whole,
  i.e. could not be divided
- In a nutshell, it’s like a cache money

What is UTXO?

- Foundation of the Bitcoin ecosystem
- Stands for “unspent transaction output”
- May be spent only as a whole,
  i.e. could not be divided
- In a nutshell, it’s like a cache money
- Or rather, as a travel checks

Let’s do it…

Let’s do it… in Rust!



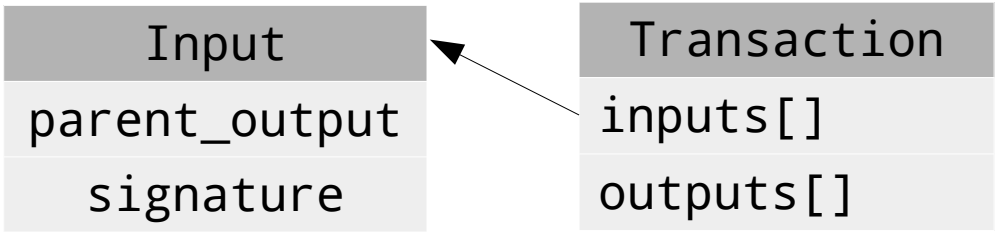
utxo::Transaction

```rust
/// Single transaction to be dispatched
#[cfg_attr(feature = "std", derive(Serialize, Deserialize, Debug))]
#[derive(PartialEq, Eq, PartialOrd, Ord, Default, Clone, Encode, Decode, Hash)]
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>,
}
```

Transaction
inputs[]
outputs[]

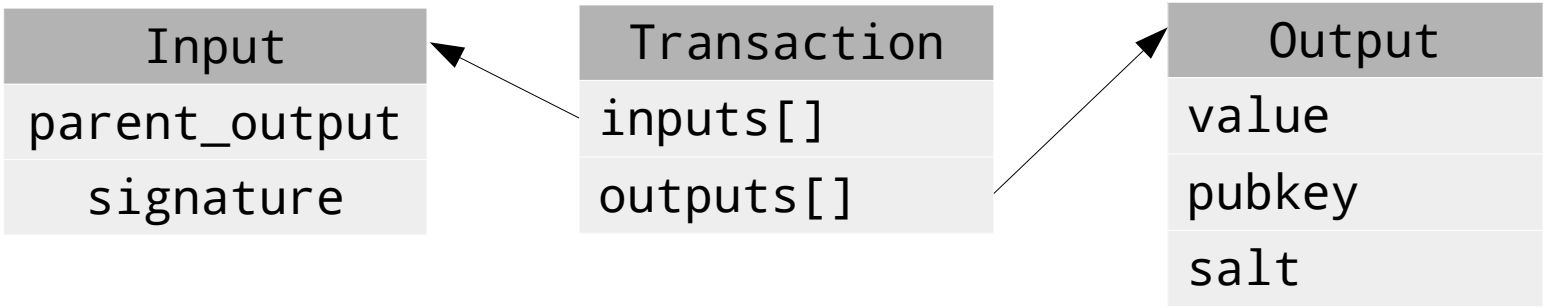
utxo::TransactionInput

```rust
/// Single transaction input that refers to one UTXO
#[cfg_attr(feature = "std", derive(Serialize, Deserialize, Debug))]
#[derive(PartialEq, Eq, PartialOrd, Ord, Default, Clone, Encode, Decode, Hash)]
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,
}
```



utxo::TransactionOutput

```rust
/// Single transaction output to create upon transaction dispatch
#[cfg_attr(feature = "std", derive(Serialize, Deserialize, Debug))]
#[derive(PartialEq, Eq, PartialOrd, Ord, Default, Clone, Encode, Decode, Hash)]
pub struct TransactionOutput {
    /// Value associated with this output
    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: u32,
}
```



