A Study of Oracle Systems for the QTUM Blockchain Eco-system

By

Mr. George John Chavady

A Dissertation

submitted to the University of Dublin, in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Prof. Donal O' Mahony

August 2020

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work, and has not been submitted as an exercise for a degree at this or any other university.

Signed: [redacted]

George John Chavady

26/08/2020

Permission to lend and/or copy

I agree that the Trinity College Library may lend or copy this dissertation upon request.

Signed: [redacted]

George John Chavady

26/08/2020

Acknowledgments

First and foremost, I would like to thank God Almighty for helping me complete my research. Next, I would like to express my sincere gratitude to my supervisor, Prof. Donal O'Mahony for his continuous support, guidance, encouragement and expertise during my master's thesis.

I also thank my parents Mr. John Cherian and Mrs. Aneyamma John, for all their support and love during the period of my course. I am very grateful to Mr. Manoj George, my uncle for his support during my stay in Ireland. My brother Paul John and Elizabeth John have always been a pillar of support.

George John Chavady

University of Dublin, Trinity College

August 2020

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The world presents to us a number of use cases that require decentralization. For example, consider a travel insurance policy that provides a cover for travel delay. The data critical to verify whether a flight has been delayed, is the scheduled departure time and actual departure time. The process of claims payment can be digitised using a software application. But the application could be easily manipulated by a person with access permissions, either by altering the code or the data used by the claims payment system, and hence prone to fraud. In our case the variable 'Actual Departure Time' can be altered to make a false payment. The solution is to implement a decentralized and immutable system where all the state changes are stored permanently, and no single person has complete authority over the system.

Blockchain is a technology that combines the qualities of decentralization and immutability. It achieves decentralization using a peer-to-peer network to connect all the users of the system, and each member of the network equally participates to reach a consensus. Each communication in the network is recorded as a transaction and stored permanently in the network. It uses various types of consensus protocols to reach an agreement on whether a transaction is valid. There is no single entity that fully controls the operation in the blockchain. The cryptography techniques and incentive schemes are carefully designed to help create a distributed ledger shared by all the members of the network and is immutable.

By design, the blockchain network has no central authority with the ability to unilaterally approve invalid transactions or roll back and alter the state of previous transactions. Although, there are some blockchains that allow only one user to mine blocks, which are called private blockchains. The distributed ledger stored on every user's machine is an append-only log for storing data in an immutable manner. A new transaction is only added at the end of the ledger, and no previous transactions can be modified. Each user possesses a private key that is used to authorize transactions, preventing forgery. Moreover, a new transaction is not immediately added to the ledger but is bundled together with other pending transactions into a block. This is done to prevent double spending attacks which refers to the same currency used more than once in several transaction. [1]

There are several blockchains such as Bitcoin, Ethereum, Quorum, Hyperledger etc. Ethereum was the first blockchain that supported the implementation of smart contracts. A Smart Contract is a program deployed in a distributed network that can acquire outside information and update its internal state automatically. [4] The ultimate goal of the smart contract is to use computer intelligence instead of human intelligence to make the system more efficient and credible.

1.1 Motivation

The true potential of a smart contract can be exploited when data can be made available to the decentralised system in which the smart contract is deployed from the outside world. An oracle is an entity which provides trustworthy information from a service that is external to a blockchain. E.g., Suppose that George and Paul place a bet on who the winner of the US presidential election will be. George believes that the Republican candidate will win, while Paul believes that the Democrat will be the winner. They agree on the terms of the bet and lock their funds in a smart contract, which will release all the funds to the winner based on the results of the election.

Since the smart contract is passive and cannot interact with external data, it has to be supplied with the necessary information by an entity that has access to the outside world. This entity is called an oracle. After the election is over, the oracle queries a trusted API on behalf of the parties involved in the contract to find out which candidate has won and relays this information to the smart contract. The contract then sends the funds to Alice or Bob, depending on the outcome, more importantly depending on the data sent by the oracle service. Without the oracle relaying the data, there would not have been a way to settle this bet by transfer of funds. It has to be noted that the efficiency of the oracle service is quintessential to the proper working of the smart contract. If the service opts to send data that is false, this could result in funds being transferred to the wrong party. [9]

Several oracle solutions are currently available in the market. Oraclize.it [6] fetches data from a specified web source and publishes it to a blockchain application. They also maintain cryptographic proofs which show that the information originated from the correct source. TownCrier [8] is another oracle, which works in a similar fashion. It makes use of Intel Software Guard Extensions [8] to protect against tampering.

