Comenius University in Bratislava
Faculty of mathematics, physics and informatics

Optimistic Design Pattern in eUTxO Models

Master's thesis

2022
Bc. Michal Porubský

Comenius University in Bratislava
Faculty of mathematics, physics and informatics

Optimistic Design Pattern in eUTxO Models

Master's thesis

Study programme:	Computer science
Field of study:	Computer science
Training work place:	Department of computer science
Supervisor:	Ing. István Szentandrási, PhD.

Bratislava, 2022
Bc. Michal Porubský

14463702

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Michal Porubský
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Optimistic Design Pattern in eUTxO Models
Optimistický dizajnový vzor v eUTxO modeloch

Anotácia: Smart kontrakty ponúkajú spôsob ako dať garancie pri decentralizovanej exekúcii programov na blockchainoch. Rozšírený model nepoužitých výstupov transakcií (eUTxO model) rozširuje klasický model UTxO ktorý poznáme napríklad z kryptomien Bitcoin alebo Cardano o smart kontrakty. Tento model je veľmi nový a teda existuje len veľmi málo výskumu ohľadom dizajnových vzorov takýchto aplikácii. Táto práca by chcela na tomto stavať a zadefinovať optimistický dizajnový vzor, ktorý by mohol pomôcť s problémom limitovaného kontextu smart kontraktov, ktoré sa snažia nebyť obmedzené čo sa týka priepustnosti a zároveň potrebujú uchovávať centralizovaný stav. Toto by sme chceli aj demonštrovať na jednoduchej aplikácii.

Vedúci:	Ing. István Szentandrási, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	09.11.2021

Dátum schválenia: 09.11.2021 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

....

študent vedúci práce

14463702

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Michal Porubský
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Optimistic Design Pattern in eUTxO Models

Annotation: Smart contracts offer a way to enforce specific logic in decentralized computations on blockchains. Extended unspent transaction output (eUTxO) model introduces smart contract functionality to the UTxO model known from Bitcoin or Cardano. Due to its novelty, there exists only little research on design patterns of such contracts. We aim to fill in the gaps here and introduce an optimistic design pattern which could be a possible solution to circumvent the limited scope of smart contracts in the eUTxO model in the quest of fighting the concurrency issue decentralized applications face when a centralized state is needed and demonstrate it on a sample application.

Supervisor:	Ing. István Szentandrási, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	09.11.2021
Approved:	09.11.2021

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

Student

Supervisor

Thank you notes: I would like to thank everyone that was there for me when I needed it and helped me along the way.

Abstrakt

Smart kontrakty ponúkajú spôsob ako dať garancie pri decentralizovanej exekúcii programov na blockchainoch. Rozšírený model nepoužitých výstupov transakcií (eUTxO model) rozširuje klasický model UTxO ktorý poznáme napríklad z kryptomien Bitcoin alebo Cardano o smart kontrakty. Tento model je veľmi nový a teda existuje len veľmi málo výskumu ohľadom dizajnových vzorov takýchto aplikácii. Táto práca na tomto stavia. Zadefinovala optimistický dizajnový vzor, ktorý je jedným z riešení ako pomôcť s problémom limitovaného kontextu smart kontraktov, ktoré sa snažia nebyť obmedzené čo sa týka priepustnosti a zároveň potrebujú uchovávať centralizovaný stav. Toto zároveň demonštrujeme na jednoduchej decentralizovanej aplikácii.

Kľúčové slová: Smart kontrakty, DeFi, Cardano, eUTxO, dizajnový vzor, problém konkurencie

Abstract

Smart contracts offer a way to enforce specific logic in decentralized computations on blockchains. Extended unspent transaction output (eUTxO) model introduces smart contract functionality to the UTxO model known from Bitcoin or Cardano. Due to its novelty, there exists only little research on design patterns of such contracts. In this text we help to fill in the gaps and introduce an optimistic design pattern which is a solution to circumvent the limited scope of smart contracts in the eUTxO model in the quest of fighting the concurrency issue decentralized applications face when the state centralization is needed. We also demonstrate this approach on a sample decentralized application.

Keywords: Smart contracts, DeFi, Cardano, eUTxO, design pattern, concurrency issue

Contents

List of Figures

Introduction

The aim of this thesis is to study the extended unspent transaction output (eUTxO) model that builds on top of the classical unspent transaction model used for years in some of the well known cryptocurrencies such as Bitcoin, Cardano, Ergo, etc. The eUTxO model is a novel introduction to Cardano that brought smart contract support to the model in September, 2021 [3].

The model offers better scoped deterministic environment well suitable for security analysis [2] [23]. Compared to Ethereum, the blockchain most commonly used for smart contract development, this is a big improvement. That is why we consider it worth the effort to study, shed light on some of the most common obstacles decentralized application developers in the eUTxO model face and contribute to the overall security of the way decentralized applications are written in the end.

Being in the early phase of the model adoption, the obstacles are just beginning to be identified and solutions offerred. This study takes one of such obstacles, namely the **concurrency issue** we discuss in Chapter 2, it describes potential solutions to it and takes an in-depth look at the agent solution that we as well as the community consider superior [5] [19]. We then go on and define a design pattern aiming to achieve full decentralization of the priviledged agent role that is inherently introduced by the solution implementation.

There is an increasing number of projects impacted by the concurrency issue. They tend to come over and over with their own, slightly modified, but in general pretty similar solutions to it. Most of those can be considered to be agent solutions we take a deeper look at in section 2.4. More often than not, however, those solutions result in the otherwise decentralized applications getting centralized in the control of a handful of people known as agents who are responsible for the smooth application operation. Centralizing the service operation is not desired as that contradicts the whole reason why decentralized applications emerged.

Despite the full decentralization being desired, it is a hard problem to fully achieve it, resulting in protocols avoiding solving it for now. It is not an easy problem to solve, but we hope that this text and the design pattern we define helps with this and projects will aim to the full decentralization.

In the first chapter 1, we create a baseline for the whole thesis. We start by discussing blockchain in general. Quickly, we move on to describe the eUTxO model we are mostly interested in and cover all of its relevant specifics in great detail. Most importantly, we explain the role of smart contracts in the model.

The second chapter 2 takes a look at a particular common obstacle blockchain developers face in the model which is the already mentioned concurrency issue. We discuss how and why the problem arises and discuss potential solutions to it. We conclude the chapter by explaining the agent-request solution along with the priviledged agent role that we will be taking a better look at later in the text.

The third chapter 3 is the chapter where we explain the design pattern we suggest that leads to the optimistic decentralization of the agent role into anyone that meets the initial requirements. We mention how we can limit the exploitable surface by a malicious agent, but that in the end we simply trust agents to be honest and build a mechanism that is able to punish dishonest agents. There can be two major approaches how that mechanism could work and we discuss both of them.

The final chapter 4 is a demonstration of the design pattern on a demo problem. The chapter is accompanied by an actual runnable implementation we wrote that is located on the USB coming with this text. We discuss various details that all together form the more complex approach to the agent punishing mechanism described in the previous chapter 3.3.1.

Chapter 1

eUTxO model

In this chapter we give a quick introduction into blockchain, describe its purpose and some of the technical specifics. We go on by explaining the unspent transaction output (UTxO) model some blockchain protocols make use of. Finally, we put a special emphasis on the extended version of the UTxO model (eUTxO) which represents the baseline model for this thesis.

1.1 Blockchain

Blockchain can be viewed as a distributed ledger. It consists of blocks of transactions strongly cryptographically linked to each other. Every block references the previous block's cryptographic hash. This makes it impossible to alter any part of this linked chain of blocks without modifying all the subsequent blocks as well [21] [6]. A transaction is the tiniest component that modifies the ledger. A block is a batch of transactions.

Blocks are validated and accepted into the chain by nodes. Nodes are connected components of a peer-to-peer network running a consensus protocol to agree on the current chain state. The consensus protocol is usually highly Byzantine fault tolerant [21] [27].

A digital token is a possession that is tracked in the distributed ledger. A cryptocurrency also referred to as a digital currency is a special kind of a token which usually has real-life fiat value determined by the supply and demand on exchanges. There is an increasing number of web shops that accept cryptocurrencies as another form of payment for the goods. There are even countries that accept Bitcoin as legal tender [8]. Bitcoin is the first cryptocurrency there was [1] and the biggest in terms of market cap [16]. Therefore it is considered to be the safest and least volatile to accept as a payment or to invest in. It is often referred to as the digital gold as investors tend to believe it can preserve value long-term. A token, however, does not necessarily need to have an inherent value and may not even be tradable.

A special type of tokens are non fungible tokens (NFTs). Every such token is uniquely identifiable. In other words, there can be exactly one in quantity. They are commonly used to keep track of an owner of a particular real-life commodity, a painting, an audio track and others [29]. However, a proper legal basis is lacking at the time of writing this thesis. We discuss tokens in Cardano in more depth in subsection 1.3.2.

A place which can hold tokens is often referred to as an address. An address is derived from a cryptographic public key from a strong assymetric cryptography scheme. It is said that tokens are at an address if they are noted to belong to an entity holding the private key corresponding to the public key of the address in the distributed ledger.

Once tokens reside at an address, only the holder of the private key can send them to another address by creating and cryptographically signing a transaction which describes the token movement. The private key corresponding to the previous address is used for the signature. After the transaction submission, nodes verify that the relevant signature is present. They are able to do this as the public key is part of the address and thus a publicly known information. This is part of the standard transaction validity checks.

1.2 Unspent transaction output model

Unspent transaction output model, often referred to as the UTxO model, is a model of how to store and how to determine the current unused balances of addresses. It is used in Bitcoin, Cardano, Ergo and many other blockchain protocols with slight alternations. In this text, we will focus on Cardano specifically, as that would be the chain we will use to demonstrate the Optimistic design pattern in Chapter 4.

The building block of this model is a **transaction** which consists of inputs and outputs. Every transaction output contains a bundle of tokens, resides at a single address and is either spent or unspent. Whenever a transaction output is used in another transaction as an input, that transaction is accepted into the blockchain by nodes running the consensus protocol, that transaction output is considered spent and can not be used as an input in any other transaction. This is enforced as another standard validation performed by the nodes [2]. The current distribution is **the set of all unspent transaction outputs** which the nodes keep track of. This fact gave the model its name.

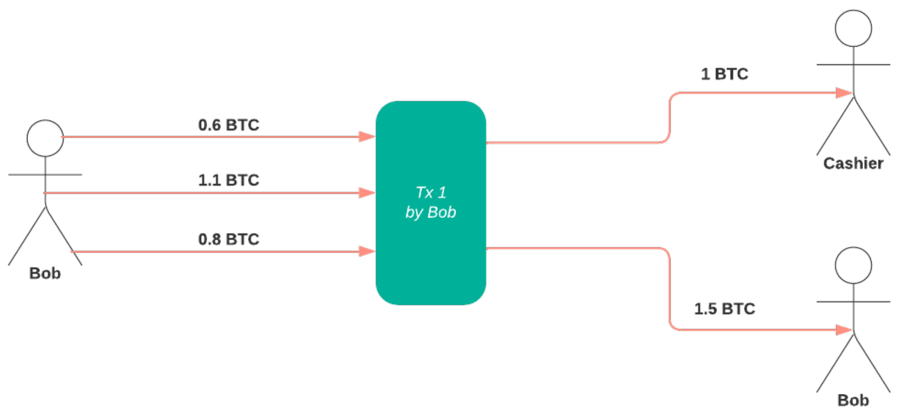


Figure 1.1: An example transaction in UTxO model. Cashier would get 1 Bitcoin and the change would stay at the Bob's address.

