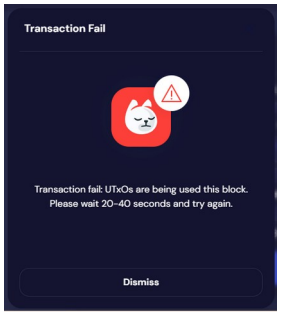
A concurrent DEX for Cardano

Peter Brühwiler

BscThesis

16.12.21

The Concurrency Issue



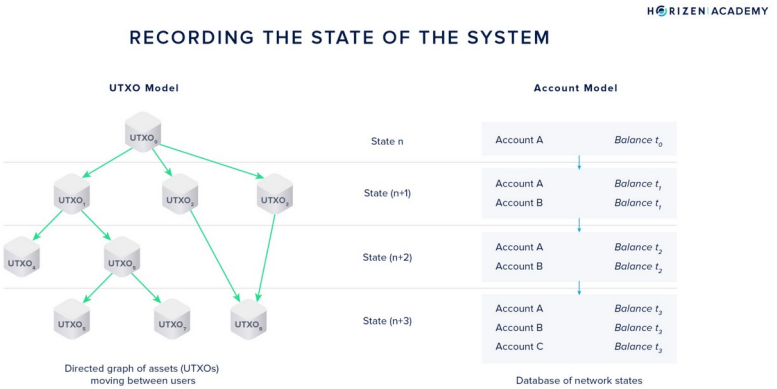
Minswap blog post

Known about it for long,
Only testnet
It’s not a fundamental flaw, but is simply a design challenge that must be addressed.

Took longer than many teams expected: In summer began the Alonzo Era,
But had to change Thesis topic

known about the concurrency challenge since we first began building on Cardano over 6 months ago. It’s an issue that every competent team and development lab building DeFi protocols on Cardano must overcome. It’s not a fundamental flaw, but is simply a design challenge that must be addressed.

The UTxO model



Instead of Changing a global state, you spend and create UtxOs

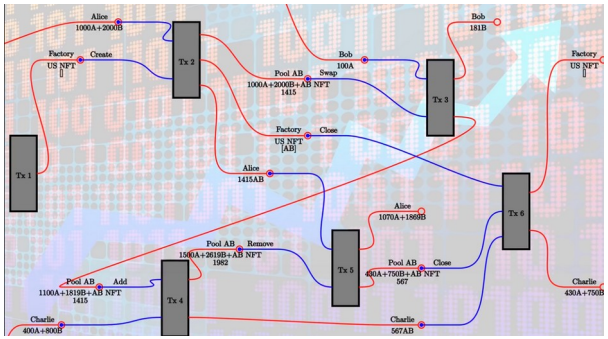
Like Cash: Spend 10.- , you get 5.-, seller gets 5.-

Only available in next Block

Problem arises, if multiple users want to spend the same UtxO.

Why woud they? EUTxO, big innovation of Cardano

Simple AMM style DEX

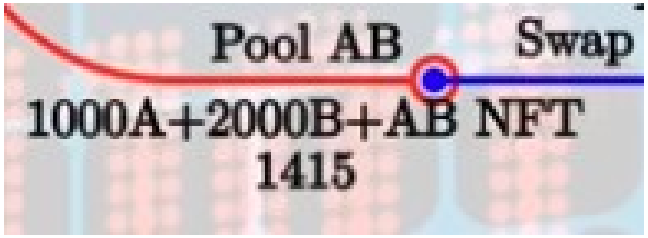


A chain of creations and spendings of UtxOs

Will come back to the whole picture.

But First: The meaning of the two additional elements in Pool AB compared to Bob’s UTxO

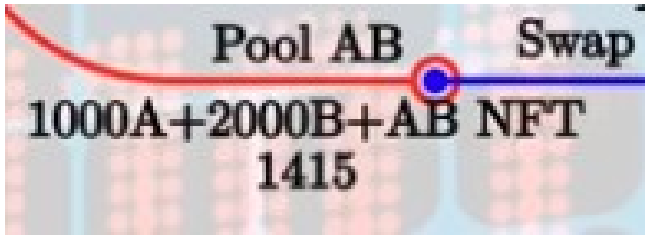
State of an eUTxO



- UTxOs at script addresses have a Datum in addition to the value
- Exchange Rate of the DEX given by the relative values of A and B
- The Datum keeps track of the amount of liquidity tokens issued by this pool
- The Datum is some script-specific data specified by the party who created the output