Oracles are essential implementations in numerous applications where there is a need for data that is external to be brought into the blockchain. The current oracles provide a solution without robust security guarantees that the blockchain provides. These therefore are impediments to security and could possibly become centralized points-of-failure or attack. Thus, the oracles in the Blockchain eco-system remain a subject of research and innovation. In this work we explore the existing literature and state of the art in blockchain, decentralized applications and oracles in block chain. We also aim to work on designing an Oracle system that improves upon existing architectures.

1.2 Research Question

Can we find an improvement upon existing oracle implementations in the blockchain eco-system?

1.3 Challenges

1. Lack of substantial literature on Oracles and their implementations.
2. Bringing innovation to the existing work on Oracles.
3. Insufficient documentation for oraclize.it service and other oracles.
4. Fewer implementations of smart contracts and oracles available on the web or other resources.
5. Compatibility of solidity compilers was often an issue. Hence, had to avoid using compilers having versions greater than 4.25.

1.4 Overview

The use of oracles to send data that is external, into the blockchain is critical to the proper functioning of smart contracts deployed within a blockchain. The concept of oracles is an interesting development in the blockchain industry. In order to gain a proper understanding of the working of oracles, it is important to understand the state of the art in blockchain. This work consists of review of Bitcoin and Ethereum blockchain concepts. Bitcoin blockchain uses a design that implements decentralization and immutability to manage the bitcoin currency. Unlike Bitcoin that allows only simple operations to be performed to ensure security, Ethereum also allows code of complex nature to be executed on the blockchain. Further, we understand the QTUM blockchain eco-system that uses the best of both Bitcoin and Ethereum blockchains. QTUM provides a Turing-complete blockchain stack that can execute smart contracts and decentralized applications and, uses the Ethereum Virtual Machine (EVM). However, in contrast to Ethereum, QTUM is built upon Bitcoin’s Unspent Transaction Output (UTXO) model and employs a Proof-of-stake consensus mechanism that is more practical for business adoption. A transaction in the blockchain consists of inputs and outputs. The outputs that have not been spent yet are referred to as UTXOs. We examine the various oracles currently available in the market and categorises them as centralized and decentralized. The work also explains implementations of smart contracts and oracles on the Qtum blockchain. An innovation from the existing implementations of oracles has also been discussed and designed which is a contribution to the literature on Oracles for blockchain.

1.5 Thesis Structure

The thesis is organized as follows. Chapter 2 presents the literature review on Blockchain, Bitcoin Core, Ethereum and Smart Contracts. Chapter 3 examines the different oracles that currently exist. Chapter 4 focuses on the design of a novel oracle system. Chapter 5 discusses the implementation of smart contracts and oracles that interact with the contracts deployed in the Qtum blockchain. Chapter 6 evaluates the newly proposed design of oracles and describes the novel aspects considered. The last section consists of conclusion and future works.

Chapter 2

Literature Review

2.1 Blockchain

Todays’ payment systems facilitate an exchange of currency between two entities namely a payer and a payee. Apart from the payer and payee, a payment system typically involves two more entities that act on behalf of the parties involved in the transaction. One entity manages funds on behalf of the payer, known as the issuing bank (or issuer), and another entity that maintains an account for the payee, known as the acquiring bank or acquirer.

The operations of a typical cash-like system (traditional banking system) are depicted in Figure 2.1. In a cash-like system, the payer's account is charged a fee before the actual payment takes place. Transactions require that a payment be made to the intermediary bank which usually is a percentage of the amount that the payer wishes to transfer. Businesses need to expend their profits towards high banking fees when the total amount transferred and the number of transactions increase. [7] Further, the amount is transferred through intermediaries and organisations that maintain their own logs with unrestricted access to alter them. Blockchain improves the payment system by ensuring and assuring parties of security through immutability, higher transfer speeds, lower conversion fees and a trustless service. This is achieved first and foremost by eliminating the need for centralized control (e.g., by banks) to transfer funds and perform third-party authorizations through the implementation of a shared distributed ledger. The distributed ledger is an append only log that stores all the transactions that occur, with a guarantee that they cannot be altered. Security is enforced by maintaining a hash of the previous block within every block such that the genesis (or the first) block can be verified. These transactions are made public so that all the stake holders could access them and check for integrity if needed. It requires only the parties involved to authorize using a private key.