As an analogy, it is possible to think of it in a similar manner as a buyer who pays in a shop with his coins and banknotes. His wallet is an analogy to an address on the blockchain. There are multiple banknotes and coins present in the wallet. Those represent unspent transaction outputs. Tokens can refer to the value those coins and banknotes represent. When the buyer pays in a shop, he makes a transaction. The inputs of the transaction are coins he uses to pay. The outputs are of two types. The first type represents coins (UTxOs) that are left in the cash register. Those will reside at the beneficiary address upon transaction inclusion into the blockchain. The other type of outputs are those representing the remainder coins that are returned by the cashier to the buyer. After the transaction is finished, the buyer's wallet contains different coins. The current token distribution changed. A similar transaction can be seen in Figure 1.1.

The important difference, however, is that every UTxO can be spent at most once. This does not have an analogy in the real world as coins can keep changing their holder while remaining the same. An identifier of a UTxO is a tuple of the identifier of the transaction where it was created and its index among that transaction outputs. Naturally, whenever a UTxO is spent in a transaction, only new UTxOs with new identifiers referencing that transaction are created. This forms an immutable chain of all state changes.

1.3 Extending the UTxO model

An extended version of the previously described unspent transaction output model is a model introduced to Cardano by a hard fork in September 2021 [3]. It adds smart contract support into the UTxO model which was the standard before. In the following text, we will cover validator scripts and minting policies. Reward and certification scripts are out of scope of this thesis as they are designed to manage staking rewards distribution only and our text does not aim to touch these topics.

Inspired by [17], we can summarize the updates on top of the UTxO model as follows:

- **Address concept generalized**. So far, we explained an address to be a place where tokens reside. We expanded on it and said that it is derived from a public key of the entity owning those tokens. The derivation often involves computing a strong cryptographic hash and including the hash in the address. Doing that, we can see the address as a lock which defines how the tokens can be used. By having the address reference a public key, UTxOs residing at the address can be unlocked and used only when a valid signature corresponding with that public key is included in a transaction that spends those UTxOs.

  The extended model builds on this analogy and introduces another entity. The entity is a smart contract often interchangeably referenced to as a validator script. An address in eUTxO model can either reference a cryptographic public key hash which would unlock UTxOs by a valid signature or it can reference to a hash of a compiled smart contract validator script. In the latter case, the validator script itself would need to be part of the transaction, it would be run by nodes of the decentralized protocol and it would need to validate the transaction spending UTxOs at that address. If the validator script would not validate the transaction, it could not be accepted into the blockchain [2].

  Based on whether an address refers to a public key hash or a validator script, we say the address is a payment address or a script address.

- **State of UTxOs**. This enables unspent transaction outputs to carry arbitrary additional data. This is powerful in combination with smart contracts as validator scripts can maintain a state and decide on whether to validate a transaction or not taking the state into account as well.

1.3.1 Smart contracts

The high level role of smart contracts is to enable parties in an agreement to a trustless execution of the agreement without any intermediary while putting the trust into the code of the smart contract instead. The smart contract is often open sourced and audited with a very special emphasis on security to ensure it works exactly how it was agreed upon and does not contain bugs.

As mentioned in previous sections, smart contracts have the form of validator scripts in Cardano. By creating a transaction and sending tokens to a script address, the transaction initiator locks the tokens at that script. Any following transaction manipulating with those tokens must follow the logic of the validator script corresponding to the address. Any other transaction is disallowed.

An interesting note is that **anyone** can manipulate with UTxOs at a script address if the validator script does not restrict this. This enables the creation of decentralized applications serving a potentially very big community of people that want to use and benefit from the same application even utilizing shared resources. The applications can manage anything from a decentralized exchange exchanging native tokens, operating a fair auction up to providing loans.

A validator script, however, can not initiate any fund movement on its own. We remind that the only way to move funds is to create a transaction. Smart contracts in Cardano can not create and submit transactions. Smart contracts are only validator scripts which decide whether to validate or dismiss transactions.

As of the time of writing this thesis, validator scripts in Cardano were meant to only be written in **Plutus**. Plutus is a language built specifically to write smart contracts. It is a language built on top of Haskell to make use of the purely functional programming approach and a strict type system [12]. We can say that a validator script is a pure Haskell function with the signature of:

$$validate :: Datum \to Redeemer \to ScriptContext \to Bool$$

Let's describe the respective elements to fully grasp what is and what is not possible to decide with validators scripts.

Script context

Script context stands for the information about the pending transaction and the currently running script. A pending transaction is the transaction that is being validated at the moment the validator script is run. This is the only transaction the currently running script sees. In contrast to Ethereum and its smart contracts [11], this is a significant difference, whereas **the validator has a very limited scope to the currently pending transaction**. It does not see any other transactions and UTxOs. Even though this changes and complicates the whole design of decentralized applications that want to be implemented in Cardano, it brings more predictable scope of the contracts and that enables better security analysis, even enabling formal verification.

As can be seen in Figure 1.2, the validator script receives almost the whole pending transaction and its data. The most notable elements are:

- **Transaction inputs and outputs**. Input UTxOs and output UTxOs. Every UTxO has an address where it resides, a value which is a bundle of tokens it holds and optionally a datum representing the state.

- **Minted value**. There can be tokens that are minted in the pending transaction. Minting tokens means creating new such tokens. On the other hand, burning tokens is their destruction. Burning of tokens can be expressed in the minted value by putting negative numbers there.

  Similar to validator scripts, there are minting policies in Plutus which define how various tokens can be created (minted) and how they can be burned. Refer to subsection 1.3.2 for more information.

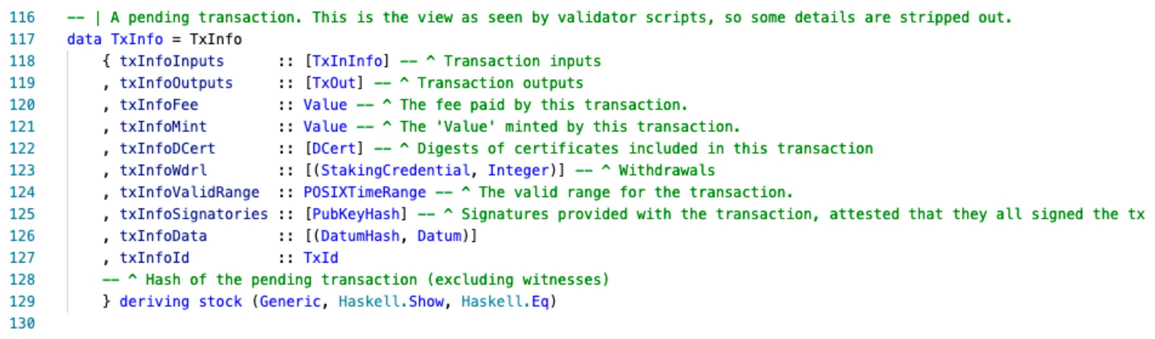
- **Signatories**. A validator script can see all the parties who signed the transaction. This is often used as a mechanism of saying that all the parties that signed the transaction agree with what the transaction defines. Thanks to this, it is for example easy to build a contract that validates whenever at least three out of four parties agree with a transaction, etc.

- **Transaction validity range**. A transaction validity range is a time interval when the transaction can be accepted into the blockchain. Nodes check this as a part of the standard transaction acceptance checks [2]. The transaction validity range can be used to lower bound or upper bound the current time. If a short enough difference between the start and end of the validity range is enforced by the validator scripts, even the current time approximation is possible.

Datum

Datum stands for the arbitrary data that can be part of script UTxOs in the extended UTxO model. It can be any JSON-like data structure. It is used to store the state of smart contracts on the chain.

The state is set by the person creating the UTxO and can not be changed. Any person spending the UTxO needs to ensure that the validation script would validate with the current state that is part of the UTxO. If the validation passes, however, there can be a new UTxO created on the same script address with a different state. It is often the case that the validator script wants to limit the possible next states and



```haskell
-- | A pending transaction. This is the view as seen by validator scripts, so some details are stripped out.
data TxInfo = TxInfo
    { txInfoInputs      :: [TxInInfo] -- ^ Transaction inputs
    , txInfoOutputs     :: [TxOut] -- ^ Transaction outputs
    , txInfoFee         :: Value -- ^ The fee paid by this transaction.
    , txInfoMint        :: Value -- ^ The 'Value' minted by this transaction.
    , txInfoDCert       :: [DCert] -- ^ Digests of certificates included in this transaction
    , txInfoWdrl        :: [(StakingCredential, Integer)] -- ^ Withdrawals
    , txInfoValidRange  :: POSIXTimeRange -- ^ The valid range for the transaction.
    , txInfoSignatories :: [PubKeyHash] -- ^ Signatures provided with the transaction, attested that they all signed the tx
    , txInfoData        :: [(DatumHash, Datum)]
    , txInfoId          :: TxId
    -- ^ Hash of the pending transaction (excluding witnesses)
    } deriving stock (Generic, Haskell.Show, Haskell.Eq)
```

Figure 1.2: Pending transaction info type taken directly from the source code of Plutus V1 library.

validate only if the state transition is allowed.

Note that there can be multiple UTxOs on the script address. However, they do not have to have the same state. This is crucial to the model and it enables a variety of interesting applications. Refer to the vesting example in 1.3.1 for an example on how this can be used.

In practice, datum size is very limited. As seen in the Figure 1.2, the datum needs to be included in any transaction that spends the UTxO with the datum. The size of any transaction is limited to $16kB$ [7] as of the time of writing this thesis which inherently limits the size of datum itself. This fact dissolves all ideas around storing too much information in the script state unless distributed data structures are used.

If it were to happen that someone would create a UTxO with too big datum, nobody would be able to spend it as such transaction would not be accepted by nodes due to its size. This would lock any value sent into the UTxO until the Cardano team updates the protocol parameters. It can happen but it certainly is not something acceptable to depend on while designing decentralized applications.

Redeemer

Redeemer is arbitrary data sent to the validator script by the person initiating the transaction spending an UTxO on a script address. While a datum can be viewed as the state of the script, a redeemer is analogous to an action that the initiator intends to perform.

Vesting example

Let us give a simple example of an application. Imagine that we wanted to give out money as a gift to our children. However, we want them to be able to access no sooner than when they reach eighteen years of age as we don't want them to spend that until that.