Redeemer and Context

- The Redeemer is normally used to activate the purpose of the contract, it must be specified by the party consuming the output
- Plus, every script transaction is embedded in a Context
- ..While a Bitcoin Tx must just be signed, a Cardano script Tx must be validated, given the three parameters




```
data ScriptContext
```

Constructors

```
ScriptContext
    scriptContextTxInfo :: TxInfo
    scriptContextPurpose :: ScriptPurpose
```

```
TxInfo
    txInfoInputs :: [TxInInfo]
    txInfoOutputs :: [TxOut]
    txInfoFee :: Value
    txInfoMint :: Value
    txInfoDCert :: [DCert]
    txInfoWdrl :: [(StakingCredential, Integer)]
    txInfoValidRange :: POSIXTimeRange
    txInfoSignatories :: [PubKeyHash]
    txInfoData :: [(DatumHash, Datum)]
    txInfoId :: TxId
```

Simple validator

```
mkValidator :: () -> Integer -> ScriptContext -> Bool
mkValidator _ r _ = traceIfFalse "wrong redeemer" $ r == 42
```

Defines under which conditions a UtxO can be spent.

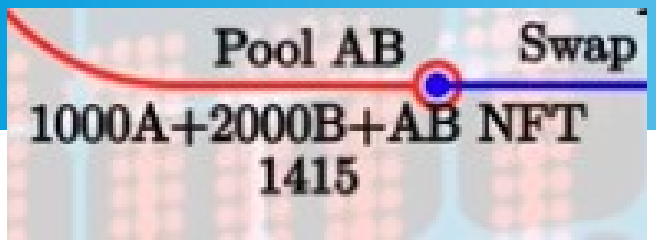
Utxo from PubKey address can be spent if signed by private Key

Utxo from mkValidator adress can be spent if script evaluates to TRUE.

MkValidator address is the hash of the script

...Very simple example: We only care about the Redeemer

Only evaluates to TRUE if I pass along the Redeemer 42 in the spending transaction



Once per block

- So, back to Tx3 with the Redeemer ‘Swap’
- Only one user can consume the state
- He then creates a new UTxO, containing the modified state available in the next block.
- Blocktime: 20s
- Possible solution for a DEX to avoid congestion: Use of the orderbook pattern instead of an AMM, done so by the first live DEX on Cardano, Muesliswap

Order book

```
data Order = Order
  { oCreator     :: !PubKeyHash
  , oBuyCurrency :: !CurrencySymbol
  , oBuyToken    :: !TokenName
  , oBuyAmount   :: !Integer
  }
  deriving (Generic, ToJSON, FromJSON)
```

- No global state, Every order is a single UtxO
- Order is defined in the Datum
- The currency value locked in the UTxO is the inverse of the order
- So: To buy 15 ADA at an exchange rate of 1.5, 10 USDT must be locked in the UTxO.
- In order to consume the UTxO (and send the value to own wallet), the transaction must contain an Output to the PubKey with the specified amount
- This can be made more efficient by putting multiple buy and sell orders in one transaction.

```
{-# INLINABLE mkOrderValidator #-}
mkOrderValidator :: OrderDatum -> OrderAction -> ScriptContext -> Bool
mkOrderValidator od redeemer ctx = case redeemer of
  CancelOrder ->
    traceIfFalse "signature does not match creator in datum" checkSig
  FullMatch ->
    traceIfFalse "expected creator to get all of what she ordered" correctFull
      && traceIfFalse "only matches of pairs of orders allowed" twoParties
```

