A pub-sub mechanism for Cardano and Plutus

Javier Casas Velasco

March 2019

Abstract

On this paper we demonstrate how we can use DataScripts to communicate different programs using the Cardano Blockchain and the Plutus Smart Contract programming language. We construct a mechanism that works in a way akin to a pub-sub communication system. This mechanism has been developed during the development of the GiG Economy Token[1].

Contents

1 Introduction

Blockchains provide a new way to intercommunicate different actors in the world. The blockchains provide a solution for the old problem of getting to cooperate different actors that may not want to cooperate. In this sense, the main use of the Blockchain has been so far as a ledger that is mostly handled manually. When two actors want to communicate, they use manual approaches (such as chats, videoconferences, or plain old phone calls) before actually using the Blockchain, in most cases as a ledger for payment.

In order to be able to automate some of this manual communication, an automatic way of communicating these actors shall be developed, so that the actors can communicate without resorting to manual ways. The GiG Economy Token[1] uses this mechanism to communicate freelancers and employers prior to actually dealing with the payment details.

1.1 The Publisher-Subscriber (pub-sub) pattern

One of the main building blocks of distributed process communication is the Publisher-Subscriber (also known as pub-sub) pattern. It allows one or many broadcaster processes to send messages to one or many receiver processes. Many other patterns of distributed systems can be constructed from it. For example, client-server communication between two processes can be achieved by having two pub-sub patterns, one for the client to send the queries to the server, the other for the server to send the responses back to the client.

The main roles in the pub-sub pattern are:

Publisher The publisher sends data messages to the channel. There may be many publishers broadcasting messages at the same time.

Subscriber The subscriber receives messages from the channel, and acts on them. There may be many subscribers receiving messages from the channel, and each subscriber receives all the messages.

Channel A channel is an abstraction that allows publishers to send messages to all the subscribers.

Message A message is an arbitrary piece of data that can be encoded and sent in a channel to be received and decoded by the subscribers listening on the channel.

Broadcasting The pub-sub pattern uses broadcasting as the single way of communication. A copy of each message sent to the channel is delivered to each subscriber.

The main difference between pub-sub and other many-to-many communication patterns is that pub-sub delivers all the messages to all the subscribers,

whereas other patterns may decide to route the messages to a restricted group of receivers based on an arbitrary policy.

1.2 The Cardano blockchain

Cardano is a blockchain based on the Outboros Proof of Stake[2]. It provides a model for universal transactions called UTxO. The Outboros Proof of Stake protocol provides protection against many of the usual attacks against blockchains, while being a Proof of Stake protocol, and thus requiring very little computational power in comparison with Proof of Work blockchains. On top of the usual cryptocurrency operations, Cardano seeks to provide other services, including Smart Contracts.

1.3 Plutus Smart Contracts in the Cardano blockchain

A Smart Contract is a transaction in a blockchain governed by an arbitrary program. This is different from the usual transactions, that are usually governed solely by one or several cryptographic signature.

The Smart Contracts for the Cardano blockchain will be written in a language called Plutus, and based on Haskell. At the time of writing this paper, Plutus is in heavy evolution, and lacks yet a testing network. The only way to execute Plutus smart contracts is through the Plutus Emulator[3], or through the Plutus Playground[4], which uses the emulator under the hood.

The way Plutus Smart Contracts are implemented is quite simple, but requires some understanding:

- A stakeholder pays an amount to the address of a smart contract (which is the hash of the smart contract). In this payment, the stakeholder provides a DataScript: a piece of data in the Plutus format that can be used to configure the smart contract.
- A second stakeholder wants to spend the amount the first stakeholder paid to the smart contract address. To do so, he has to provide a smart contract which hash coincides with the first address, and a second piece of data called the RedeemerScript, and, when executing the smart contract code with the DataScript, the RedeemerScript and the transaction as parameters, it has to terminate without any errors. Then the transaction is authorised.

This implementation provides several interesting approaches that we are going to use to construct our pub-sub mechanism.

2 Publishing and receiving messages

In order to create a pub-sub mechanism, we need a way to write messages to the Cardano blockchain, and a mechanism to recover messages from the blockchain.

The blockchain is public, and we will use this property to derive the broadcasting property of pub-sub from it.