We will create a vesting script for this and lock the funds there. More specifically, if we had five children, we would create a transaction creating five transaction outputs on the vesting script address each having its own state. The datum of a particular vesting UTxO would define the person that is to be gifted and the time that he or she becomes an adult. Note that since we create five vesting UTxOs to give away to five different children, we can specify five different datums with the respective child's public key hash and the date.

There would be only one redeemer possible denoting the only action that can be made. The only action is for the child to claim his gift given he or she already reached eighteen years.

The vesting validator would consist of two checks. The first check would verify that it is indeed the child who this particular vesting UTxO was created for claiming the gift. This would be determined by checking that the transaction is signed by a private key corresponding to the public key saved inside the datum. The public key in the datum was put there in the faith that it belongs to the child. Hence we rely on its private key to be in the sole possession of him/her as well.

The second check would pass if and only if the date of the person becoming an adult has already passed. This can be checked by seeing the transaction validity range of the claiming transaction and comparing it to the timestamp written in the datum. The timestamp in the datum was put there by us, meaning we can trust this timestamp. If the transaction validity range start is past the timestamp saved in the datum, that means that either the transaction will be discarded by the nodes if the current time is not past that time or it means that the child has become an adult. If those two checks validate, the child is free to take his/her gift anywhere they want.

1.3.2 Minting policies

Minting policies are a special type of smart contracts with a bit of a different purpose. Instead of governing how UTxOs at a script address can be spent, they govern the minting and burning of tokens. They do not and can not restrict the transmission of tokens that are already minted. Minted tokens are considered part of UTxOs and only a validator script can control spending of its UTxOs. For additional clarity, we will not refer to minting policies as validators in this text.

Minting policies are also written in Plutus and have a similar function type signature to validators:

$$\mathit{policy} :: \mathit{ScriptContext} \rightarrow \mathit{Redeemer} \rightarrow \mathit{Bool}$$

The only difference in the signature is that the datum is missing. All other types stay the same. This means that minting policies are unable to have a state. This is natural as minting policies do not control the spending of UTxOs and only UTxOs can contain datums. Policies need to allow or forbid minting or burning of tokens based on the pending transaction info and additional data (redeemer) sent by the transaction initiator only.

In order to be able to consider the usage of minting policies in decentralized application designs, let us explain more about tokens, their properties, differences, importance and usage.

Tokens

Previously, we described a token as any possession whose owner is tracked in the blockchain ledger. We mentioned that some tokens have monetary value, some do not and that there is a special group of tokens (NFTs mentioned in 1.1) that exist in uniqueness and why that could be important. In this section, we try to discuss the importance of tokens in more technical detail based on [15].

In Cardano since the multi-asset support [24], a token is a tuple of:

- **Policy id**. Policy id is the hash of the compiled minting policy which governs the minting and burning of this token. Similar to validator scripts where the relevant hash is a part of the script address, the hash of the policy is part of the token itself to very clearly identify it.

  This is very important as it essentially means that in order to create a new token in a transaction, nodes can check that the correct minting policy is included as a part of the transaction, they can run the policy and check that the policy allows minting that token.

- **Token name**. Token name is an arbitrary bytestring. It often serves as a human redable identifier, but is not an identifier in itself. The most important part of a token is its policy id. There can exist multiple very different tokens with the same token name. This is a common scam in which a person that mints a valueless token with a token name of something valuable tries to sell it for profit.

  In addition to the human readable identifier role, a token name can also be used to differentiate between different kind of tokens that are all governed by the same minting policy.

There is a special token circulating over Cardano blockchain. It is the native cryptocurrency of Cardano called **ADA**. Its policy id is an empty bytestring [24]. As we discussed, the policy id is the hash of the minting policy. Naturally, no minting policy hashes to an empty bytestring. That means that ADA can not be minted nor burned in any transaction. It is fixed in the total supply of 45 billions that was there right from the beginning.

It is a currency, can be traded on various exchanges and is accepted as a form of payment. As such, whenever we talk about ADA, we talk about a token with monetary value. We do not obsess over the price itself and its non-stability over time as it is not essential to this text.

All other tokens need to have a minting policy which is the only piece of code that can limit how and when a particular token is minted.

For example, minting an NFT is possible to achieve by a special minting policy that allows the minting of exactly 1 token if and only if a particular constant UTxO is spent in the pending transaction. Any UTxO ever can be spent at most once. This fact limits the overall possible supply to at most 1 such token.

In relation to decentralized application designs, let us mention a few use cases of how tokens can be used when designing an application:

- **Permission tokens**. There can be sensitive operations in the protocol. As such, it is not always desirable for everyone to be able to execute them. A way to limit this can be restricting the set of users able to perform the operation. There are two ways to easily authenticate users able to execute the action. The first one is to whitelist a specific set of public key hashes whose signature is required to be present in the transaction for the validator script to authorize the action. This is the simplest option. The disadvantage is that this is not scalable to much bigger number of whitelisted users. That is because the validator script needs to know the whole list of whitelisted users to determine whether to allow or dismiss the pending transaction. As we mentioned already, the size of both the datum and the script is limited. Another complication is that it is not easy to change the whitelisted list as changing that would mean implementing the logic into the validator script itself. This is often undesirable, because complicating the scripts open doors to more vulnerabilities.

  The other option is to authenticate users based on an ownership of a token. This is a more flexible and granular approach. It is possible to hardcode a token identifier into a validator script and the script is able to check that the token is present in the transaction attempting to execute a priviledged role. The script itself does not need to bother about who is able to own it. It is assumed that the token would not be transferred to a non-whitelisted party. This assumption can actually even be enforced by wrapping the token inside a script UTxO, as can be seen later in chapter 4.

- **Value tokens**. Tokens can be minted and awarded to users participating in a protocol in return for their action. Those tokens can enable them to later perform an action only they are entitled to.

  Let us show this on an example. Looking at WingRiders automated market maker (AMM) decentralized exchange [19], users providing liquidity to the protocol provide valuable tokens and get liquidity provider tokens from the protocol in return. Liquidity provider tokens are tokens minted by the protocol. They enable users to retrieve their valuable tokens (liquidity) back anytime. This is also an interesting example of how newly minted tokens can have an inherent value. That is, because they enable users to do an action that gets different tokens that already have monetary value. Not all tokens need to have monetary value, though.

- **Validity tokens**. Minting policies see the whole pending transaction context. In combination with the fact that validator scripts are called only when a UTxO is being spent from the script address and not called when they are created, it suggests a possible way to combine the two to ensure a proper creation of a UTxO on a script address.

  Imagine that only script UTxOs with a specific **validity token** are considered valid for the purposes of the protocol. The minting policy would allow minting this token only into a specific script UTxO. It could additionally check that some other conditions hold, for example checking a valid initial state construction. If the script validates that the validity token can never leave the script UTxO, we just constructed a simple protocol where we are able to ensure correct script UTxO initialization.

This idea can be generalized. It is not possible to restrict users from creating script UTxOs on any script address with any datum. It is possible, however, to limit the definition of valid UTxOs to those holding a specific token and limit that token lifecycle. As a result, the protocol can protect itself from the mentioned potentially malicious behavior.

Chapter 2

Concurrency issue

In this chapter we look at various reasons why decentralized applications often tend to want or need a centralized state in the eUTxO model. We continue on by explaining a problem such applications inherently face which is the concurrency issue and describe potential solutions. Then we pick a solution employing a special Agent role, go into more detail and explain why we think it is more superior compared to other solutions. We conclude the chapter by listing problems that may originate by introducing this role in various applications.

2.1 Centralized state

In the eUTxO context, state and datum are often used interchangeably. A centralized state then refers to a single UTxO that holds the whole application state in its datum.

It is common for applications to need to have the whole context in order to deterministically decide for the next application state or to agree on the final state. The need can arise naturally based on the specifics of the problem itself as can be seen in the example of an auction application in 2.1.1 or it can be a business requirement and have business implications as can be seen in the decentralized exchange example described in 2.1.2.

2.1.1 Auction example

An auction is a trustless application where sellers can offer their precious NFTs to highest bidders. The benefits of recruiting smart contracts is that the code can ensure that the seller sells to the highest bidder while making sure that the highest bidder actually receives the NFT he pays for. We will return back and build on this example in our final chapter 4.

A straightforward way to fulfill the requirements would be for the buyer to lock the NFT into an auction script and for any temporarily highest bidder to lock his offerred money into the script as well. That way, the auction validator script could make sure that if the bidder wins the auction, the offered money is transferred to the seller and the NFT goes to the bidder. It is also possible to make sure that the money is returned to the bidder in case he is outbid by somebody else.

Note, that the explained solution strictly depends on the fact that the state of the particular auction is centralized. Since there is only one NFT that is being sold, it is not possible to parallelize this into multiple UTxOs easily. We could not guarantee to multiple bidders that they will receive the NFT if their bid is the highest on the contract level unless they update the state of the currently highest bid and bidder in the correct place which is arguably the UTxO holding the NFT they bid for.

It is possible to parallelize this by making use of another kind of a consensus mechanism or recruiting another party, though. We will discuss these later in this chapter. The point of this example is to give an example of how making the state centralized helps with the design and is preferable for many applications.

2.1.2 Decentralized exchange example

An AMM decentralized exchange [20] is a trustless service that provides the exchanging of tokens without the unnecessary coordination with the other party that is providing the wanted tokens. Simplified, the idea lies in that there are places called liquidity pools with adequate quantity of both tokens. Liquidity providers provide both tokens into the pool in exchange for part of the fees from every swap that happens. The ratio of tokens inside the liquidity pool determines the exchange rate. A user that wants to swap tokens for other tokens interacts with this liquidity pool, the exchange rate is calculated for him and receives a fair amount of wanted tokens back. A fair exchange rate is assumed over the long term, as the contrary would open profitable money-earning opportunity for anyone to even the exchange rate out with the other exchanges.

This above relatively simple idea achieved great success. The first and most popular decentralized exchange that implemented it on Ethereum chain called **Uniswap** surpassed 100 billion trade volume in 2021 [18].

Let us repeat that the exchange rate is calculated automatically based on the ratio of tokens inside of liquidity pools. A swap puts one kind of tokens inside the pool and takes the other tokens back, inherently changing the exchange rate. This means that the more tokens inside, the more stable the exchange rate is going to be over swaps happenning. If we see the state as the amount of tokens inside the pool along with the tokens themselves, we can observe how centralizing the state (liquidity) into one place is actually beneficial for the end customers, making it desirable for both the protocol and the users.

2.2 The problem

We have now briefly explained what centralized state means and why it may be desirable on examples. The essential problem that inevitably comes with centralizing the state while wanting the application to be available to a number of users is called the **concurrency issue**.

The concurrency issue is the inability to satisfy multiple customers in parallel given a centralized state is used. In its essence, this is caused by the fact that the state is a datum in a UTxO and at most one accepted transaction can ever spend that UTxO while producing a new one with the updated state. Multiple customers submitting transactions that interact with the same protocol thus compete for the inclusion into the block by nodes. That's because their transactions are mutually exclusive [10] [5] [19] [4].