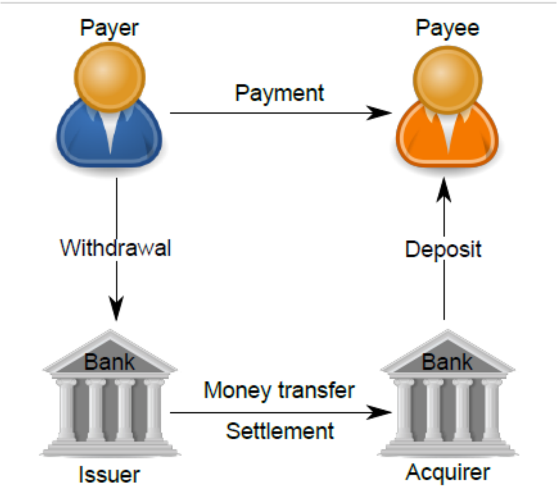


Figure 1: Cash-like Payment System Architecture

More formally, a blockchain can be defined as a system that decentralizes the working of an application where there is no single entity that controls the operation of the blockchain. The network has no central authority that is able to unilaterally approve invalid transactions or manipulate the state of the system through any means aside from normal submission of transactions for processing. The system uses a distributed ledger for storing the transactions. A new transaction can be added only at the end of the ledger, and no previous transactions can be modified. Moreover, a new transaction is not immediately added to the ledger but is bundled together with other pending transactions into a block. [1]

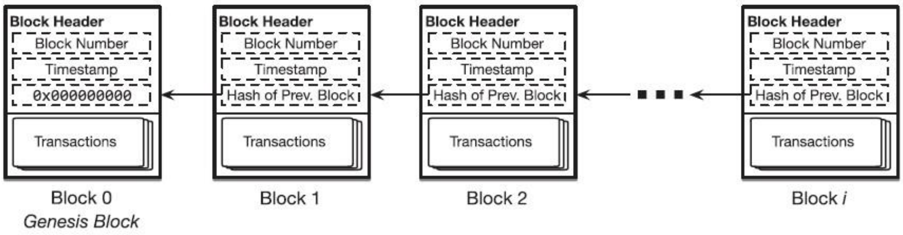


Figure 2: A Simple Blockchain

Blockchain as a Ledger

Blockchain is a distributed ledger technology which consists of three elements. First, the ledger is continuously amended and persistent, where all transactions effected on the blockchain are perpetually stored. Secondly, a blockchain is a distributed peer to peer ledger, where an entire copy of the blockchain is stored on every node in the network. If new transactions are affected, majority of nodes in the network must verify the legitimacy of the effected transaction and, if confirmed, every node is updated with the transaction. With this distributed authentication process, the core feature of blockchain which is the lack of a central entity or intermediary is facilitated. And thirdly, blockchain is asymmetrically encrypted and requires private and public keys to make a transaction. [10]

For example, Bitcoin technology uses a ledger to record the transactions and also track the ownership of tokens called Bitcoins. The tokens are distributed among nodes that represent accounts that are each uniquely identified by a public key. Bitcoin associates a public key with a balance in Bitcoins. Each item added to Bitcoin’s blockchain-backed ledger serves as a record of a transaction which denotes a transfer of Bitcoins from one public key to another.

2.2 Bitcoin

Bitcoin is a blockchain that supports the working of the bitcoin cryptocurrency. In Bitcoin, payments are performed by issuing transactions that transfer Bitcoin coins, referred to as BTCs, from the payer to the payee. Both payers and payees are peers in the network and are referenced in transactions by Bitcoin addresses. Each address of a peer in the network is mapped to a unique public/private key pair, which is used to transfer ownership of BTCs among addresses. A private key in bitcoin is a large random number between 1 and 2 ^ 256, which could take an attacker billions of years to try all possible combinations. An Elliptic Curve Digital Signature algorithm (ECDSA) on the private key is used to generate a public key. Users are able to generate public/private key pairs and the bitcoin address is a fixed length binary quantity of 26 to 35 alphanumeric characters which is derived from the public key. [2]