Batching

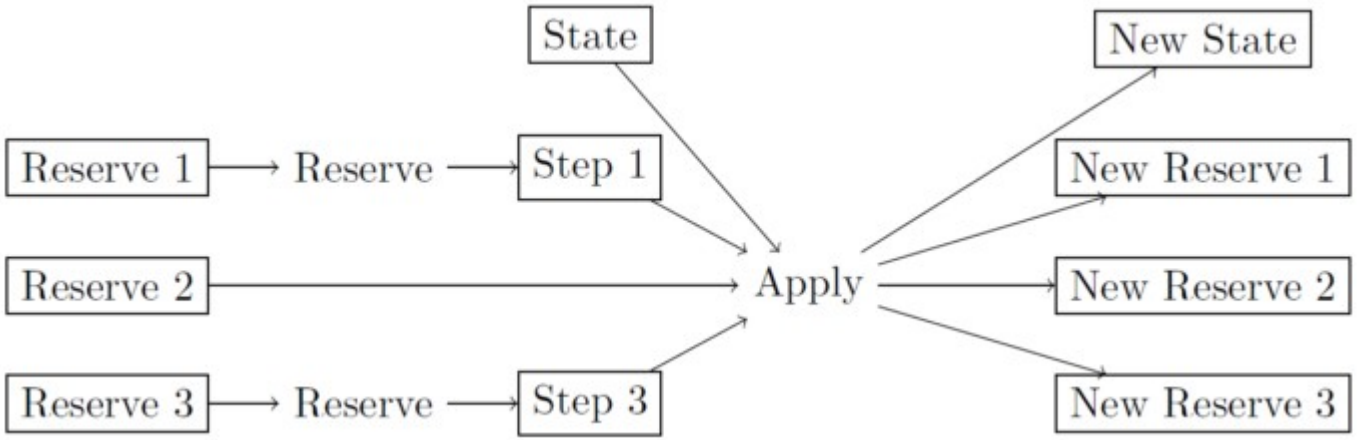
- Problem: No exchange without counterpart.
- So, back to the AMM model, but with ..

  .. multiple state UTxOs. Problem: The smaller the liquidity in a pool, the higher the slippage.

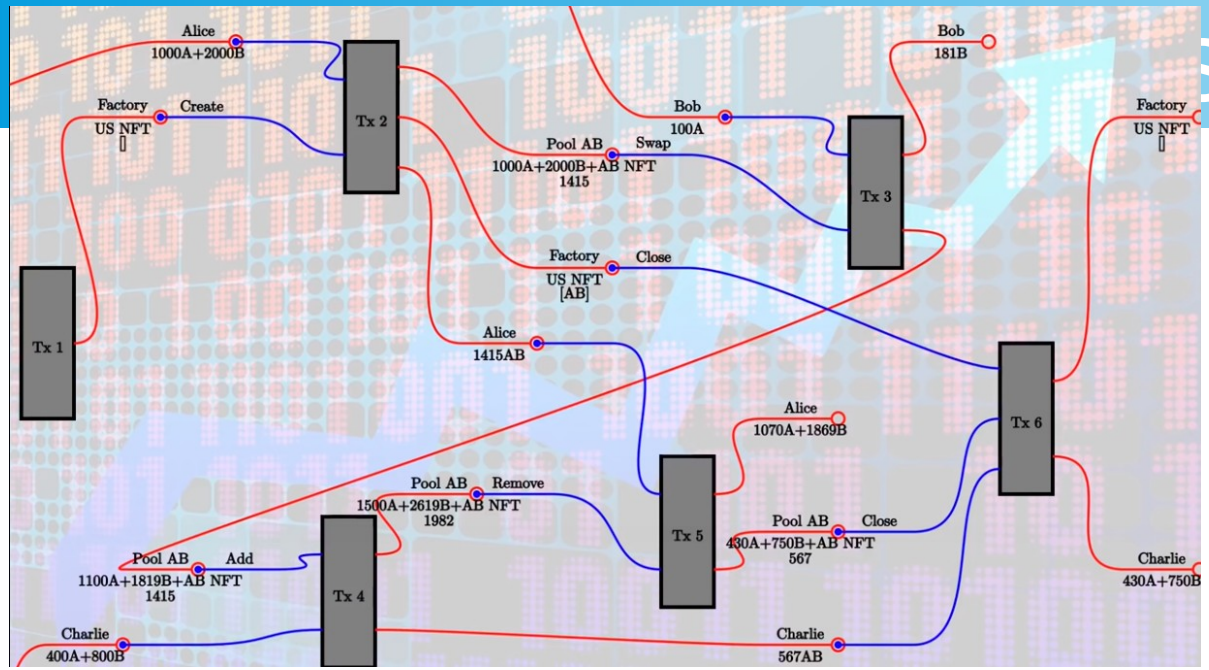
  .. an additional batching layer. That’s the goal most development Teams are pursuing

The Batching Layer

- Instead of building a transaction including the state UTxO, the user makes a reservation. (f. ex by consuming Reserve 1 and creating Step 1, below)
- All the Steps and unused Reserves are consumed together with the state UTxO.
- Unused Reserves must be included to make the transaction deterministic
- A specified number of new reserve UTxOs have to be created for the next round.