The implications of the problem are twofold. Firstly, it limits the throughput of the application into one user interaction per block creation. A new block in Cardano is created by nodes around every twenty seconds as of the time of writing this thesis. This is a very serious limitation and for many applications an unimaginable obstruction.

Secondly, even if we were to accept the throughput limitation, it leads to a terrible user experience. A user signs an entire transaction that contains the identifiers of specific UTxOs it tries to spend. In the case of multiple users competing for their transactions' acceptance, it is not possible to accept one transaction, hold the other transaction and accept it in another block. That's because the UTxO reference of the new state would have changed, making the held transaction invalid as it attempts to spend an already spent UTxO. The user whose transaction did not make it into the block is required to sign a new transaction with updated references again and compete in the next block.

Finally, let us mention that the concurrency issue is not present in this form in chains not using the eUTxO model and thus is a relatively new problem in the ecosystem. This is one of the reasons why it is not easy to port decentralized finance applications from Ethereum's account based model into the eUTxO model on Cardano even though those applications are open sourced.

2.3 Possible solutions

In this subsection, we discuss some of the possible solutions to the concurrency issue and the preferred solution. As picking the optimal solution is not the purpose of this thesis, we will only simply describe the options. We want to give credit to Sundaeswap for covering these in bigger detail in their thorough blogpost [5].

2.3.1 Naïve

The naïve solution is no real solution to the problem. It accepts the problem and its consequences, still employs a centralized state in a single UTxO and allows direct interactions with it by users.

We mention this as a solution simply because it may be enough for an application to not solve the issue at all. If the expected number of concurrent interactions is too small, it may be sufficient for the application to provide a nice retry option to its users.

2.3.2 Constant factor improvement

The likelihood of the concurrency issue occuring can be improved by a constant factor by splitting the state into a constant number of partial states. In the auction example, this would mean keeping track of k highest bidders in k separate UTxOs for some constant k and a following consensus transaction that would enforce the spending of all k partial states, computing the winner and concluding the auction.

Bidders could interact with any of the k auction UTxOs to express their bid which can reduce the number of bidders competing for the same UTxO supposing the bidders choose the auction UTxO randomly. k needs to be a constant number so that the contract can enforce that all k auction UTxOs are present in the final consensus transaction. All auction UTxOs need to be present to make sure the highest bid wins the auction.

2.3.3 Request batching

Request batching is a strategy with many different variations. The common idea, however, is to split the whole process of interacting with the centralized state into 2 steps:

1. **Request creation.** This step consists of a non-blocking transaction done by users whereas they express their binding intention to perform an action with the

centralized state by locking any tokens with data (datum) describing the intent into a newly created **request script** UTxO.

2. **Request execution.** In this second step, possibly multiple requests are batched and executed in a single transaction against the UTxO holding the protocol centralized state.

Thanks to the separation of the logic into steps, we are able to avoid users retrying their transactions as there is nothing blocking users to concurrently create script UTxOs. Furthermore, note that the user signature is required to create a request, but it may be another person that executes the request. That is the reason why users lock any required tokens necessary to execute the action into the request script UTxOs directly. Another benefit is that the request can be executed against the centralized state anytime.

It might seem that we arrived at a perfect solution to the problem. However, the safety of such a protocol depends on the specifics:

Escrow tokens

This strategy demonstrated on the decentralized exchange example is referred to as Escrow tokens by Sundaeswap in [5]. Similar to the constant factor improvement strategy in the auction example, it employs a constant number of requests that are identified by a special token limited in supply. All requests with the token need to be present in the batch transaction when requests are carried out.

The constant number of requests at any given point in time helps to guarantee that all created requests are carried out by whoever is responsible for the execution. The validator script is able to check that a specific constant number of request inputs is present in a transaction. If the number of requests was arbitrary, the smart contract would not able to verify that no request is left out and ignored.

The disadvantage of Escrow tokens strategy is that the concurrency issue is once again improved only by a constant factor because there can be multiple users competing to receive the escrow token that is limited in supply.

Open batching

Open batching stands for adhering to the simple request batching model and letting anyone batch requests and execute them. It allows for arbitrary number of requests which means that it is not able to guarantee on the contract level that all requests would be executed. However, the request creator could batch his request himself. He would be competing with other batchers for block inclusion, though.

There are two major problems with allowing anyone to batch transactions. Firstly, since the number of requests is arbitrary and the fact that validators see the pending transaction only, it is impossible to know how many requests there are on the contract level. As a result, it is impossible to prevent someone from batching 1 request only. We mentioned that this may not be desirable as it may mean that some requests are left out forever. What we did not mention yet is that this kind of behavior may actually conduct a denial of service attack on the protocol. This could be achieved by an attacker batching his own requests with near-to-zero value and leaving out all the other requests. Since only one batching transaction can be included in a block, the attacker doing so could limit the application throughput fundamentally.

Secondly, it may be important that the order in which requests were created is preserved in some applications, not manipulated and that the order in which they are batched is the same. An example of such an application is the decentralized exchange. We mentioned that performing a swap changes the exchange rate of assets. Controlling and manipulating the order of swap requests in their execution can be exploited for personal gain. An example of such behavior is **front-running** [25].

In front-running, an attacker can notice a swap big in quantity which is expected to tangibly change the exchange rate. He can create and put his own swap request before executing that request, profiting of the prior knowledge of the rate change. Similarly, he can execute his own swap changing the rate and let the unknowing user exchange tokens for an unfair rate.

Preserving the order is another problem that is not possible to enforce on the validator level given the limited validator scope. That is, if we assume an arbitrary number of requests.

2.4 Agent role

As we have seen, it is not a good idea to allow anybody to batch requests. Due to the outlined higher responsibilities of the batching party, the ability to batch requests is a priviledged role. In the following text, we call the batching party an agent and refer to his role as Agent role. If we assumed that any agent was an honest player, we could use the request batching solution to solve the concurrency issue and increase the throughput of decentralized applications in need of a centralized state. There would be close to no disadvantages to it except for it taking longer time to completion due to the process consisting of two transactions which both need to be created and accepted into the blockchain one after the other.

As described in section 1.3.2, the authentication of agents can utilize permission tokens. The script holding the state can validate that an agent token is present every time batching occurs.

It is not an easy task to identify honest players and allow only them to become agents, though. What's more, concentrating the power into a small number of agents has its risk, too. If all agents stopped batching, the protocol would halt as it is dependent on them.

Chapter 3

Optimistic design pattern

In this chapter we propose a design pattern which aims to address the problems arising from the Agent solution to the concurrency issue described in the previous chapter. We explain that the idea is to trust agents to be honest and disincentivize them from misbehaving. Further on, we discuss two ways of watching out for proofs of a potential misbehavior, list their pros, cons and potential usage in various applications.

3.1 Limiting the exploitable surface

It is important to note that agents can not do anything they like. By spending request scripts and the script UTxO holding the protocol state, any validation defined either in the state script UTxO or in request scripts can run on the batching transaction. It is critical for those validators to check as many attributes and invariants as possible, to narrow the possibly exploitable surface down to the bare minimum.

Unfortunately, as described in section 2.4, it is impossible to verify all properties we would like. Those unverifiable include certain properties such as enforcing the order of requests or checking if the presence of all created requests is provided in the transaction. This limitation is there mainly due to the limited scope of validators to the currently validated transaction only. This is evident in the latter example we provided. If the validator can not check the whole blockchain, how can he find out how many requests there are? He is able to see the requests included in the transaction only. If there is any request ommitted, it is not possible to spot it.

In contrast, it is still very possible to ensure a fair execution of the requests that are part of the batching transaction.

3.2 Punishing agent misbehavior

Optimistic design pattern is a pattern we would like to suggest to be used in a decentralized application that would like to use the request batching model to solve the concurrency issue. It acknowledges that some properties are not possible to ensure in Cardano blockchain by validator scripts only and that being an agent is a priviledged role. At the same time, however, it would like to decentralize agents as much as possible to avoid a single point of failure.

The design pattern enables anyone to become an agent by meeting the initial criteria. The criteria would include putting down a sufficiently big collateral consisting of tokens with monetary value and locking it inside a **collateral script**. The collateral serves to discourage agents from misbehaving. It is a lot of value that is locked by every agent. If agent is honest, nothing happens to the collateral and the agent can reclaim it after he wraps up being an agent. On the other hand, the protocol allows slashing of the collateral in case the agent is dishonest and misbehaves. The process of putting down the collateral and releasing an agent token (making the user an agent) for it can be automized, as can be seen in the next chapter 4 demonstrating the design pattern.

Being an agent and fulfilling the agent role needs to be a profitable job to do. If it was not, there would be no incentive for the public to want to do it which would understandably limit the resulting protocol decentralization. As a result, there is often a constant fee that can be taken from every request agents execute. It is possible to make sure that dishonest agents can not collect these agent fees by collecting them separately and holding them off until a sufficient time passes during which no slashing took place. Losing the locked collateral is definitely more disincentivizing, but this can effectively increase the value that the agent has in stake.

Finally, it is crucial to be able to take the agent token back. The agent token enables the agent to perform agent duties. If we were unable to take the token back, the agent would have nothing to lose after his collateral was already slashed and could continue manipulating the protocol for his own gains. We achieve this by never distributing agent tokens directly into agent wallets as that way we would have lost any control over them, but by locking agent tokens into **agent scripts** instead. Agent scripts ensure that the owner of the script can use the agent token in order to batch requests. What's more, it allows for slashing of the agent token if there is a proof of agent misbehavior. It can not be possible for the agent to unlock and take the agent token away.

Optionally, since the agent token is locked in a script already and the **agent script** validation needs to pass whenever it wants to be used, it is possible to control many other conditions as well. An example we already mentioned could be limiting the agent to collect agent fees into a pre-defined script which can ensure that a sufficient time passes without the agent getting slashed before releasing the accumulated fees into the agent's wallet. Another example would be limiting the frequency of the agent being able to use his agent token.

3.3 Proofs of agent misbehavior

In order to slash agent's collateral and take away his ability to batch requests, it is critical to obtain a proof of agent misbehavior. This section discusses two main types of how such proofs could look like and how the following agent slashing could be designed.

3.3.1 On-chain proofs

On-chain proofs refer to the ability to find and use proofs already directly located on the blockchain. It is arguably the most elegant solution as everything is controlled and verified purely by validator scripts. It is also the more complex approach to implement. Sadly, it is not possible to be used for every type of a decentralized application as the existence of such proofs in the form of UTxOs is not natural for all applications.

In the auction example described in 2.1.1, an on-chain proof of an agent misbehavior could be an overlooked and left-out bid request that was not executed. If we were able to verify that the bid request was created at a certain time during which the auction was still ongoing, it was the highest bid, still is, and the auction was later closed by concluding another smaller bid as the highest, we could take the agent that concluded the auction and punish him for clearly overlooking and not executing the bid request he was supposed to. We could then penalize him by slashing his collateral and taking away his ability to be an agent.

As we will demonstrate on the auction example in the final chapter 4, it is actually possible to fulfill all of the mentioned prerequisites and make this work.

