GiG: A Decentralized Platform for the Gig Economy

Ishtar Eve

February 2019

Abstract

A decentralized application for the labor marketplace that connects employers with independent contractors is described. GiG is intended to reduce friction and to eliminate fees collected by employment agencies, recruiting platforms, and financial institutions. Common bugs will be avoided by using Plutus, the strongly typed functional programming language of the Cardano blockchain.

The native currency of the system is the Gig Economy Token (GET). The algorithm used for its creation prevents speculative bubbles, and funds a Treasury System DAO (Decentralized Autonomous Organization). The DAO finances expenses for the ecosystem through proposals created and voted by the GiG community.

GiG offers multiple tools that will give value to freelancers and employers, gaining the ability to interact without the need of trust, transacting on a peer to peer basis (without using financial institutions).

(Specific information about GiG functionalities on this section using bulletpoints).

Note: GiG is under construction. New versions of this paper will be updated periodically.

1 Introduction

(Current state of the gig economy marketplace, problems that the GiG protocol solves.)

A study by Intuit predicts that the gig economy will be about 43% of the workforce by 2020.

(Paper organization)

(Thoughts)

All around the world, there's lots of work that needs to be done, and many people that need to work. Unfortunately, the people who are willing to pay a certain amount of money for a certain work to be done, very often can't connect with the people who would benefit from the working opportunity and who would be willing to do the work in exchange for such amount.

2 GET Token Creation

GET tokens are created when ADA is received by the GET creation smart contract (GCSM). The amount of GET created and awarded to the ADA sender's wallet α, will be an arbitrary constant ϕ (currently set to 1000 GET), divided by the block height ℓ, starting from the first block since the GET creation smart contract gets published. That is,

$$\alpha = \frac{\phi}{\ell}$$

This policy has several effects:

1. It provides an incentive for adopting quickly the GiG Economy Token.
2. After enough time, the conversion rate plateaus, thus providing a stable coin.
3. Creating new tokens costs more over time, thus putting a positive price pressure on the existing tokens.
4. After enough time, creating new tokens is significantly expensive, thus helping the Gig Economy because working for tokens is less expensive than working for ADA, and then exchanging that ADA for GET.

3 Treasury System and DAO funding

All the ADA received by the GCSM is managed by a treasury system DAO based on the research made by IOHK for the Zendao [1]. The objective of this treasury system is the funding and administration of the development of the GiG Economy Token platform.

4 Creating Job Offers

An employer signs a job offer posting transaction with his Cardano private key. This transaction will always include an amount of get offered for the task, and a description of said task. The transaction might also include other information about the task, such as a location list, a expected schedule, and other instructions or information about the task.

This is implemented by posting a DataScript transaction to a Smart Contract that we decided to call the "Job Board". This initial transaction may have no ADA associated, and it's intended to broadcast to the community that a new job is available. It has to include enough information for the freelancers to decide if it's interesting to them, and this is implemented by using a DataScript with fields to describe the job being offered, and includes the public key of the employer.

The first iteration of the protocol contains just enough information to prove communication can be achieved, and may be extended later with fields of interest.

```
data JobOffer = JobOffer
  { joDescription :: ByteString
  , joPayout      :: Int
  , joOfferer     :: PubKey
  }
  deriving (Show, Eq, Ord, Generic)
PlutusTx.makeLift ''JobOffer
```

5 Applying for Job Offers

In order for a freelancer to find jobs, first he must be checking the place where employers post jobs: the Job Board. Thus, the first step for a freelancer is start listening for transactions in the Job Board address. As employers post new transactions there, the freelancers will be notified by their wallets of these operations. Then, the freelancers must be able to parse back the descriptions. The system uses the Plutus programming language for smart contracts, and this language, when compiled to the Plutus bytecode (known as CEK code), is not readable by human beings. In order for a freelancer to read and understand a job offer, we need to offer him a human-readable version of the posted job. This is done by means of the mechanism described in "A pub-sub mechanism for Cardano and Plutus"[2]. Once this Plutus bytecode is parsed back to a standard machine representation, it's easy for the system to convert this machine representation into a user interface for the freelancer.

Now that the freelancer can understand what the job demands, he can apply to it. In this context, application means that the freelancer considers himself good enough to deliver the job required, but the employer may disagree. This is why further filtering before the job is done and the payment is delivered is needed. In this case, the freelancer posts his application to a different address, that we call the "Job Application Board". This address is derived automatically from the main Job Board address, hashing into it the specific details of the job.