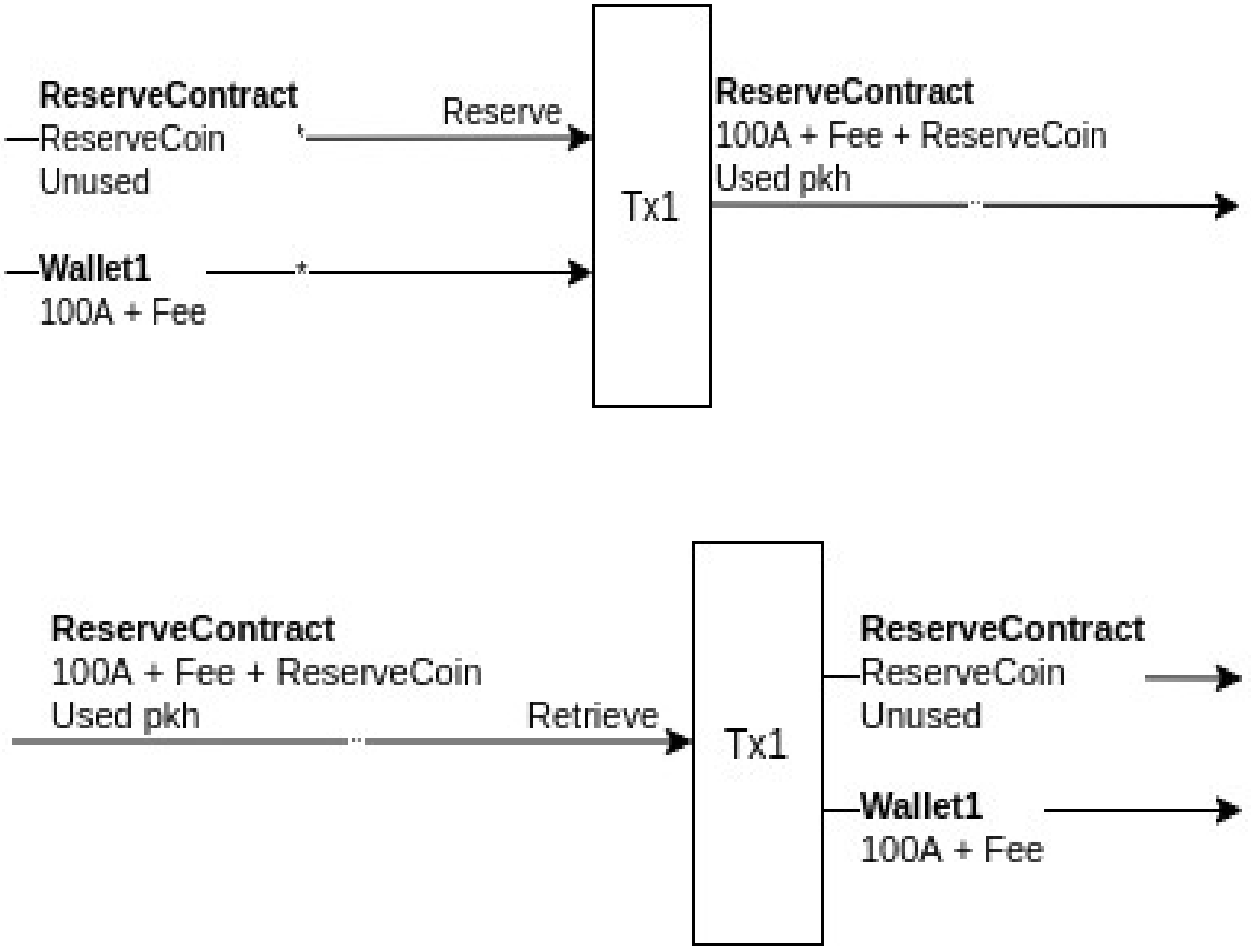
start..