2.1 Publishing messages to the blockchain

First, we need to publish messages to the blockchain. For this, we will use smart contracts, but not in a complete way. We will encode our messages in DataScripts, and will publish them to the blockchain by paying to the address of a Smart Contract with the DataScript as attached data.

For this, we can use the standard mechanisms that Plutus provides:

- `Ledger.Types.lifted` converts a bunch of data to Plutus format.
- `Wallet.API.payToScript` pays an amount of Ada to the address of a Smart contract with the DataScript provided.

A usual implementation looks like this:

```
payToContract :: (WalletAPI m, WalletDiagnostics m) => Value -> m ()
payToContract val = payToScript_ defaultSlotRange contractAddress val ds
  where
    ds = DataScript (Ledger.lifted contractData)
    contractData = ...
```

2.2 Receiving messages from the blockchain

The Wallet API supports a wallet listening for activity in arbitrary addresses. This is intended for the wallet to be notified when it receives a transaction, or when activity occurs at a smart contract of interest. This mechanism is based on two Wallet API endpoints:

- `Wallet.API.startWatching` starts watching the specified address, and adds any transactions that happen on it to the dictionary of watched addresses.
- `Wallet.API.watchedAddresses` returns the dictionary of watched addresses, that includes all the associated transactions for all the watched addresses, including DataScripts for associated payToScript transactions.

This means we can recover data from the blockchain, but in the format of Plutus smart contracts. The messages will be accumulated on the `AddressMap`[7] in arbitrary order, and this is something we have to keep in mind when constructing our systems.

3 Converting arbitrary data from and to DataScripts

The blockchain will accept and return data in the Plutus format, but we want to process it as Haskell data, so our standard Haskell applications can use it.

3.1 Converting arbitrary data into DataScripts

Converting arbitrary data to a DataScript is well supported in the system. As stated previously,

- `Ledger.Types.lifted` converts a bunch of data to Plutus format.

The only requirement is that the data has a `Language.PlutusTx.Lift.Class.Lift` instance, which can be manufactured automatically by using the `Language.PlutusTx.makeLift` Template Haskell incantation.

```
module MyModule where

import qualified Language.PlutusTx            as PlutusTx
import           Ledger                       (Slot(..), PubKey)
import           Ledger.Ada.TH                (Ada)

-- Declare our datatype
data MyDatatype = MyDatatype
    { foo :: Slot
    , bar :: Int
    , baz :: Ada
    }

-- Create a Language.PlutusTx.Lift.Class.Lift instance for it
PlutusTx.makeLift ''MyDatatype
```

3.2 Recovering arbitrary data from DataScripts

Lifting data to the Plutus format is as easy as using the `Ledger.Types.lifted` function. But the way back is not that easy. To be able to achieve this, we have to figure out the Plutus format, and a way to interpret Haskell datatypes out of it.

So let's start with an in-depth overview of Plutus.

3.2.1 Core Plutus types

From the Plutus core types[5], let's get to understand the four builtin types:

```
-- | A constant value.
data Constant a = BuiltinInt a Natural Integer
                | BuiltinBS a Natural BSL.ByteString
                | BuiltinSize a Natural
                | BuiltinStr a String
        deriving (Functor, Show, Eq, Generic, NFData, Lift)
```

There are four builtin types: `Integer`, `ByteString`, `Natural` and `String`.

Just after the previous code[6], we can see the different values that can be part of a term:

```
data Term tyname name a
  = Var a (name a) -- ^ A named variable
  | TyAbs a (tyname a) (Kind a) (Term tyname name a)
  | LamAbs a (name a) (Type tyname a) (Term tyname name a)
  | Apply a (Term tyname name a) (Term tyname name a)
  | Constant a (Constant a) -- ^ A constant term
  | Builtin a (Builtin a)
  | TyInst a (Term tyname name a) (Type tyname a)
  | Unwrap a (Term tyname name a)
  | IWrap a (Type tyname a) (Type tyname a) (Term tyname name a)
  | Error a (Type tyname a)
deriving (Functor, Show, Generic, NFData, Lift)
```

Following from these two pieces of code, we can get the final form of a builtin value in Plutus:

- `Constant _ (BuiltinInt _ _ i)`
- `Constant _ (BuiltinBS _ _ bs)`
- `Constant _ (BuiltinSize _ sz)`
- `Constant _ (BuiltinStr _ str)`

For example, the Plutus program `$$(Ledger.compileScript [|| 3 ||])` will be converted to `Constant _ (BuiltinInt _ _ 3)`.