The bitcoin blockchain operates on top of a loosely connected peer-to-peer network, where users can join and leave the network as they wish. The peers or users connect to the blockchain network by requesting a list of current bitcoin peer addresses from the Domain Name System seeds. DNS is a protocol that maps domain names to IP addresses of users connected to the network. For example, the domain name www.oracle.com translates to the addresses 91.182.212.36 (IPv4). [15] Each Bitcoin address is computed from an ECDSA on the public key, for which the address owner knows the corresponding private key using a transformation based on hash functions. Hashes are one-way functions which allow the computation of the address using the public key but makes it infeasible to retrieve the public key from the address alone.

A Bitcoin transaction is created by digitally signing a hash of the previous transaction outputs, together with the public key of the recipient and referencing them as inputs. The output is a list of addresses that can collect the coins transferred by the transaction. A transaction output can only be redeemed once, after which the output is no longer available to other transactions. This process is facilitated in bitcoin by the implementation of Unspent Transaction Output model (UTXO, see section 2.2.1). The recipient in the transactions is able to redeem the coins using his private key that matches the public key used in the transaction creation process. Once ready, the transaction is signed by the user and broadcast in the P2P network. Any peer can verify the authenticity of a BTCs by checking the chain of signatures using public keys.

The difference between the input and output amounts of a transaction is collected in the form of fees by Bitcoin miners. Miners are peers that participate in the generation of Bitcoin blocks. These blocks are generated by solving a hash-based proof-of-work (PoW) algorithm (see section 2.2.2). More specifically, miners must find a nonce value that, when hashed with additional fields (the Merkle hash (see section 2.2.3) of all valid transactions, the hash of the previous block), the result is below a given target value. If such a nonce is found, miners then include it in a new block, thus allowing any entity to verify the PoW. The underlying proof-of-work allows different miners to find the nonce value and create different blocks nearly at the same time which is when a fork in the blockchain occurs. Forks are inherently resolved by the Bitcoin system through a mechanism where the longest blockchain that is backed by the majority of the computing power in the network will eventually prevail. [7]

2.2.1 UTXO Model

Bitcoin transactions use outputs from previous transactions as inputs in the construction and execution of a new transaction. For example, consider that Alice wants to send Bob 1 bitcoin and the transaction fee required is 0.25 bitcoins. Such a transaction could have the following inputs:

Input 1 – 0.5 BTC
Input 2 – 0.25 BTC
Input 3 – 0.5 BTC

The inputs considered above were outputs from the previous transaction. Considering the transaction fee of 0.25 BTC, the output of the transaction, i.e. the number of bitcoins Bob would actually receive, would be:

Output 1 – 0.5 BTC
Output 2 – 0.5 BTC

Bob would thus receive 1 bitcoin at the end of the transaction. The output of a transaction can either be classified as an unspent transaction output (UTXO) or be classified as spent transaction output. The unspent transaction output later becomes an input for transactions performed by Bob. For transactions such as that of Alice’s to be valid, they must only use unspent transaction outputs as inputs. This validity is checked for by the implementation of a UTXO set.

More formally, an unspent transaction output (UTXO) is an abstraction of electronic money that can be used for future transactions. Each UTXO represents a chain of ownership implemented as a chain of Digital Signatures where the owner signs a message (transaction) transferring ownership of their UTXO to the receiver's Public Key. The receiver node is able to unlock the value sent by the payer using its private key that matches the public key specified within the transaction. The UTXO model helps parallel processing of payments. It can be compared to bills where each bill can be used to perform parallel transactions.

UTXO Set

The function of the UTXO set is that of a global database that shows all the spendable outputs that are available to be used in the construction of a bitcoin transaction. When a new transaction is initiated by a user, it uses an unspent output from the UTXO set of the user, resulting in the set shrinking. On the contrary, when a new unspent output is created (for example, through mining), the UTXO set will grow.