The freelancer posts to this Job Application Board his public key, and possibly some relevant information. For the first iteration of the protocol, we have decided that the freelancer will only post his public key.

```
data JobApplication = JobApplication
```

```
  { jaAcceptor    :: PubKey
  }
  deriving (Show, Eq, Generic)
PlutusTx.makeLift ''JobApplication
```

6 Accepting Applications

At this point, both the employer and the freelancer share some common agreement (the job contract), and both parties have copies of the public keys of the other. They can now use standard asymmetric-cipher communication to further discuss the job terms and the ability of the freelancer to deliver the job. This is not covered by this paper, but it's relevant for the mechanism to be complete, and therefore will be addressed in further publications.

After the employer and the freelancer have reached an agreement, it is signed by the employer creating an escrow and locking the funds for payment. At this point, the freelancer can see the employer willingness to pay for the job by observing the escrow with the associated funds on the blockchain.

The underlying datatype in a DataScript that configures an escrow looks like:

```
data EscrowSetup = EscrowSetup
  { esJobOffer        :: JobOffer
  , esJobApplication  :: JobApplication
  , esArbiter         :: PubKey
  }
  deriving (Show, Eq, Generic)
PlutusTx.makeLift ''EscrowSetup
```

At this point, we have introduced another stakeholder without describing his role: the arbiter. The arbiter is a fundamental piece of the dispute resolution process that is described a few sections below.

7 Escrow Release

The freelancer now will proceed to deliver the work. When the employer agrees that the work has been delivered, he can release the funds in the escrow to the freelancer by posting a transaction to the escrow smart contract attaching his order to release as a RedeemerScript. This transaction is signed with the employer's private key, and assigns the funds to the freelancer's wallet.

```
data EscrowResult
  = EscrowAcceptedByEmployer Signature
  ...
  deriving (Show, Eq, Generic)
PlutusTx.makeLift ''EscrowResult
```

8 Dispute Resolution

Not every time the freelancer and the employer will agree on the outcome of the work delivered. When this happens, the employer may choose to not release the funds, which puts the situation in a stalemate, as the freelancer may not want to work more, and the employer has the funds locked in the smart contract.

To break this stalemate we introduce a third party: an arbiter. The arbiter has been agreed upon before the escrow creation, and his public key is now part of the escrow DataScript. The arbiter has power to release the funds to the freelancer, or back to the employer, but not to himself. It is expected for the arbiter to review any evidence provided by the employer and the employee, and release the escrow in favor of the actor whom he deems is in the right.

The escrow is released by the arbiter by posting a transaction to the escrow address with the result of his judgment (accept escrow to freelancer, or reject escrow back to employer), written as a RedeemerScript, setting the output of the transaction to the wallet of the freelancer or the wallet of the employer, and finally signing the transaction with his private key.

```
data EscrowResult
  = EscrowAcceptedByEmployer Signature
  | EscrowAcceptedByArbiter Signature
  | EscrowRejectedByArbiter Signature
  deriving (Show, Eq, Generic)
PlutusTx.makeLift ''EscrowResult
```

9 Smart contracts

Therefore, the system uses three main smart contracts: one for broadcasting the job, another for freelancers to apply to job offers, the final one for escrow and dispute resolution.

9.1 Job Board Smart Contract

The Job Board Smart Contract is somewhat simple, as it doesn't have to do advanced data processing and validation of transactions. Its main objective is forwarding information from the employer to the freelance, and back. In order to signal that a job is no longer available (for example, because it has been performed and is no longer needed), the employer is offered the option of closing it. This becomes the only validation this smart contract will do: whoever posts to the Job Board is the only one that can take it down.

The system uses the Watched Addresses abstraction provided by Plutus. This abstraction provides a way to specify which addresses (such as the Job Board and the Job Application Board) we want to observe, and the system will tell us non-complete transactions that are currently there. Because we use this system to read back job offers, we can take down a job offer by completing

the transaction that posted the job offer, and thus making it disappear from everyone's Watched Addresses.

```
jobBoard :: ValidatorScript
jobBoard = ValidatorScript ($$(Ledger.compileScript [||
  \() (JobOffer {joOfferer}) (t :: Validation.PendingTx) ->
    let
        Validation.PendingTx {
          pendingTxInputs=[_],
          pendingTxOutputs=[
            Validation.PendingTxOut {
              pendingTxOutData=Validation.PubKeyTxOut pubkey
            }
          ]
        } = t  -- It's fine if this fails matching,
               -- as it will cause the validator to error out and reject the transaction.

        inSignerIsSameAsOutSigner = $$(Validation.eqPubKey) pubkey joOfferer

    in
    if inSignerIsSameAsOutSigner
    then ()
    else $$(PlutusTx.error) ()
  ||]))
```

The Job Board Smart Contract demands that whoever posted the job must take it down, and this is the only validation implemented in this contract. This allows the poster to notify the world when the job is available, and when it's no longer available, while preventing malicious third parties from closing the job contract without the employer's approval.