3.2.2 Core Plutus language

Plutus is based on Lambda Calculus. It looks and behaves in a way similar to Haskell, except it's strict. We will use this to extract the data that interests us.

3.2.3 Evaluating Plutus programs

We know how the programs will look like after they have been fully evaluated, but we haven't defined how evaluation will happen. As expected, Plutus provides a way to evaluate programs that is intended mainly to be used on the emulator to see if a program errors out or not, but we will use this mechanism in a different way.

The main function is located at `Language.PlutusTx.Evaluation.evaluateCekTrace`.

```
evaluateCekTrace
    :: Program TyName Name ()
    -> ([String], EvaluationResultDef)
```

When running a Plutus program through this evaluator function, it will generate a result and a bunch of error messages. We are interested in the `EvaluationResultDef`:

```
type EvaluationResultDef = EvaluationResult (Value TyName Name ())
data EvaluationResult a
    = EvaluationSuccess a
    | EvaluationFailure
    deriving (Show, Eq, Functor, Foldable, Traversable)
```

Now it is obvious how we can distinguish a success from a failure, but what is that `(Value TyName Name ())` ? A bit of looking around finally completes almost the whole piece of the puzzle:

```
newtype Script = Script { getPlc :: PLC.Program PLC.TyName PLC.Name () }
newtype ValidatorScript = ValidatorScript { getValidator :: Script }
newtype DataScript = DataScript { getDataScript :: Script }
newtype RedeemerScript = RedeemerScript { getRedeemer :: Script }
```

The Validators, the Data Scripts, and the Redeemer Scripts are nothing more than Scripts, which are nothing more than `Program TyName Name ()`, that when evaluated, produce an `EvaluationResult` that may contain a `Value TyName Name ()`. Finally, we pattern match on the `Value TyName Name ()`, and extract the corresponding constructor of `Constant`.

3.2.4 Extracting fields from complex datatypes

The previous schema allows us to extract Plutus core datatypes, as long as they are standalone. But, what if the value is part of a composite datatype? After all, we all happily use ADTs in Plutus, and would be a bad surprise if we can't extract values from them.

The answer lies again in Lambda Calculus in the Haskell style. Let's have a small example:

```
data Foo = Foo
          { fooBar :: Int
          , fooBaz :: ByteString
          }
          | Bar
          { barBar :: Int
          }
PlutusTx.makeLift ''Future
```

If we could extract any of the fields, we could use our evaluation mechanism to figure out what is the actual value by pattern matching on the result. In Haskell we would happily do it by using the accessor functions fooBar, fooBaz and barBar. These functions will not work in Plutus, and, helpfully, Plutus suggests us to use pattern matching. Let's construct a few Plutus functions to extract the values:

```
fooBarAccessor = $$(Ledger.compileScript [|| \(Foo fooBar _) -> fooBar ||])
fooBazAccessor = $$(Ledger.compileScript [|| \(Foo _ fooBaz) -> fooBaz ||])
barBarAccessor = $$(Ledger.compileScript [|| \(Bar barBar) -> barBar ||])
```

Now, by using plain old Lambda calculus, we should just be able to apply an accessor to a value of Foo and get either one of the fields out, or an error. To do this, we need another function provided by Plutus:

```
applyScript :: Script -> Script -> Script
```

Essentially, applyScript takes an `f` and an `a`, and returns `f a`. Which then we run through `evaluateCekTrace` to get a value or an error out. And now we are almost ready to complete the way back to Haskell from Plutus. We start with a bunch of functions for extracting the core values:

```
getInt :: (a, EvaluationResult (Term b c d)) -> Maybe Integer
getInt (_, EvaluationSuccess (Constant _ (BuiltinInt _ _ x))) = Just x
getInt _ = Nothing

getBS :: (a, EvaluationResult (Term b c d)) -> Maybe ByteString
getBS = ...

getString :: (a, EvaluationResult (Term b c d)) -> Maybe String
getString = ...
```

These functions can't just return the datatype, instead they have to offer it as a `Maybe`. After all, nothing stops us from trying to extract values from datatypes that don't have such values, and the only way to allow that to work properly is by offering some kind of signalling for failure, for example, `Maybe`.