Transaction Example

Transaction Example

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

Transaction Example

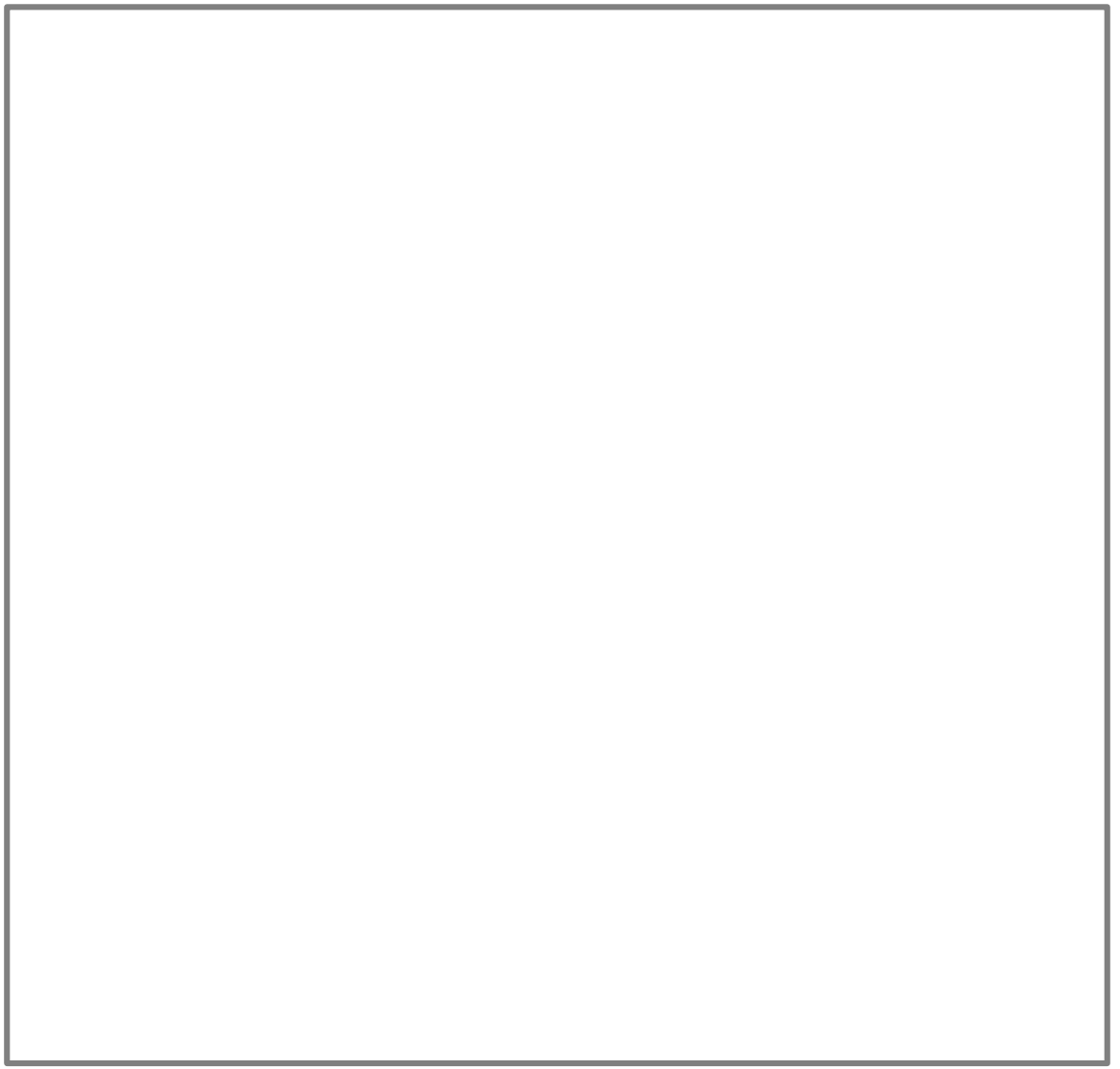
UTXO 0x1
value: 1000
key: Alice
salt: 42

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

Transaction Example

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

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



Transaction Example

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

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

inputs[0]
parent: 0x1
(signature)

Transaction Example

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

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

inputs[0]
parent: 0x1
(signature)

inputs[1]
parent: 0x2
(signature)