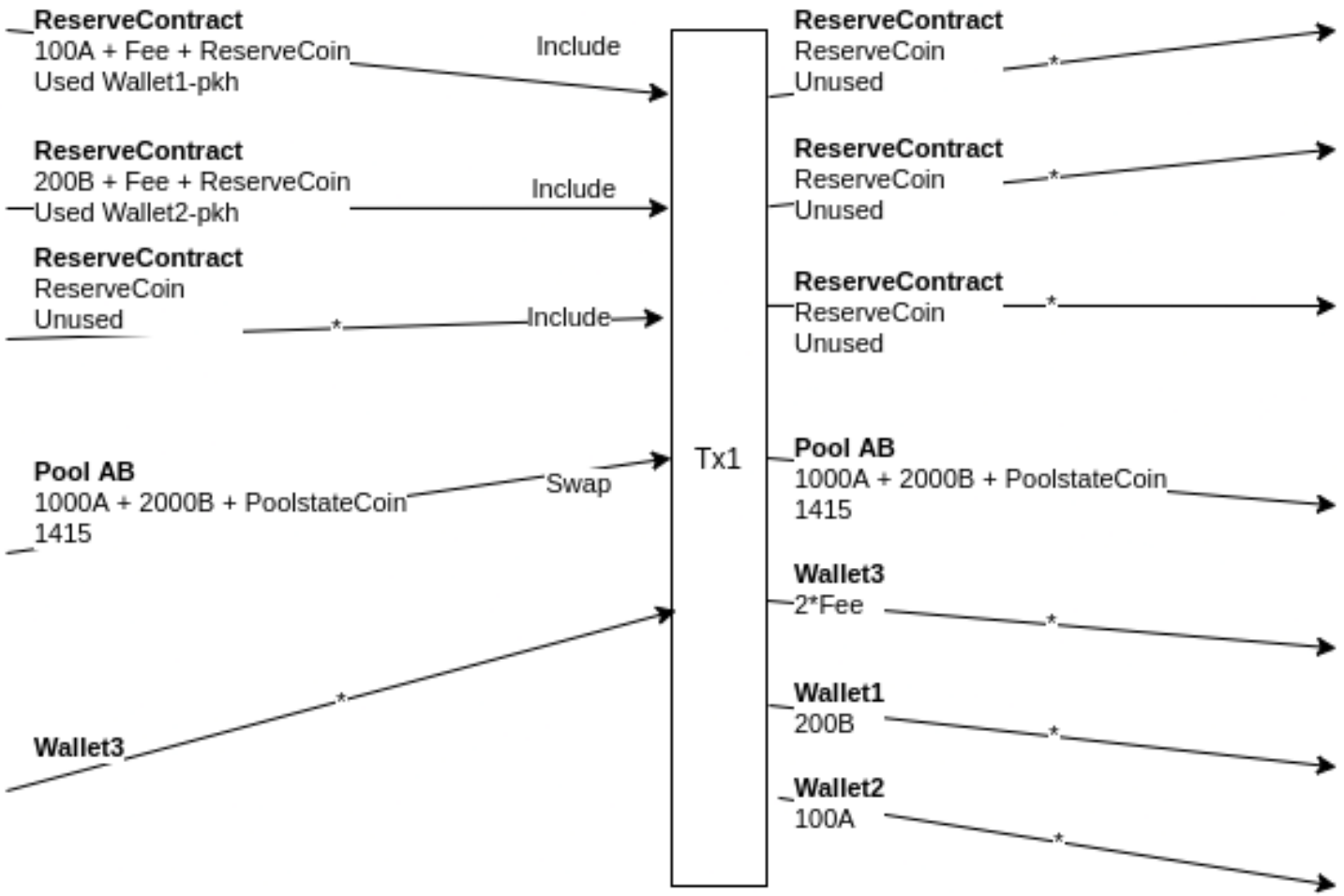
With the following major changes:

- Tx2: Create n UTxOs at a Reserve contract address
- Let users reserve and retrieve Reserve UtxOs
- Tx3: Include n Reserve UTxOs
- Tx6: Burn n Reserve UTxOs

Reserve and Retrieve



The Swap


ReserveContract
100A + Fee + ReserveCoin
Used Wallet1-pkh
Include
ReserveContract
200B + Fee + ReserveCoin
Used Wallet2-pkh
Include
ReserveContract
ReserveCoin
Unused
*
Include
Pool AB
1000A + 2000B + PoolstateCoin
1415
Swap
Wallet3
*
Tx1
ReserveContract
ReserveCoin
Unused
*
ReserveContract
ReserveCoin
Unused
*
ReserveContract
ReserveCoin
Unused
*
Pool AB
1000A + 2000B + PoolstateCoin
1415
Wallet3
2*Fee
*
Wallet1
200B
*
Wallet2
100A
*