9.2 Job Application Board Smart Contract

Once a job has been published to the Job Board, we can derive a second address: the Job Application board address. This is done by using one of Plutus mechanisms that allow to use plain old currying/uncurrying to apply one of the parameters of a multi-parameter function and get as result a unique version of that function with that parameter applied.

```
jobAcceptanceBoard :: ValidatorScript
jobAcceptanceBoard = ValidatorScript ($$(Ledger.compileScript [||
  \(_ :: JobOffer) () (_ :: JobApplication) (_ :: Validation.PendingTx) ->
    -- TODO: Implement some validation here
    ()
  ||]))

jobAddress :: JobOffer -> Address
```

```
jobAddress jobOffer = Ledger.scriptAddress (ValidatorScript sc)
  where
    sc = (getValidator jobAcceptanceBoard) `applyScript` (Ledger.lifted jobOffer)
```

We derive the Job Acceptance Board address by applying the `JobOffer` to the `jobAcceptanceBoard` validator, which constructs a new validator specific to our Job. This assumes no jobs will ever be repeated, which is something that may have to change in the future.

The employer is expected to be listening on the corresponding Job Acceptance Board addresses for the jobs he has posted, with the intention of getting a feed of applicants to his job openings. The freelancers are expected to post their contact information as a transaction, with a `JobApplication` `DataScript` to the Job Acceptance Board, without associated payment. This way, they can communicate their interest in the job offer and forward contact information.

9.3 Escrow Smart Contract

The Escrow Smart Contract is significantly more complex than the Job Board and the Job Application Board Smart Contracts for two reasons:

- The Escrow has funds that malicious actors may seek to steal.
- The Escrow may have to release these funds to different parties.

For this reason, the Escrow must have much tighter security and validation.

As usual, every Plutus Smart Contract is configured with a DataScript. In the Escrow DataScript we introduce the full job description (including the public key of the employer), as well as the description of the accepted freelance (including his public key), and the public key of the chosen arbiter.

```
jobEscrowContract :: ValidatorScript
jobEscrowContract = ValidatorScript ($$(Ledger.compileScript [||
  \ (result :: EscrowResult)
    (setup :: EscrowSetup)
    (tx :: Validation.PendingTx)
    ->
    let EscrowSetup {
          esJobOffer=JobOffer {
            joOfferer=employerPubKey
          },
          esJobApplication=JobApplication {
            jaAcceptor=employeePubKey
          },
          esArbiter=arbiterPubKey
        } = setup
    in
...
  ||]))
```

The Escrow Smart Contract will now validate the actions that try to spend the escrow, and will check that the signer of the action is in position to execute the action, by checking the signatures of these actions.

```
jobEscrowContract :: ValidatorScript
jobEscrowContract = ValidatorScript ($$(Ledger.compileScript [||
  \ (result :: EscrowResult)
    (setup :: EscrowSetup)
    (tx :: Validation.PendingTx)
    ->
...
    let
      eqPubKey = $$(Validation.eqPubKey)
      signedBy' (Signature sig) (PubKey pk) = ...
      (&&) = $$(PlutusTx.and)
    in
    case result of
      EscrowAcceptedByEmployer sig ->
        if (signedBy' sig employerPubKey) && (eqPubKey destPubkey employeePubKey)
        then ()
        else $$(PlutusTx.traceH) "Bad acceptance by employer" ($$(PlutusTx.error) ())
      EscrowAcceptedByArbiter sig ->
        if (signedBy' sig arbiterPubKey) && (eqPubKey destPubkey employeePubKey)
        then ()
        else $$(PlutusTx.traceH) "Bad acceptance by arbiter" ($$(PlutusTx.error) ())
      EscrowRejectedByArbiter sig ->
        if (signedBy' sig arbiterPubKey) && (eqPubKey destPubkey employerPubKey)
        then ()
        else $$(PlutusTx.traceH) "Bad reject by arbiter" ($$(PlutusTx.error) ())
  ||]))
```

10 Prototype application

At the time of writing this document, Plutus is not yet available in a testnet, and whoever want to try it have to use it through the Plutus Playground, or through the underlying emulator the Plutus Playground uses: the Mockchain.

10.1 Beyond the Plutus Playground

The Plutus Playground offers a nice and easy to use interface for interacting with the Mockchain, but it is severely limited, specially in the terms we are going to use.

- First of all, the Plutus Playground offers us an interface we may or may not agree to use, but, in any case, is not the interface we expect from the

system. We want the ability to use our own interface, in order to apply the UI/UX we consider adequate for the project.