3.3.2 Off-chain voting proofs

In contrast to on-chain proofs, off-chain voting proofs can be utilized for any application and in every scenario that is at all detectable by anyone. It builds up from the fact that the whole blockchain is publicly available and explorable, including all transactions that happen and the order of transactions in which they were accepted into the chain. Even though validator scripts see only a limited scope, looking at the blockchain history from an off-chain perspective, one can see everything and can look out for whether a particular agent misbehaved or not.

The idea of off-chain voting proofs utilizes an inner governance structure of the application itself to find and vote for agent misbehaviors. The governance structure refers to any **decentralized autonomous organization** (DAO) structure there is. Most truly decentralized applications employ one as a DAO is essentially a decentralized form of the protocol ownership whereas thousands, even millions of people can influence the future of the project. People that are part of the DAO benefit from the fair and error-free run of the application. As a result, they are naturaly incentivized to stop and punish any dishonest agent as soon as possible. There is often a governance token that identifies the shareholders that is limited in supply [26].

Anyone who is part of the structure can claim that a particular agent misbehaved and optionally include details about it. This could be done purely off-chain, announced e.g. via social media or similar. That action starts a voting. Any project governance token holder can validate the claim off-chain themselves since the whole blockchain data is public and cast a vote on-chain to bindingly express what they found out about the agent.

Casting votes on-chain can easily prevent double-voting since casting any vote could require the person to lock his governance tokens into the vote script for as long as is needed in order to get the voting results. Note that in this voting, anonymity of the vote participants nor confidentiality of the votes casted are not essential properties. The anonymity of the participants is provided partly by the fact that there are no names in the blockchain ledger, only public key hashes. It is not a trivial job to assign names to public keys. It requires some forensic analysis. Despite that, the field is studied and it is often very possible to find out the identity of a person holding the public key [22]. It happened even in the real life when there was enough incentive to do it.

All the votes can be seen by anyone who is looking at all UTxOs present on the voting script address from off the chain. He can weigh the votes by the number of governance tokens locked in the votes and find out the results.

However, it is a problem to make the results available in a single UTxO on-chain in order to be read by validator scripts. We need that in order to present a unified proof of agent misbehavior that can be included in the transaction slashing the agent's collateral. The reason why it is a hard problem is that the number of project shareholders that voted can be arbitrarily big. Naïve solutions could easily exceed the maximum transaction size. Solving this would require a very complicated, long-taking and expensive on-chain consensus procedure to count all the votes.

Instead, to bring the voting results on-chain, we assume that a generic **oracle solution** already exists and we utilize it. An oracle is a decentralized application whose purpose is bringing real-world data into the blockchain. There are numerous oracle solutions in Ethereum [13], the leading blockchain in the number of decentralized applications and there are plans for launching an oracle on top of Cardano soon [9] [14].

An oracle functionality is by definition exactly what we need to bring the voting results into a single UTxO datum on the blockchain. Having achieved that, we can later use the results as a proof of agent misbehavior and create a transaction punishing him if the vote concluded that he actually misbehaved.

Chapter 4

Demonstration: Auction

In this chapter, we describe a decentralized NFT auction application employing the agent request design and demonstrate the Optimistic design pattern with the on-chain proof approach on it. We explain various technical decisions in great detail along the way. We implemented the contract code that we are about to discuss, along with an off-chain demonstration of the described application. It is located on the USB drive included with this thesis.

4.1 Protocol design

We will demonstrate the Optimistic design pattern on a similar auction to the one mentioned in the previous text in 2.1.1 and 3.3.1. Sellers owning NFTs could put them up for an auction. Users can bid and the user bidding the highest price will buy the NFT. Multiple auctions being executed at once are easily possible to take place in parralel as they do not influence each other other than by increasing the number of transactions that are waiting to be accepted into the blocks which could increase the time it takes for any action.

There is a single UTxO for every auction owning both the NFT put to sale and money of the highest bidder if there is one already. It is responsible for both the locked bid money and the NFT to be able to guarantee that the bidder could be granted the NFT and at the same time, the seller could be payed upon the auction closing.

In order to avoid concurrency issue limitations when multiple people would attempt to bid for the same auction in the same block (see Chapter 2), we use the agent request model. Agents are the only ones that are able to take the currently highest bid request UTxO, take the auction UTxO with the previously highest bidder and mark the outbidding.

Anyone is able to become an agent allowing for a true decentralization. A collateral is required to be put down in order to become an agent. Agent token identifying agents is minted when that happens and is locked inside an agent script along with the collateral itself.

Anyone is free to watch out for a proof of agent misbehavior. We utilize on-chain proofs in the form of a left-out highest bid which was clearly overlooked by a particular agent. Given such proof exists, it can be included in a transaction that slashes the agent of his collateral and burns his agent token, effectively taking away his ability to harm the protocol further. This works unless he put down multiple collaterals in exchange for multiple agent tokens. In that case, he is given another chance which could result in slashing of another collateral if he misbehaves again.

To be able to fairly slash misbehaving agents and avoid slashing honest ones, we enforce that:

- **We keep track of the last agent interaction with auction UTxO**. This is part of the auction script datum and it is enforced by the auction validator. We keep track of both the timestamp when the agent interaction occurred and the agent that interacted with the auction.

- **We know the time bids are created**. This is tricky. In general, we are unable to run any validation on the creation of UTxOs. However, we can utilize validity tokens mentioned in 1.3.2 for this. The token for this use case is called the **bid validity token**. Its minting policy is written in such a way that bids owning this bid validity token are guaranteed to have been created around the timestamp written in their datums. See 4.2.2 for more.

- **Auction is properly created**. As we mentioned, the agent that last interacted with an auction is mentioned in the auction UTxO datum. This information is used in order to slash that agent if applicable. If an auction UTxO could be created with a malicious datum referencing an agent that did not interact with the pool, he could be slashed even though he did nothing wrong.

  To avoid this, we utilize an **auction validity token**. It can be minted only into a freshly created auction UTxO, needs to be present in auction UTxO at all times and makes sure that the auction datum is not malformed.

Furthermore, we want to avoid a misbehaving agent slashing himself such that he is granted the slashed collateral. If we allowed this, the agent would not lose anything on it and could play the protocol. To make this right, we require that most of the slashed collateral goes to the protocol treasury. Protocol treasury address is fixed for the auction. We still allow for the person slashing to take part of the collateral to incentivize people looking out for malicious behavior.

With all the mentioned pre-requisities, we are now able to fairly slash a misbehaving agent and take away his agent token. Let's take a look at the specific elements of the application in more detail.

4.2 Tokens

Same as in most applications, we also make use of different tokens in our auction application. They are functional tokens essential to the correct functioning of the application. As such, it is critical that they follow a strictly defined closed lifecycle. That is important to be aligned with the reasoning given in this text.

To make sure that a functional token is not misplaced, we enforce that in order to mint a functional token, there can be no such token in transaction inputs, no script where it is to be minted is present in the transaction inputs and that only one such token is minted in the transaction and it is minted into the correct script found by its hash. This holds for all tokens described in this section.

4.2.1 Agent token

Agent tokens identify agents. It can be part of agent script UTxOs only (see 4.3.1) and exactly one in quantity. It can be minted only in the transaction where a user becomes an agent, into the agent script UTxO and can never leave the agent script unless it is burned.

Whenever that agent script that holds agent token is spent in a transaction, it is carefully checked that the agent token is not taken away and is present in the agent script transaction output.

Agent token ends its lifecycle in either of two scenarios: Either the agent stops being an agent, he unlocks his collateral and burns the agent token or a proof of his misbehavior is found and he is slashed in which case the agent token is burned as well.

In a way, agent token also serves as an agent script validity token. The minting process of agent tokens also checks that the collateral is put down and locked inside the same agent script that is being created. Thanks to that, we can say that any agent script UTxO holding an agent token has collateral locked inside of it.

4.2.2 Bid validity token

As mentioned in 4.1, a bid validity token is minted only when a new bid script (see 4.3.2) UTxO is created and is a property of that particular bid until the bid is not used or cancelled in which case it is burned.

Its purpose is to limit the transaction validity range of that transaction to big enough to cover for blockchain latencies but at the same time short enough to provide a good estimate of the time when the bid was created. It then checks that this deterministic estimate given the transaction validity range is put inside the bid script datum. Given bid script does not allow for a modification of this timestamp, we can say that any **valid** bid contains a time approximation of the time it was created. A bid is considered valid if and only if it holds a bid validity token.

4.2.3 Auction validity token

Similar to bid validity tokens, as mentioned in 4.1, an auction validity token is minted only when a new auction script (see section 4.3.3) UTxO is created and is a property of that particular auction until the auction is not concluded in which case it is burned.

Its purpose is to mainly check a proper auction datum creation. That means that no agent interaction is recorded in the datum. It acts as a defense mechanism preventing an unfair slashing of honest agents.

In addition to that, we also check that the auction contains the auctioned NFT asset. That makes it easier for bidders to trust that any auction with auction validity token claiming to be auctioning a particular NFT actually holds it. Furthermore, it prevents malicious agents from applying the bid with a malicious auction UTxO that only says it auctions the asset the bidder wants, but it does not really own it.

4.3 Smart contracts

There are three scripts that form the auction application together. Every script holds a state and allows multiple redeemers that describe the possible actions. Let us explain how the different validators work together.

4.3.1 Agent script

Agent script is holding the collateral and the agent token identifying agents. Owning such script enables the owner to participate in the protocol.

Agent datum

Agent datum is the state of the agent script. It consists of:

- **Owner**. Owner is a public key hash of the agent, the owner of the agent script instance. Only he is able to take that agent script UTxO and use it to apply a bid with an auction. No one else can use this UTxO in the name of the owner. If he wants, he can take his money, put down collateral and become an agent himself.

- **Last used at timestamp**. This timestamp tracks the last usage of the particular agent script UTxO. It is guaranteed to be empty when agent script is initialized (checked by the agent token minting policy) and needs to be updated to the current time approximation whenever an agent uses the script to apply a bid.

- **Auction validator hash**. In order to check that an auction is present in a transaction, we need to know the auction script validator hash to look for. We could fix it as a constant inside of the agent script validator, but that would mean that the agent script validator hash is dependent on the hash of the auction script. In turn, if we wanted to check the presence or any other property of the agent script inside of the auction validator, we could not reference the agent validator hash as that would cause a circular dependency.

  We avoid this issue by putting the agent script validator hash as a parameter of the auction script which makes the auction validator hash dependent on the agent script validator, but instead of putting the auction hash into the code of the agent validator, we make it part of the agent script state. This helps us to avoid the circular dependency while also making it up to the agents that create the agent script UTxOs to provide the correct hash. We argue that we can rely on the agents in this sense. That's because they want to be able to apply bids to auction UTxOs. To be able to do that, they need to reference the correct auction hash, because the auction has the agent hash as a parameter and its validator checks that the agent script datum references the correct hash.

  To sum up, the agent needs to reference the auction validator hash. If he puts the correct hash there, everything will work just fine for him. If he does not, he is unable to interact with the auction script UTxOs so he loses the very ability he put down collateral for.