Transaction Example

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

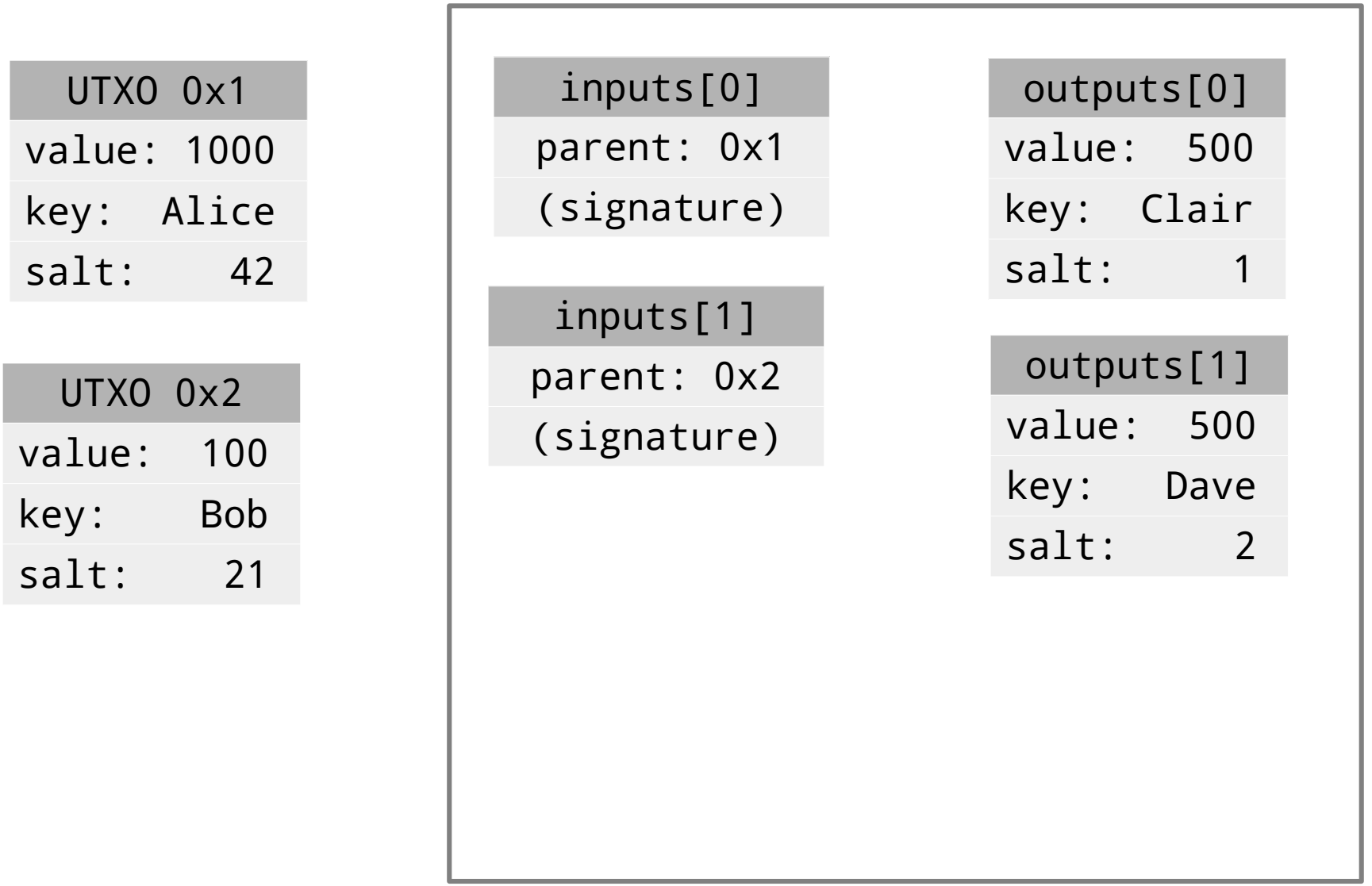
UTXO 0x2
value: 100
key: Bob
salt: 21

inputs[0]
parent: 0x1
(signature)

inputs[1]
parent: 0x2
(signature)

outputs[0]
value: 500
key: Clair
salt: 1

Transaction Example



Transaction Example

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

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

inputs[0]
parent: 0x1
(signature)

inputs[1]
parent: 0x2
(signature)

outputs[0]	
value:	500
key:	Clair
salt:	1

outputs[1]	
value:	500
key:	Dave
salt:	2

outputs[2]	
value:	50
key:	Eve
salt:	3

decl_storage!

```
decl_storage! {
    trait Store for Module<T: Trait> as Utxo {
        /// All valid unspent transaction outputs are stored in this map.
        /// Initial set of UTXO is populated from the list stored in genesis.
        UnspentOutputs build(|config: &GenesisConfig<T>| {
            config.initial_utxo
                .iter()
                .cloned()
                .map(|u| (BlakeTwo256::hash_of(&u), u))
                .collect::<Vec<_>>()
        }): map H256 ⇒ Option<TransactionOutput>;

        /// Total leftover value to be redistributed among authorities.
        /// It is accumulated during block execution and then drained
        /// on block finalization.
        LeftoverTotal: Value;

        /// Outputs that are locked
        LockedOutputs: map H256 ⇒ Option<LockStatus<T::BlockNumber>>;
    }

    add_extra_genesis {
        config(initial_utxo): Vec<TransactionOutput>;
    }
}
```