Validation logic

These are the conditions under which a reserve UTxO can be consumed:

```
mkReserveValidator :: Coin U -> Coin ReserveState -> Coin PoolState -> ReserveDatum -> ReserveRedeemer -> ScriptContext -> Bool
mkReserveValidator u rs ps reserveDatum reserveRedeemer ctx =
    case reserveDatum of
        (Unused lp n) -> case reserveRedeemer of
            Reserve ->
                outputContainsFee          &&
                outputContainsReserveToken &&
                correctLiquidityPool       &&
                datumIsUnchanged
            Include ->
                poolStateCoinIncluded
            Destroy ->
                uniswapCoinIncluded
        (Used lp pkh n) -> case reserveRedeemer of
            Retrieve ->
                (Validation.txSignedBy info pkh) &&
                outputContainsReserveToken       &&
                outputStateUnused outputDatum    &&
                correctLiquidityPool             &&
                datumIsUnchanged
            Include ->
                poolStateCoinIncluded
```

surprising that ‘Include’ comes with only one condition.
The reason: the include validation is delegated to the Swap validator.

Because consuming Swap UTxO, must make sure that swap conditions are fufilled

Surprising that so many token-conditions:
Only checking that the UTxO is present is not enough, because everyone can create UtxOs at script address. Validation only happens when UTxOs are consumed

Swap validation

- Checks that all participants get the correct amount.
- From the Context I get the values paid to all the PubKeys in the transaction.
- These values must be equal to the amounts the clients expect, given their inputs.
- For example: If User1 supplies 10% of currencyB in the input, he must get 10% of the total output of currencyA which is calculated in two steps:

```
amountForSwapInput :: Integer -> (Amount A, Amount B) -> (Amount A, Amount B)
amountForSwapInput r (a,b)
    | a * r > b = (a - b / r, 0)
    | otherwise = (0, b - a * r)
```

Only the ‘excess’ part of the currency with the higher value is put into the swap

```
totalValA' :: Amount A
totalValA'
    -- if B was put into swap:
    | newB - oldB > 0 = (oldA - newA) + totalValA
    | otherwise       = totalValB `divide` ratio

totalValB' :: Amount B
totalValB'
    | newA - oldA > 0 =  (oldB - newB) + totalValB
    | otherwise       = totalValA * ratio
```

oldA, newA, oldB, newB are the values in the liquidity pool before and after the swap

This way, the Swap is deterministic, no arbitrary order of execution

OffChain code

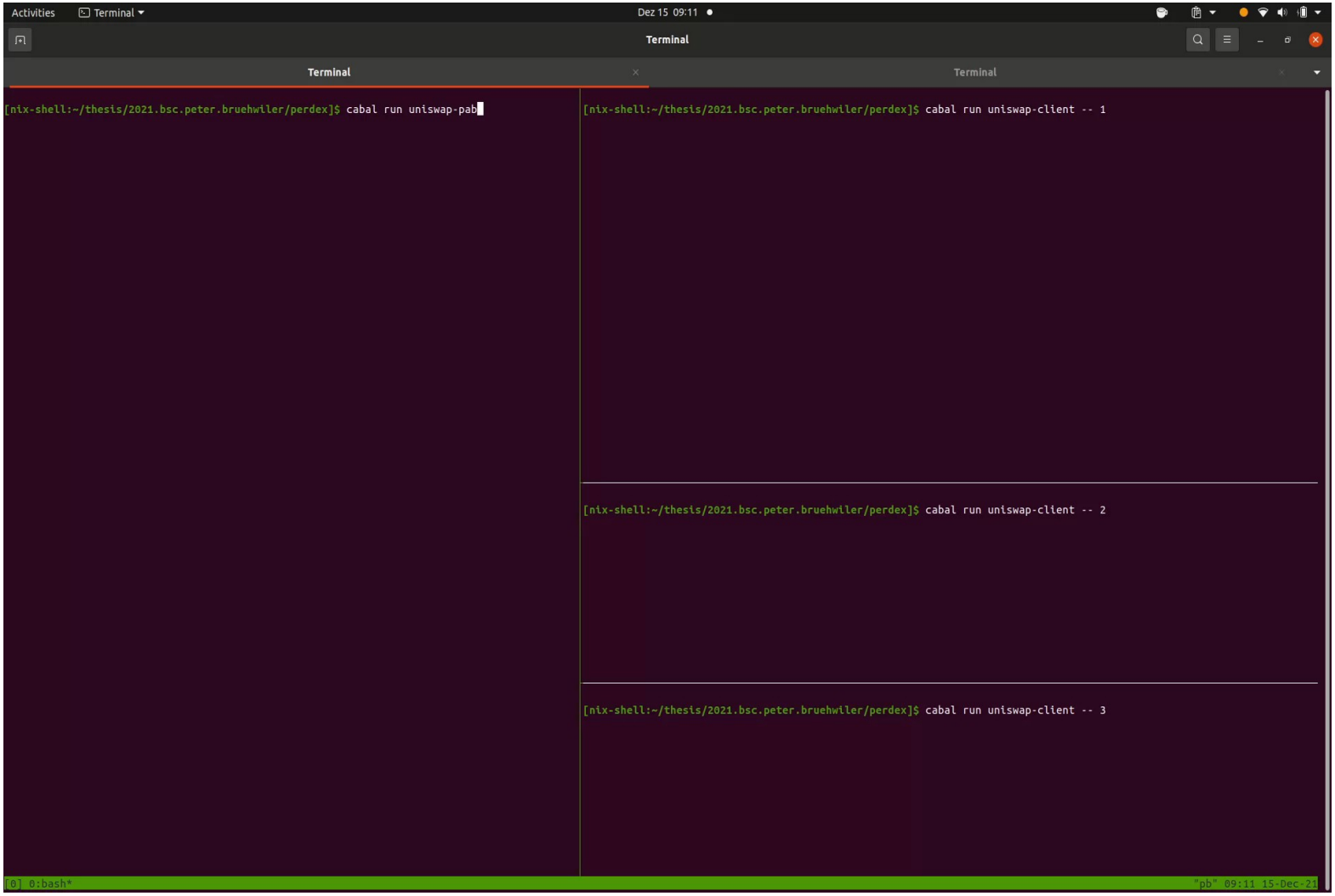
The OffChain code is not that different from the OnChain code:

- Outputs to PubKey addresses are created so that they satisfy the validation logic, given the values of all the Reservations
- ReserveCoins are minted/burned
- A new Swap UTxO with containing the new amounts of CurrencyA and CurrencyB is created

Building the swap transaction

```
tx      = Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toBuiltinData Swap) <>
          mconcat (scriptSpending $ map fst orefsAndOsUsed) <>
          mconcat (scriptSpending $ map fst orefsAndOsUnused) <>
          Constraints.mustPayToOtherScript (validatorHash usScript) (Datum $ PlutusTx.toBuiltinData (Pool lp liquidity)) val <>
          Constraints.mustPayToOtherScript (validatorHash $ reserveValidator us) (Datum $ PlutusTx.toBuiltinData (Unused lp)) (unitValue rsC) <>
          Constraints.mustPayToOtherScript (validatorHash $ reserveValidator us) (Datum $ PlutusTx.toBuiltinData (Unused lp)) (unitValue rsC) <>
          Constraints.mustPayToOtherScript (validatorHash $ reserveValidator us) (Datum $ PlutusTx.toBuiltinData (Unused lp)) (unitValue rsC) <>
          mconcat (pubKeyConstraints clientAmounts)
```

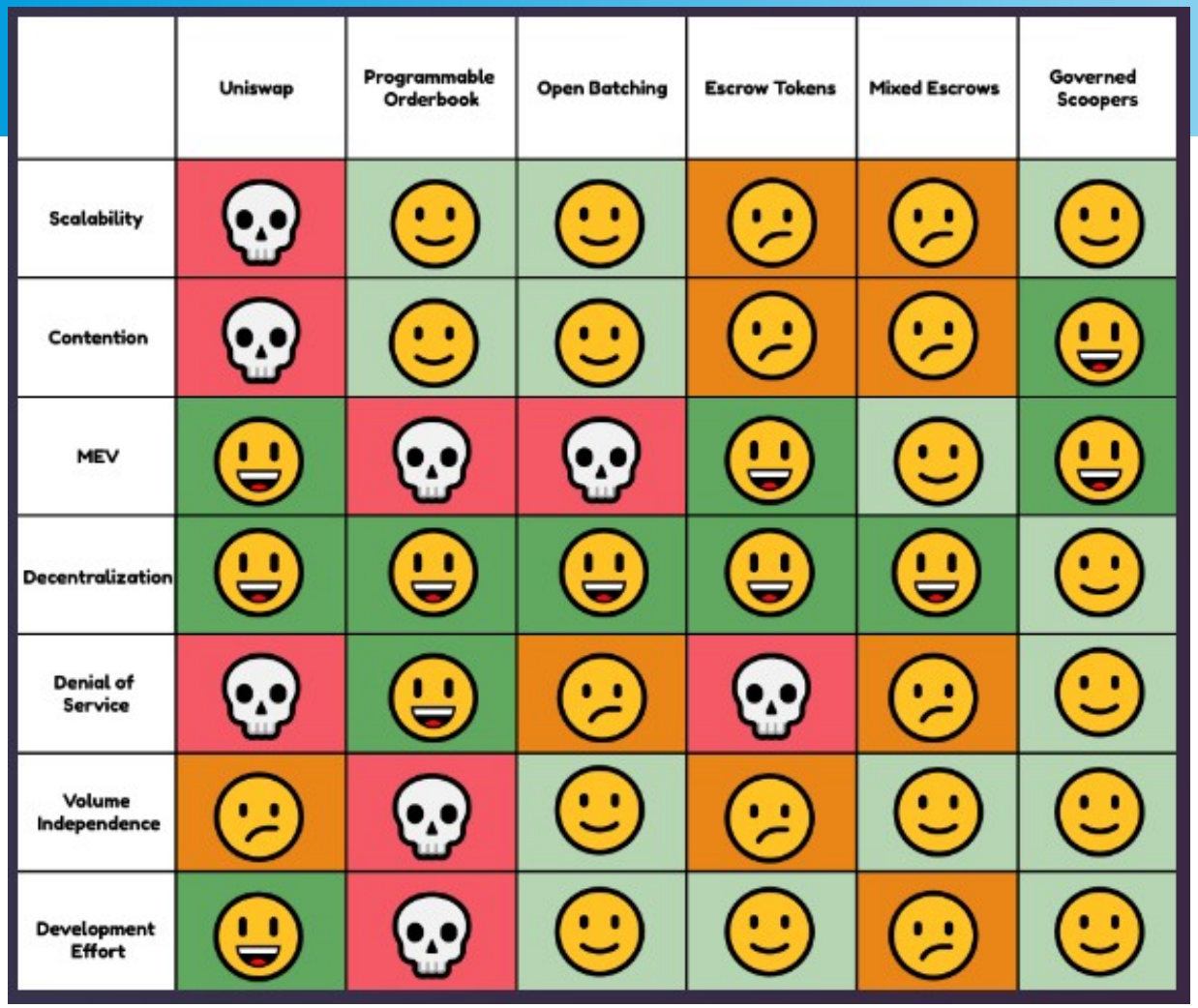
Demonstration



How decentralized is the DEX?

- Execution of the Swap happens OffChain
- Criticized as not truly decentralized
- In the nature of Cardano, a Validator can not jump into action on it’s own.
- But: The Validator can give guarantees, f.ex. that the order of Swap execution is deterministic. This could also be achieved with a timestamp

Sundaeswap



The ‘Escrow Tokens’ approach “crashes head first into the sizing limits of the Cardano blockchain.” getting minimal contention “depends on hundreds of escrow tokens, and in practice, our first implementation of this was hitting the Cardano protocol limits with 5 or 6 escrow tokens, before even adding in features like governance.”

Finally, they went for the ‘Governed Scoopers’ model:
“by aligning incentives and creating systems of self-governance, you can scale a system by building trust into the protocol.”

a single transaction is currently limited to a maximum of 16KB

Difficulties

- There are limitations because of the narrow context seen by a validator: f.ex. not possible to check if a certain PubKey has already created other UTxOs
- For the OnChain code, only a subset of Haskell is available, as it must be compiled to Plutus Core by the GHC plug-in called Plutus Tx.

Further developments

Use order tokens instead of order UtxOs,

because the reserve validator gets executed n times

Steps, described by IOG:

Order submission:

A specific MintingPolicy script can be used to mint an ‘order’ token submitted to the user's public key address.
The hash of the user's public key address, together with the order description and any necessary transaction fees, can be sent to the request script for order notification.

Order processing:

The batcher finds out about orders at the request script address.
The relevant users must sign the aggregated transaction for submission.

So why go through all the trouble?

- Tx (and fee) is deterministic: never fails if the UTxOs are available. If they aren't, transaction doesn't get submitted to the Blockchain.
- Every token is a native token: no need for smart contracts to move them around

```
Map CurrencySymbol (Map TokenName Integer)
```

- Layer 2 solutions easier to implement, because there is no worry about global state