- **Auction validity token class**. Having this token class follows exactly the same logic as the previous property, the auction validator hash. It is necessary for agents to not trust auctions that do not hold the validity token. If they failed to do that, they could be e.g. slashed due to the invalid auction's datum being malformed in a way that it says that agent overlooked a certain highest bid.

It is also checked in the auction validator that any agent script applying a bid references the correct auction validity token class. Hence, it is not possible for the agent to avoid the slashing possibility in case he provided wrong auction validity token class.

Redeemers

There are following actions that can be performed on an agent script:

- **Use agent token**. This is a redeemer to be used when an agent wants to use his agent token to apply a bid in an auction. It is checked that an owner of the agent script signed the transaction to verify that the owner really wants to do it. Moreover, it is checked that the value is not changed to prevent the agent from taking away his collateral and / or the agent token and that the **last used at** timestamp is correctly updated. To update it to a good enough time approximation, it is checked that the transaction validity range is limited enough. No other datum value can be changed. It is also checked that an auction UTxO with the correct hash and with the correct validity token is present in the transaction inputs.

  To narrow down how the transaction looks, it is ensured that there is one agent script and one auction script in transaction inputs and that one of each is also present in the transaction outputs.

- **Terminate agent role**. This redeemer can be used only by the owner of the agent script instance to stop being an agent and take back the locked collateral. This destroys the UTxO and as such needs to burn the agent token contained within. The collateral is free to be taken by the owner whose signature needs to be present in the transaction. In order for the agent to be able to terminate the agent role, he needs to wait for sufficiently long after his last interaction with an auction. This time is enforced in order to avoid agents cashing out after they harm the protocol, allowing for a sufficiently long time to slash them.

  To prevent various edge cases, it is explicitly checked that no auction is present in such a transaction and that only one agent script was spent.

- **Slash agent**. This action can be performed by anyone, the agent owner's signature is not required. The actual validations are delegated to the auction validator. In the agent validator, we just check that an auction with the validity token is present both among the transaction inputs and the outputs and that the agent script is destroyed. Agent script destruction needs to be accompanied by agent token being burned as we want to maintain the invariant that the agent token always lives only inside an agent script.

Note that the enforcement of the number of script inputs and outputs in such transaction serves to make sure a particular redeemer is used on the auction script. In this case, checking an ongoing auction is present and an agent script that is destroyed clearly identifies the only valid possibility and that is the **Read for slashing** auction redeemer (see 4.3.3).

4.3.2 Bid script

The bid script UTxOs are used as places where bidders express which NFTs they wish to buy and for how much. To express a binding wish, they are required to lock the money inside the bid script. That money in itself is not binding as long as it is contained within the bid UTxO. However, an agent can come, spend the bid and put the money inside an auction UTxO. Once the money is in the auction UTxO, there is no way for the bidder to cancel the bid and retrieve the money. He either wins the auction and gets the NFT or he is outbid and his money is returned. That is why we say that the wish to bid is made binding by locking the money inside the bid script instance.

The bid script needs to protect the bidder. As long as the bid UTxO exists, the bidder needs to always be able to take away his money. Once it is destroyed by an agent, the money can only be put inside the auction. Finally, once the money is inside the auction, he can either be outbid and returned the money or he can be compensated. The bidder can never lose on this. He creates the bid UTxO to either win the auction or not to lose anything except the transaction fees. That is the contract he is signing for and the bid script needs to protect this.

Bid datum

Bid datum is the state of the bid script. It consists of:

- **Owner**. Owner is a public key hash of the bidder. Only he is able to cancel the bid and unlock all of his locked funds.
- **NFT asset**. The asset class of the NFT which uniquely identifies the NFT the bidder offers his money for.
- **Bid created at**. A timestamp that approximates the time of when this bid was created. As described in 4.1, validity of this timestamp is guaranteed for all valid bids identified by an ownership of a bid validity token.

- **Auction validator hash**. The auction script validator hash. It can not be in the code itself and needs to be in the datum to prevent circular dependency between the auction and bid scripts.

  It is in the bidder's best interest to put the correct hash into the datum. By putting a different hash there, he has nothing to gain and everything to lose. An auction with the hash and owning the validity token is free to take the bidder's money in exchange for a promise that he is currently the highest bidder and he would either win the auction or be returned the money. Putting the correct hash there, he trusts the code of the auction that promises this. Putting the wrong hash there, he is subject to any alternative logic that the other script stands for. For example, if the script whose hash was put there would validate every time with no validation logic at all, anyone would be able to construct a transaction unlocking the bidder's money that is locked inside the bid referencing the wrong hash.

- **Auction validity token class**. This follows the same logic the auction validator hash does. The bid script's code refuses to interact with auctions not owning the validity token.

Redeemers

There are following actions that can be performed on a bid script:

- **Use bid**. This redeemer actually serves two different actions. It can either be used to outbid a previously highest bidder or it can be used to use the bid as a proof of agent misbehavior. In any case, the bid script actually does not check much. It delegates the validation to the auction validator which allows for exactly these two distinct use cases.

  It checks that there is exactly one auction input in the transaction and one output. It also checks that the auction owns the auction validity token and that the auction sells the NFT that the bidder wants.

  As there are two separate use cases allowed to use this redeemer, it is not clear whether the bid UTxO will be destroyed or not and thus not appropriate to check anything about the bid validity token burning in this place. It is checked, however, in the respective auction script validation paths.

- **Cancel bid**. The owner of the bid can cancel the bid anytime. It is checked that the owner signed such a transaction and that the bid validity token is burned.

4.3.3 Auction script

An auction script instance is initialized by a seller who wants to auction an NFT. The NFT is locked there. We maintain that the NFT is a valuable item and therefore one of the responsibilities of the auction script is to protect the NFT. It can be either sold or taken back by the seller in case no bidder participated in the auction.

In addition to that, the auction is the cornerstone of all the remainder logic. Both agent script and the bid script delegates some parts of their validations to the auction script. Therefore, it needs to carefully check as much as possible to satisfy all parties.

Auction datum

The auction datum is the state of the auction script instance. It consists of:

- **Seller**. Seller is a public key hash of the person creating the auction UTxO. Only he can cancel the auction in case no one bid yet. In case someone bid already, he is the person that needs to be paid the bid amount upon the auction closing.
- **Auctioned asset**. This identifies the asset that is being auctioned. The auction validity token makes sure that the asset is indeed inside of the auction UTxO. Furthermore, bid script makes sure that a bid interacts only with an auction that sells the asset that the bidder wants.
- **Deadline**. The deadline of the auction. There can be bids accepted only before this deadline. After the deadline has passed, the auction does not accept any more bids and can be either closed if there is a bidder or it can be cancelled in case no one bid.
- **Highest bid**. Highest bid keeps track of the highest bidder and the offer there is so far. It is checked that there is no highest bid upon the auction UTxO creation in the auction validity token minting policy. Every following offer needs to be strictly greater than this currently highest one.
- **Last agent interaction**. This property keeps track of the last agent interaction, both the timestamp and the public key hash of the agent script owner. It is ensured that no agent interaction is set upon the auction creation. This value is used to identify the agent that overlooked the highest bid and thus is considered to have misbehaved.
- **Auction validity token class**. Similar to other validator hashes and token class references, this field is here in order to avoid circular dependencies. This value is not to be trusted by agents nor the bidders. They need to set the correct value inside their agent or bid datums themselves. It can not happen that the

auction will reference wrong auction validity token class and still contain the correct validity token. This is prevented by the auction validity token minting policy. However, it still could happen that a malicious auction UTxO would reference a wrong token class. That is one of the reasons why agents as well as bidders reference the auction validity class themselves and do not rely on any hash that is put here.

It is used to make sure the auction validity token is burned in case of an auction closing or cancellation.

Redeemers

There are the following actions that can be performed on an auction script:

- **Outbid**. This action is applicable when a higher bid is included in the transaction. This bid amount is locked inside the auction UTxO and the previously highest bidder is returned his money. This is applicable only if the auction deadline has not passed yet and can be initiated by an agent only. As discussed, that is to avoid the concurrency issue that would otherwise occur when multiple users would like to interact with the auction UTxO themselves in parallel.

  It checks that the agent references the correct auction hashes, the offer is high enough and that the auction datum and value are exactly correctly modified which includes storing the data about that concrete agent and his interaction. There needs to be exactly one bid script, one agent script and one auction script as part of transaction inputs and one agent script and one auction script in the transaction outputs. The bid is thus destroyed. It is checked that the bid validity token is burned.

  It is also made sure that only valid bids are able to outbid an auction. This is to strictly enforce the way users should use the protocol. Avoiding the minting of bid validity tokens could result in inability to slash agent's collateral as we could not trust bid's creation timestamp unless it owns the bid validity token.

- **Close auction**. Closing of an auction means that there is a winner of the auction, deadline has passed and this is the time that the NFT is sent to the highest bidder and the offerred money goes to the seller.

  As it is a one-off action per any auction, there are no problems with concurrency. It does not matter who performs this action. It is strictly defined who receives what. The transaction initiator simply pays for the transaction. As a result, it is expected that the person would be one of the two participants as they are the ones interested in closing it off.

It is checked that the deadline is in the past, seller as well as the winner are compensated, the auction UTxO is destroyed and thus the auction validity token is burned. There needs to be exactly one auction UTxO in the transaction outputs and none in the outputs.

Furthermore, there can be no bid scripts present whatsoever in neither the inputs nor the outputs.

- **Cancel auction**. The seller is able to cancel his auction if his signature is present, **no bidder's money is locked in the auction so far**, the auction UTxO is destroyed and the auction validity token is burned. To stress it out, the most important constraint is that there is no bidder's money locked in the auction so far. This way, the auction can guarantee the bidder that if he really is not outbid by someone else, he will win the NFT as the buyer can not cancel the auction.

  The seller is free to take his NFT wherever he wants when he cancels the auction. Additionally, there can be no bid script included in the transaction inputs or outputs.

- **Read for slashing**. The slashing transaction consists of a left-out bid which is left untouched, the auction it was supposed to be applied with that is also left untouched and acts as a part of the proof and, finally, the agent script that is being destroyed.

  Technically, the auction datum is modified a bit to reset the last agent interaction field to no interaction. This is purely to allow only one slashing of a particular agent for a single auction misbehavior at a time. He needs to misbehave again to be subject to another slashing. Most of the collateral needs to be put inside the treasury. The rest, however, can be taken by anyone that found about the agent's misbehavior. The agent token needs to be burned.

  The slashing is possible whenever:

  1. Both the auction and the bid contain validity tokens.
  2. The bid would have won if it were applied.
  3. The bid is not cancelled.

4. Bid was created before there was the last agent interaction with the auction, meaning the agent should have seen it and applied that bid instead. There is an agent tolerance time interval constant in the implementation, as agents can not be expected to react straight away. Accepting transactions into blocks takes non-trivial time making zero-tolerance practically impossible.
5. The agent that is being slashed is the one that last interacted with the auction.