decl_storage!

```
decl_storage! {
    trait Store for Module<T: Trait> as Utxo {
        /// All valid unspent transaction outputs are stored in this map.
        /// Initial set of UTXO is populated from the list stored in genesis.
        UnspentOutputs build(|config: &GenesisConfig<T>| {
            config.initial_utxo
                .iter()
                .cloned()
                .map(|u| (BlakeTwo256::hash_of(&u), u))
                .collect::<Vec<_>>()
        }): map H256 ⇒ Option<TransactionOutput>;

        /// Total leftover value to be redistributed among authorities.
        /// It is accumulated during block execution and then drained
        /// on block finalization.
        LeftoverTotal: Value;

        /// Outputs that are locked
        LockedOutputs: map H256 ⇒ Option<LockStatus<T::BlockNumber>>;
    }

    add_extra_genesis {
        config(initial_utxo): Vec<TransactionOutput>;
    }
}
```

State Transition Function

$$new\_state = stf\,(old\_state, transaction)$$

utxo::Module

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        /// Dispatch a single transaction and update UTXO set accordingly
        pub fn execute(origin, transaction: Transaction) -> Result {
            ensure_inherent(origin)?;

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

            Self::update_storage(&transaction, leftover)?;
            Self::deposit_event(Event::TransactionExecuted(transaction));

            Ok(())
        }

        /// Handler called by the system on block finalization
        fn on_finalise() {
            let authorities: Vec<_> = Consensus::authorities()
                .iter().map(|&a| a.into()).collect();

            Self::spend_leftover(&authorities);
        }
    }
}
```

utxo::Module::execute()

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        /// Dispatch a single transaction and update UTXO set accordingly
        pub fn execute(origin, transaction: Transaction) -> Result {
            ensure_inherent(origin)?;

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

            Self::update_storage(&transaction, leftover)?;
            Self::deposit_event(Event::TransactionExecuted(transaction));

            Ok(())
        }

        /// Handler called by the system on block finalization
        fn on_finalise() {
            let authorities: Vec<_> = Consensus::authorities()
                .iter().map(|&a| a.into()).collect();

            Self::spend_leftover(&authorities);
        }
    }
}
```

utxo::Module::on_finalize()

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        /// Dispatch a single transaction and update UTXO set accordingly
        pub fn execute(origin, transaction: Transaction) -> Result {
            ensure_inherent(origin)?;

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

            Self::update_storage(&transaction, leftover)?;
            Self::deposit_event(Event::TransactionExecuted(transaction));

            Ok(())
        }

        /// Handler called by the system on block finalization
        fn on_finalise() {
            let authorities: Vec<_> = Consensus::authorities()
                .iter().map(|&a| a.into()).collect();

            Self::spend_leftover(&authorities);
        }
    }
}
```

utxo::Module::check_transaction()

```
/// 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>),
}

/// Result of transaction verification
pub type CheckResult<'a> = rstd::result::Result<CheckInfo<'a>, &'static str>;

/// Check transaction for validity
pub fn check_transaction(transaction: &Transaction) → CheckResult<'_> {
    ...
}
```

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

utxo::Module::update_storage()

```
/// Update storage to reflect changes made by transaction
fn update_storage(transaction: &Transaction, leftover: Value) → Result {
    // Calculate new leftover total
    let new_total = <LeftoverTotal<T>>::get()
        .checked_add(leftover)
        .ok_or("leftover overflow")?;

    // Storing updated leftover value
    <LeftoverTotal<T>>::put(new_total);

    // Remove all used UTXO since they are now spent
    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(())
}
```

utxo::Module::update_storage()

```
/// Update storage to reflect changes made by transaction
fn update_storage(transaction: &Transaction, leftover: Value) → Result {
    // Calculate new leftover total
    let new_total = <LeftoverTotal<T>>::get()
        .checked_add(leftover)
        .ok_or("leftover overflow")?;

    // Storing updated leftover value
    <LeftoverTotal<T>>::put(new_total);

    // Remove all used UTXO since they are now spent
    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(())
}
```

utxo::Module::update_storage()

```
/// Update storage to reflect changes made by transaction
fn update_storage(transaction: &Transaction, leftover: Value) → Result {
    // Calculate new leftover total
    let new_total = <LeftoverTotal<T>>::get()
        .checked_add(leftover)
        .ok_or("leftover overflow")?;

    // Storing updated leftover value
    <LeftoverTotal<T>>::put(new_total);

    // Remove all used UTXO since they are now spent
    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(())
}
```

utxo::Module::spend_leftover()