- Second, the Plutus Playground doesn't allow us to do advanced manipulation of smart contracts, such as reading and parsing back DataScripts.
- Finally, the Plutus Playground doesn't have the ability to save files to a version control system that all modern software development workflow expects.

For these three reasons, we had to discard the Plutus Playground soon, and jump quickly to directly use the Mockchain in a prototype. This allowed us to understand deeply Plutus, and estimate real costs of integration with a real blockchain.

10.2 Architectural considerations of the prototype

The tech stack we chose for the prototype is a conservative choice given the most significant constraints in the system:

- Plutus is written in Haskell, therefore the prime candidate programming language would be Haskell.
- We have some experience in web development, web development offers somewhat easy user interfaces with the usual entities (buttons, text, actions), and creating a web application allows us to offer the system easily over the Internet by just providing an URL. Therefore, constructing the prototype as a web application is our prime architectural decision.
- Although the blockchain is distributed in its nature, there is a *centralized* concept behind it, being this concept the *global agreement on a single chain*. We can simulate this global agreement locally without the problems that a distributed system creates by having a single Mockchain stored in a single machine. Although this defeats the distributed purpose of a real blockchain, we consider it good enough for a prototype phase, keeping in mind that a real blockchain with Plutus doesn't exist yet.

Following these three reasons, we have used as a base for our prototype a simple Yesod scaffolding with SQLite as database. Among the many web frameworks available, we chose this one because:

- It is a web framework based in Haskell, therefore integrating Plutus should not be significantly hard.
- It is an *opinionated* web framework. Opinionated web frameworks are the ones that have opinions: they have an opinion that you will use an specific architectural pattern, that you will use an specific template system, that you will use an specific data storage. In contrast, *unopinionated* web

frameworks feature an architecture where the developers have to connect whatever they need for the situation. In our experience, opinionated web frameworks work well for the large majority of projects, because their *opinions* are conservative chosen, and a lot of integration effort has been already spent making their opinions fit nicely. Therefore, we chose Yesod in the Haskell world, because it is the opinionated web framework that mostly resembled the *King of web frameworks*: Ruby on Rails.

- It is well supported by the community. This is important, as it will save time because there will be available plugins for most situations, and books, forums and blogs to help unblock the development team when they need it.

10.3 Design approach of the prototype

Now that we decided to use a web application, we can start to consider what this means in terms of it being a frontend for Plutus and the Mochckain:

- We are going to store a single Mockchain in memory, and use it as our datastore.
- Actions performed by the different stakeholders of the system will be performed against this single in-memory Mockchain, usually by adding new blocks to it.
- We should try to use only the methods and operations that the Wallet API provides, as any other operation is not expected to be available on the real blockchain.
- If we have to use operations not provided by the Wallet API, we have to be careful to ensure these operations are plausible and somewhat expected to be available later. In any case, we can't do operations that break the constraints of the blockchain, such as rewriting blocks.

These constraints have proven to be a reasonable challenge, and have pushed our learning on how to construct distributed apps significantly. The only constraint we have broke regularly is the ability to rewrite the blockchain. By virtue of storing the Mockchain in memory, every time we restarted the system it forgot its history and restarted in a clean state. This has proven very useful for testing and validating ideas, and is expected on usual development workflows, but it's also something to keep in mind, as doing it in production systems is expected to be almost impossible.

10.4 Lessons learnt from the prototype

In our opinion, the usage of a web framework has paid significantly, as it provided a simple but powerful way to explore the user interfaces around Smart Contracts. We had to figure out when we have to offer different buttons for different actions,

guided by the available information on the blockchain. This also lead us to interesting information distribution issues, that eventually lead to the discovery of unlifting and the Plutus Pub-Sub pattern[2].

Ironically, the *opinion* of using a SQLite database has not been used, as we focused on using the blockchain as our distributed store of data. Storing data in the blockchain is expected to be expensive[3], so we can't truly discard the option of using an external and cheaper data storage.

Creating
Geolocation
DAO funding
Creating Job Offers
Escrow Creation
Booking inquiries Booking a Freelancer

Making a transaction Management Tools Automatic Timesheets Communication Between Parties Job Completion

References

[1] A Treasury System for Cryptocurrencies:Enabling Better Collaborative Intelligence Bingsheng Zhang1, Roman Oliynykov2, and Hamed Balogun3

[2] A pub-sub mechanism for Cardano and Plutus, `https://github.com/javcasas/plutus-pubsub-paper/blob/master/paper.pdf`, Javier Casas, March 2019.

[3] Transaction fees in Cardano SL, `https://cardanodocs.com/cardano/transaction-fees/`, March 2019.