4.4 Transactions

To summorize the behavior and offer a different point of view on the scripts' interaction, let us describe all the trasactions possible in the protocol. Even greater details and constraints are offerred in the runnable off-chain part of the actual implementation that is included on the USB coming together with this text.

4.4.1 User transactions

The following transactions are able to be built and submitted by any user.

Create auction

Anyone owning an NFT they wish to auction can take it, lock it inside a newly created auction script UTxO and set the auction deadline in the datum freely. In order to create a valid auction, it is needed to mint an auction validity token in that transaction and put it into the auction UTxO.

Close auction

Anyone can find auctions with a clear winner that are past their deadlines and close it off. They are required to pay the winner off the NFT that is locked in the auction UTxO and pay the seller the bid money that is also locked in the auction UTxO. It is also necessary to burn the auction validity token from the auction UTxO.

Cancel auction

Anyone who is an owner of an auction can cancel their own auctions and retrieve the locked NFTs from them in case there is no bid applied yet with them. They need to destroy the auction UTxOs which needs to be accompanied by the burning of the auction validity tokens.

Create bid

Anyone who is interested in buying a particular NFT can offer money for it by creating a new bid script UTxO while locking the money in there. In the datum of the newly created bid UTxO, he can express exactly which NFT he bids for. In order to create a valid bid, it is required to mint a bid validity token in this transaction and put it inside the bid UTxO. To be able to do that, it is necessary to restrict the transaction validity range to not be arbitrary big. That serves to be able to approximate the current time close enough and to properly estimate when the bid was created.

Cancel bid

Anyone who has previously created a bid is free to cancel their bids as long as they still exist. Applying a bid (see 4.4.2) destroys the bid and thus the pure existence of the bid means that the bid was not applied. The cancellation destroys the bid by the bid owner and thus it is necessary to burn the bid validity token contained within.

Slash an agent

Anyone is free to observe that a particular agent misbehaved by identifying an auction and a bid belonging to that auction which would have outbid the highest bid marked in the auction. The bid needs to have been created before there was the last agent interaction on that auction allowing for some time tolerance to clearly identify that the agent written down in the auction's datum overlooked the highest bid and did not apply it. This makes for undoubtedly malicious behavior and is punishable. Anyone that observes and successfully creates the slashing transaction can take a significant part of the agent collateral for himself.

The transaction itself looks like this, describing only the transaction structure that is enforced:

- **Transaction inputs**.
  - The overlooked highest bid. The bid could not have been cancelled. If it were, we would have nothing to include here.
  - The auction it belongs to.
  - The agent script of the agent that is responsible for the auction's last agent interaction.
- **Transaction outputs**.
  - The overlooked highest bid that is left untouched.

  - The auction where the last agent interaction is reset. Otherwise it is left untouched.
  - The most of the collateral goes to the protocol treasury.
  - Small but still lucrative part of the collateral goes anywhere the user creating this transaction pleases.

- **Minting**. The agent script is destroyed. Thus the agent token previously contained within needs to be burned.

Become an agent

Anyone is free to become an agent. An agent is identified by an agent token and thus a person aiming to become one needs to mint the token. The minting policy allows minting agent tokens only into agent scripts given a collateral is put down and locked into the script as well. As a result, the transaction consists of the creation of a new agent script UTxO where the user collateral is locked and the newly minted agent token is locked. The owner public key hash of the agent script is specified freely by the user.

4.4.2 Agent transactions

The following transactions are able to be built and submitted by an agent.

Apply a bid

Applying bids is an action only agents can perform. The reason is the concurrency issue we would otherwise suffer from that is described in Chapter 2. The process of applying a bid first of all requires the agent to scan the blockchain off-the-chain and find the highest bid for that particular auction. After he has identified it, he spends the bid, takes the auction UTxO and outputs a new auction UTxO where the newly identified highest bidder would be noted down. Spending the bid involves burning the bid validity token contained within.

In case there was previously a highest bidder noted in the auction UTxO, there was the bid money locked in the auction. The agent is thus required to pay him off the whole amount as he stopped being eligible for the auctioned NFT. The agent puts the newly highest bid amount into the auction UTxO instead.

To identify as an agent, the auction validator requires an agent token present in the transaction. As a result, the agent is required to include his agent script UTxO in the transaction as well. He just presents it in the transaction and outputs it as well.

Finally, it is necessary to update the state of both the auction and the agent script in order to keep track of the last interaction. The auction needs to remember who and when lastly interacted with it to allow for fair slashing if applicable. The agent script needs to keep track of the timestamp of the last usage of the script to be able to wait sufficiently long before it allows the agent to retrieve his collateral. To be able to approximate the current time, it is necessary to enforce a small enough transaction validity range.

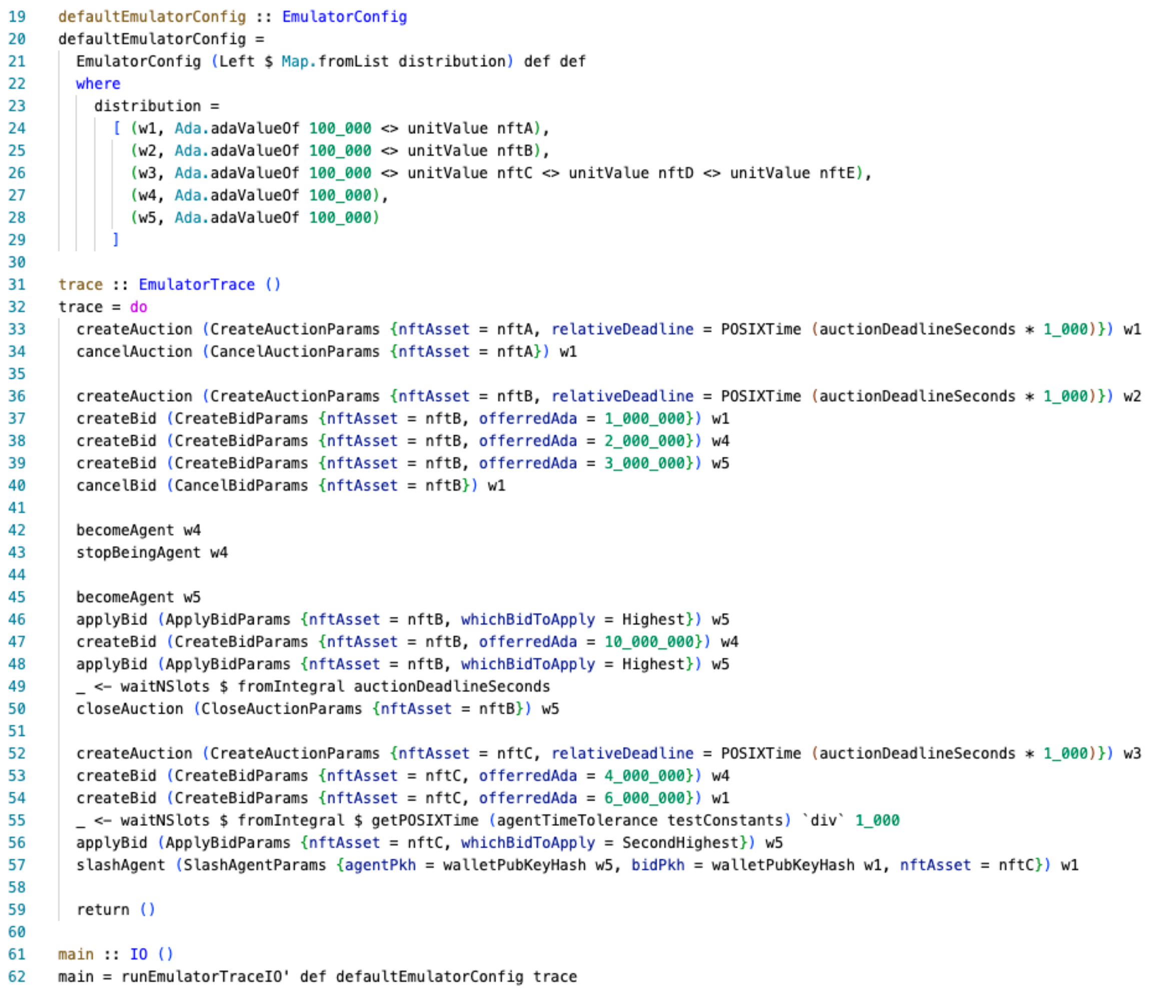
Stop being an agent

An agent put down collateral in order to be able to become an agent. He is free to stop being an agent and retrieve his collateral. The only requirement is that he needs to wait sufficiently long after his last usage of the agent token in order to allow for observers to find out whether he misbehaved during his agent interaction or not. After the required time has passed, he can destroy his agent script UTxO, burn the agent token and take the collateral wherever he wants.

4.5 Implementation testing

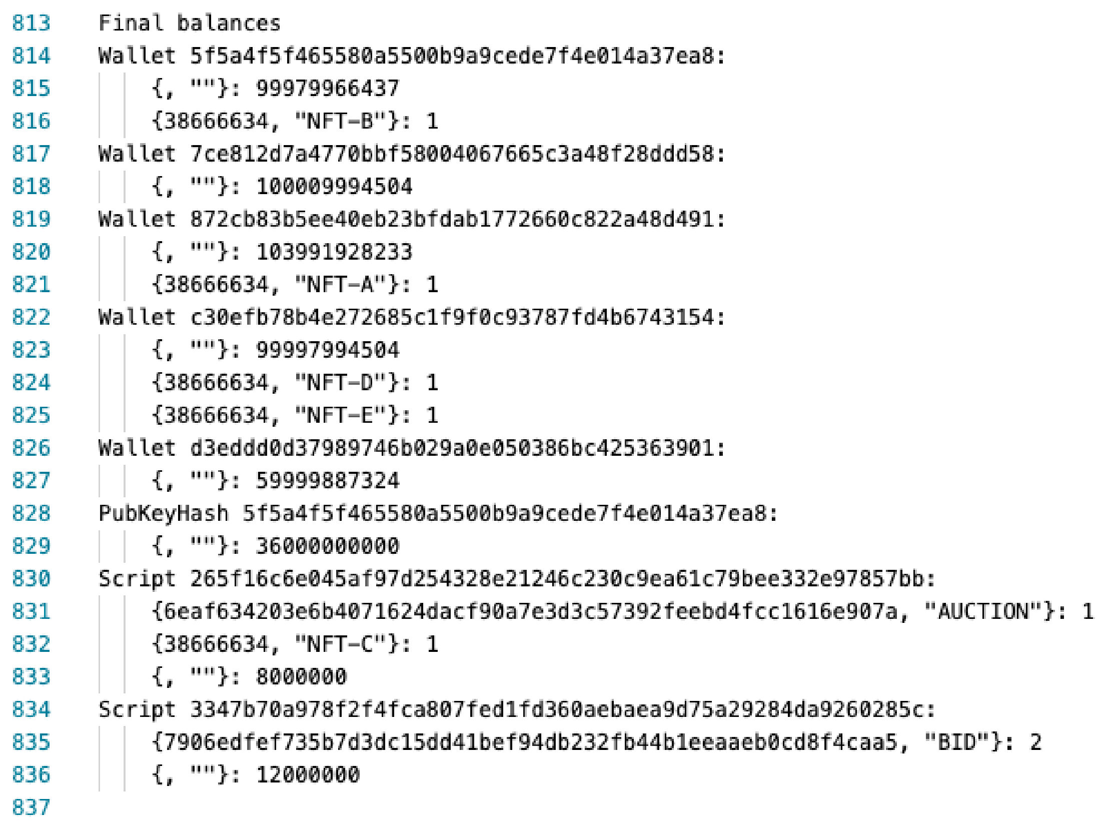
We have run and tested various scenarios combining the above transactions. It can be tested out modifying the **Trace.hs** file, compiling and running it. More information about how to run it can be seen in the **README.md** file. Logs from the execution of the flows listed in the file are also located on the USB drive in file **trace.logs**.