```
/// Redistribute combined leftover value evenly among authorities
fn spend_leftover(authorities: &[H256]) {
    let leftover = <LeftoverTotal<T>>::take();
    let share_value = leftover / authorities.len() as Value;

    if share_value == 0 { return }

    for authority in authorities {
        let utxo = TransactionOutput {
            pubkey: *authority,
            value: share_value,
            salt: System::block_number() as u32,
        };

        let hash = BlakeTwo256::hash_of(&utxo);

        if !<UnspentOutputs<T>>::exists(hash) {
            <UnspentOutputs<T>>::insert(hash, utxo);

            runtime_io::print("leftover share sent to");
            runtime_io::print(hash.as_fixed_bytes() as &[u8]);
        } else {
            runtime_io::print("leftover hash collision");
        }
    }
}
```

utxo::Module::spend_leftover()

```
/// Redistribute combined leftover value evenly among authorities
fn spend_leftover(authorities: &[H256]) {
    let leftover = <LeftoverTotal<T>>::take();
    let share_value = leftover / authorities.len() as Value;

    if share_value == 0 { return }

    for authority in authorities {
        let utxo = TransactionOutput {
            pubkey: *authority,
            value: share_value,
            salt: System::block_number() as u32,
        };

        let hash = BlakeTwo256::hash_of(&utxo);

        if !<UnspentOutputs<T>>::exists(hash) {
            <UnspentOutputs<T>>::insert(hash, utxo);

            runtime_io::print("leftover share sent to");
            runtime_io::print(hash.as_fixed_bytes() as &[u8]);
        } else {
            runtime_io::print("leftover hash collision");
        }
    }
}
```

utxo::Module::spend_leftover()

```
/// Redistribute combined leftover value evenly among authorities
fn spend_leftover(authorities: &[H256]) {
    let leftover = <LeftoverTotal<T>>::take();
    let share_value = leftover / authorities.len() as Value;

    if share_value == 0 { return }

    for authority in authorities {
        let utxo = TransactionOutput {
            pubkey: *authority,
            value: share_value,
            salt: System::block_number() as u32,
        };

        let hash = BlakeTwo256::hash_of(&utxo);

        if !<UnspentOutputs<T>>::exists(hash) {
            <UnspentOutputs<T>>::insert(hash, utxo);

            runtime_io::print("leftover share sent to");
            runtime_io::print(hash.as_fixed_bytes() as &[u8]);
        } else {
            runtime_io::print("leftover hash collision");
        }
    }
}
```

utxo::Module::spend_leftover()

```
/// Redistribute combined leftover value evenly among authorities
fn spend_leftover(authorities: &[H256]) {
    let leftover = <LeftoverTotal<T>>::take();
    let share_value = leftover / authorities.len() as Value;

    if share_value == 0 { return }

    for authority in authorities {
        let utxo = TransactionOutput {
            pubkey: *authority,
            value: share_value,
            salt: System::block_number() as u32,
        };

        let hash = BlakeTwo256::hash_of(&utxo);

        if !<UnspentOutputs<T>>::exists(hash) {
            <UnspentOutputs<T>>::insert(hash, utxo);

            runtime_io::print("leftover share sent to");
            runtime_io::print(hash.as_fixed_bytes() as &[u8]);
        } else {
            runtime_io::print("leftover hash collision");
        }
    }
}
```

decl_event!

```
decl_event!(
    pub enum Event {
        /// Transaction was executed successfully
        TransactionExecuted(Transaction),
    }
);

fn deposit_event(event: Event) {
    let event = <T as Trait>::Event::from(event).into();
    <system::Module<T>>::deposit_event(event);
}
```

Transaction Ordering

- Transaction 1 { inputs: **A**; outputs: **B**, **C** }
- Transaction 2 { inputs: **B**; outputs: **D** }
- Transaction 3 { inputs: **C**, **D**; outputs: **E** }

Transaction Ordering

- Transaction 1 { inputs: **A**; outputs: **B**, **C** }
- Transaction 2 { inputs: **B**; outputs: **D** }
- Transaction 3 { inputs: **C**, **D**; outputs: **E** }

Tagged Transactions

- Transaction 1 { requires: **A** or ∅; provides: **B**, **C** }
- Transaction 2 { requires: **B**; provides: **D** }
- Transaction 3 { requires: **C**, **D**; provides: **E** }

Links

- https://github.com/paritytech/substrate-node-template/pull/17

- blog.parity.io
  - Substrate in a Nutshell
  - Why Rust?

Thank you!