And we continue with functions for extracting specific fields in our `Foo` datastructure:

```
getFooBar ::
  DataScript ->
  Maybe Int
getFooBar datascript = getInt r
  where
    r = evaluateCekTrace (unsafeCoerce (accessor `applyScript` datascript)
    accessor = fooBarAccessor

getFooBaz :: DataScript -> Maybe ByteString
getFooBaz = ...

getBarBar :: DataScript -> Maybe Int
getBarBar = ...
```

Finally, we take all the getters and make use of old Applicative/Alternative machinery to construct our `Foo` back:

```
getFoo ::
  DataScript ->
  Maybe Foo
```

```
getFoo datascript =
    (Foo <$> getFooBar datascript <*> getFooBaz datascript)
    <|>
    (Bar <$> getBarBar datascript)
```

4 The plutus-unlift library

All the code to extract a value from a DataScript is quite mechanical, and can be streamlined. For this, we have constructed a small library, called `plutus-unlift`, that is intended to simplify this process.

With the intention of supporting extracting arbitrarily nested datastructures, we define a typeclass, called `Unlift`, to simplify the process:

```
class Unlift a where
  unlift :: DataScript -> Maybe a
```

And we support this typeclass with some instances for basic cases:

```
instance Unlift ByteString where
  unlift (DataScript ds) = getBS $ evaluateCekTrace (unsafeCoerce ds)

instance Unlift Integer where
  unlift (DataScript ds) = getInt $ evaluateCekTrace (unsafeCoerce ds)

instance Unlift Int where
  unlift (DataScript ds) = fromIntegral <$> i
    where
      i = getInt $ evaluateCekTrace (unsafeCoerce ds)
```

Newtype instances are not hard to construct:

```
instance Unlift PubKey where
  unlift (DataScript ds) = PubKey <$> pk
    where
      pk = unlift (DataScript (extractor `applyScript` ds))
      extractor = $$(Ledger.compileScript [|| \(PubKey x) -> x ||])
```

And, finally, instances for arbitrary datatypes are also not hard to construct:

```
instance Unlift Foo where
  unlift (DataScript ds) =
    (
      Foo
      <$> unlift (DataScript
          ($$(Ledger.compileScript [|| \(Foo a _) -> a ||]) `applyScript` ds))
      <*> unlift (DataScript
```

```
        ($$(Ledger.compileScript [|| \(Foo _ b) -> b ||]) `applyScript` ds))
  ) <|> (
    Bar
    <$> unlift (DataScript
        ($$(Ledger.compileScript [|| \(Bar a) -> a ||]) `applyScript` ds))
  )
```

5 Conclusions

On this paper we have constructed a general way for publishing arbitrary data to the Cardano blockchain in the form of DataScripts, and a way for extracting back our original data from the DataScripts. With this mechanism available, lots of communication systems using Cardano and Plutus Smart Contracts are viable.

References

[1] GiG Economy Token organization at GitHub, https://github.com/gig-economy-token.

[2] Outboros Proof of Stake, https://cardanodocs.com/cardano/proof-of-stake/.

[3] Plutus Emulator, https://github.com/input-output-hk/plutus/tree/master/wallet-api/src/Wallet/Emulator.

[4] Plutus Playground, https://prod.playground.plutus.iohkdev.io/.

[5] Plutus Core Constant datatype, https://github.com/input-output-hk/plutus/blob/a04c5681c695c85035958c79bae46130cdd9778e/language-plutus-core/src/Language/PlutusCore/Type.hs#L235.

[6] Plutus Core Term datatype, https://github.com/input-output-hk/plutus/blob/a04c5681c695c85035958c79bae46130cdd9778e/language-plutus-core/src/Language/PlutusCore/Type.hs#L244.

[7] Wallet API Address Map, https://github.com/input-output-hk/plutus/blob/a04c5681c695c85035958c79bae46130cdd9778e/wallet-api/src/Wallet/Emulator/AddressMap.hs#L39.