We tried to simplify the testing interface so that it can be easily modified even without extensive knowledge of the Haskell language. Example snippet can be seen in Figure 4.1 and sample logs from such an execution in Figure 4.2.



```haskell
defaultEmulatorConfig :: EmulatorConfig
defaultEmulatorConfig =
  EmulatorConfig (Left $ Map.fromList distribution) def def
  where
    distribution =
      [ (w1, Ada.adaValueOf 100_000 <> unitValue nftA),
        (w2, Ada.adaValueOf 100_000 <> unitValue nftB),
        (w3, Ada.adaValueOf 100_000 <> unitValue nftC <> unitValue nftD <> unitValue nftE),
        (w4, Ada.adaValueOf 100_000),
        (w5, Ada.adaValueOf 100_000)
      ]

trace :: EmulatorTrace ()
trace = do
  createAuction (CreateAuctionParams {nftAsset = nftA, relativeDeadline = POSIXTime (auctionDeadlineSeconds * 1_000)}) w1
  cancelAuction (CancelAuctionParams {nftAsset = nftA}) w1

  createAuction (CreateAuctionParams {nftAsset = nftB, relativeDeadline = POSIXTime (auctionDeadlineSeconds * 1_000)}) w2
  createBid (CreateBidParams {nftAsset = nftB, offerredAda = 1_000_000}) w1
  createBid (CreateBidParams {nftAsset = nftB, offerredAda = 2_000_000}) w4
  createBid (CreateBidParams {nftAsset = nftB, offerredAda = 3_000_000}) w5
  cancelBid (CancelBidParams {nftAsset = nftB}) w1

  becomeAgent w4
  stopBeingAgent w4

  becomeAgent w5
  applyBid (ApplyBidParams {nftAsset = nftB, whichBidToApply = Highest}) w5
  createBid (CreateBidParams {nftAsset = nftB, offerredAda = 10_000_000}) w4
  applyBid (ApplyBidParams {nftAsset = nftB, whichBidToApply = Highest}) w5
  _ <- waitNSlots $ fromIntegral auctionDeadlineSeconds
  closeAuction (CloseAuctionParams {nftAsset = nftB}) w5

  createAuction (CreateAuctionParams {nftAsset = nftC, relativeDeadline = POSIXTime (auctionDeadlineSeconds * 1_000)}) w3
  createBid (CreateBidParams {nftAsset = nftC, offerredAda = 4_000_000}) w4
  createBid (CreateBidParams {nftAsset = nftC, offerredAda = 6_000_000}) w1
  _ <- waitNSlots $ fromIntegral $ getPOSIXTime (agentTimeTolerance testConstants) `div` 1_000
  applyBid (ApplyBidParams {nftAsset = nftC, whichBidToApply = SecondHighest}) w5
  slashAgent (SlashAgentParams {agentPkh = walletPubKeyHash w5, bidPkh = walletPubKeyHash w1, nftAsset = nftC}) w1

  return ()

main :: IO ()
main = runEmulatorTraceIO' def defaultEmulatorConfig trace
```

Figure 4.1: A snippet from **Trace.hs** file building transactions and testing out the user balance updates.



```
Final balances
Wallet 5f5a4f5f465580a5500b9a9cede7f4e014a37ea8:
    {, ""}: 99979966437
    {38666634, "NFT-B"}: 1
Wallet 7ce812d7a4770bbf58004067665c3a48f28ddd58:
    {, ""}: 100009994504
Wallet 872cb83b5ee40eb23bfdab1772660c822a48d491:
    {, ""}: 103991928233
    {38666634, "NFT-A"}: 1
Wallet c30efb78b4e272685c1f9f0c93787fd4b6743154:
    {, ""}: 99997994504
    {38666634, "NFT-D"}: 1
    {38666634, "NFT-E"}: 1
Wallet d3eddd0d37989746b029a0e050386bc425363901:
    {, ""}: 59999887324
PubKeyHash 5f5a4f5f465580a5500b9a9cede7f4e014a37ea8:
    {, ""}: 36000000000
Script 265f16c6e045af97d254328e21246c230c9ea61c79bee332e97857bb:
    {6eaf634203e6b4071624dacf90a7e3d3c57392feebd4fcc1616e907a, "AUCTION"}: 1
    {38666634, "NFT-C"}: 1
    {, ""}: 8000000
Script 3347b70a978f2f4fca807fed1fd360aebaea9d75a29284da9260285c:
    {7906edfef735b7d3dc15dd41bef94db232fb44b1eeaaeb0cd8f4caa5, "BID"}: 2
    {, ""}: 12000000
```

Figure 4.2: Final balances after the series of transactions have run. Token {,””} stands for ADA which is the native Cardano currency.

Conclusion

We studied the extended unspent transaction output model of Cardano. Firstly, we made an introduction into the blockchain in general. Then we focused on the eUTxO model of Cardano and deep dived into technical specifics. After the reader comprehended that, we went on and showed that centralization of an application state in the form of a single datum is very common across decentralized application designs. Such centralization, however, often results in adverse concurrency issues.

We offered an overview of a variety of solutions currently considered for minimizing the concurrency issue impact. We picked the agent-request solution as it turns out to be the solution with most advantages and least disadvantages and worked with that.

We introduced and defined a new design pattern that takes the agent-request solution to a next level that is not yet seen on the market. Assuming economically driven rational attackers, it enables for a full decentralization of agents with a strong disincentivizing punishing factor in case the priviledged agents try to play the protocol and someone finds out about it.

Moreover, we covered two different types of how such proofs of agent misbehavior could look like and how the whole punishing machinery could work either natively on the eUTxO model in the form of additional validator scripts or how it could employ an already existing oracle solution combined with off-chain voting.

Finally, in order to not be too abstract, we demonstrated the native eUTxO proof finding machinery on a simple demo NFT auctioning application which we coded from scratch and that is included with this text. In the last chapter, we discussed the implementation along with our reasoning behind the decisions impacting the overall design and security of the demonstrated application.

For the future research we suggest conducting further analysis of what kinds of on-chain proofs are available in what application use cases and building a proper hierarchy out of that. It could also be an interesting topic to split the collateral into many different agent tiers. In our demo auction example that could have the form of letting only the agents with the most at stake interact with the most valuable auctions.

Another idea is to form the tiers based on the reputation and history. The agents that were honest for a long time and never did anything wrong would therefore be rewarded for their honesty. There could even be a peer-review process whereas new agents would require older agents' supervision and take e.g. smaller fees until they prove their honest incentive in time.

Bibliography

[1] Bitcoin: A peer-to-peer electronic cash system. *www.bitcoin.org*, 2008. https://bitcoin.org/bitcoin.pdf.

[2] The extended utxo model. *Financial Cryptography and Data Security*, 2020. https://iohk.io/en/research/library/papers/the-extended-utxo-model.

[3] Alonzo hard fork upgrade successful, Sept 2021. https://twitter.com/InputOutputHK/status/1437174002603204609.

[4] Concurrency and all that: Cardano smart contracts and the eutxo model, Sep 2021. https://iohk.io/en/blog/posts/2021/09/10/concurrency-and-all-that-cardano-smart-contracts-and-the-eutxo-model/.

[5] Sundaeswap labs presents: The scooper model, Nov 2021. https://sundaeswap-finance.medium.com/sundaeswap-labs-presents-the-scooper-model-678d6054318d.

[6] Blockchain, Mar 2022. https://en.wikipedia.org/wiki/Blockchain.

[7] Cardano: Protocol parameters, Apr 2022. https://developers.cardano.org/docs/governance/cardano-improvement-proposals/cip-0009/#updatable-protocol-parameters.

[8] Central african republic passes bill to make bitcoin legal tender, Apr 2022. https://www.cnet.com/personal-finance/crypto/central-african-republic-passes-bill-to-make-bitcoin-legal-tender/.

[9] Charli3: Cardano's decentralized oracle, Mar 2022. https://charli3.io/.

[10] Concurrency and cardano: A problem, a challenge, or nothing to worry about?, Jan 2022. https://builtoncardano.com/blog/concurrency-and-cardano-a-problem-a-challenge-or-nothing-to-worry-about.

[11] Ethereum whitepaper, Apr 2022. https://ethereum.org/en/whitepaper/.

[12] Learn about plutus, Mar 2022. `https://docs.cardano.org/plutus/learn-about-plutus`.

[13] Oracles, Jan 2022. `https://ethereum.org/en/developers/docs/oracles/`.

[14] Orcfax: Truly trustworthy cardano oracle, Mar 2022. `https://www.orcfax.link/`.

[15] Plutus pioneer program - 5. week 05 - native tokens, Mar 2022. `https://plutus-pioneer-program.readthedocs.io/en/latest/pioneer/week5.html`.

[16] Today's cryptocurrency prices by market cap, Apr 2022. `https://coinmarketcap.com/`.

[17] Understanding the extended utxo model, Mar 2022. `https://docs.cardano.org/plutus/eutxo-explainer`.

[18] Uniswap: A brief history, Feb 2022. `https://blog.cryptostars.is/uniswap-a-brief-history-fe8937a6bbdc`.

[19] Wingriders: a decentralized exchange on top of cardano eutxo mode. 2022. `https://assets.wingriders.com/whitepaper.pdf`.

[20] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core, 2021.

[21] Imran Bashir. *Mastering Blockchain.* 2017.

[22] Alex Biryukov and Sergei Tikhomirov. Deanonymization and linkability of cryptocurrency transactions based on network analysis. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 172–184, 2019.

[23] Lars Brünjes and Murdoch J. Gabbay. Utxo- vs account-based smart contract blockchain programming paradigms. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 73–88, Cham, 2020. Springer International Publishing.

[24] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Muller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. Utxoma: Utxo with multi-asset support. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 112–130, Cham, 2020. Springer International Publishing.

[25] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: Front-running attacks on blockchain. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 170–189, Cham, 2020. Springer International Publishing.

[26] Christoph Jentzsch. Decentralized autonomous organization to automate governance. *White paper, November*, 2016.

[27] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing.

[28] Maladex. Research-driven cardano dex white paper v1, Oct 2021. `https://docs.maladex.com/whitepaper.pdf`.

[29] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges, 2021.