Bitcoin full nodes download every block and transaction to check them against Bitcoin's consensus rules. [13] They are required to track all the unspent outputs in existence on the Bitcoin network in order to ensure that a user is not attempting to spend bitcoins that have already been spent, i.e. a double spending does not take place. To prevent double spending and fraud, inputs on a blockchain are spent when a transaction occurs, while at the same time, outputs are created in the form of UTXOs. These unspent transaction outputs may be used their owners (the holders of private keys) for their future transactions. [14]

A user’s bitcoin balance is the sum of all the individual outputs that can be spent by their private key. Therefore, when a user initiates a transaction, the outputs from the user’s UTXO set is used. All the unspent outputs must entirely be consumed when a transaction is being conducted, with change being sent back if the total value of the outputs is larger than the value of the transaction.

For example, if a user has a UTXO worth 10 bitcoins, but only requires 2 bitcoins for their transaction, then the entire 10 bitcoins is sent with two outputs being produced:

Output 1 – 2 BTC payment to the recipient
Output 2 – 8 BTC payment back to the user's wallet as change

A transaction consumes previously recorded unspent transaction outputs and creates new transaction outputs that can be used for a future transaction. This allows bitcoins to move from one owner to another, with each transfer consuming and creating UTXOs in a series of transactions. [12]

Components in a Transaction Output

scriptPubKey is a locking script placed on the output of a Bitcoin transaction that requires certain constraints to be satisfied so that a recipient could spend the bitcoins that have been transferred to them. Conversely, scriptSig is the unlocking script that satisfies the conditions placed on the output by the scriptPubKey, and this is what allows the bitcoins to be spent.

Using the previous example, in order for Bob to spend the bitcoins received from Alice, each output will contain a locking script, scriptPubKey, which must first be satisfied by the unlocking script, scriptSig which uses Bob's private key.

To illustrate, when Alice decides to initiate her transaction with Bob, the outputs that Bob receives contains bitcoins that can be spent only when the conditions laid out by the attached scriptPubKey are satisfied. When Bob decides to spend these outputs, he creates an input that includes an unlocking script, scriptSig, that must satisfy the conditions that Alice placed on the previous outputs using scriptPubKey before he can actually spend them. [12]

2.2.2 Proof of Work

As discussed already, the bitcoin blockchain is maintained by a peer-to-peer network. The transaction required by a user is executed by the peers in the network. When users add a new entry to a blockchain's ledger, they submit a transaction to a node in the network using a Remote Procedure Call (RPC) protocol [16]. The member broadcasts the transaction to the rest of the network for inclusion in a future block. Similarly, a user may submit a query to a network member about the contents of the blockchain's ledger. The parties involved in a transaction, such as the sender and receiver of Bitcoins on the Bitcoin blockchain do not have complete control of the execution of that transaction. Instead, the task falls to the members of the network who validate the transactions and the miners include the transaction within a block during the block creation process.

The members of the network or the peers can be classified into full nodes and miners. Full nodes are network members that own a full copy of the blockchain, containing every block and thus every ledger item, and keeps this copy synchronized with the latest updates to the blockchain by continuously monitoring the network for notification of new blocks. They help broadcast the transactions of users to the rest of the network. Full nodes commit computation and storage resources to this purpose. Also, by retaining a copy of the blockchain users do not have to trust an intermediary service to query the blockchain's state or submit transactions on their behalf. A subset of the members within a blockchain's peer-to-peer network not only maintain copies of the blockchain, but also actively construct and propose new blocks to be added to the chain. This process of constructing and adding new blocks to the network is known as mining, and these members are therefore referred to as miners. Miners must follow a certain protocol to ensure, the property of consensus where all members of the blockchain network together decide on the new block that is to be added to the chain and have an identical view of all previous blocks. This means that all blockchain copies are identical across the network. While there is an additional computational cost to assembling blocks and participating in a consensus protocol, full nodes may choose to run miners, because they have a vested interest in the successful operation of the blockchain or because of more explicit incentives.

In Bitcoin and several other blockchains, miners follow a proof-of-work algorithm (see Figure 3) to determine which miner appends the next block in the chain. The main rationale for using this algorithm is to prevent miners from immediately appending a newly prepared batch of transactions as a new block on the chain. If this were permitted, then many miners could continuously and simultaneously grow the chain, making it difficult to determine a globally recognized ordering of blocks, which is required to form a unified view of the blockchain's state. The system is designed such that each newly appended block must also contain a random value called a nonce, such that a cryptographic hash of the block's contents, including the numerical value of the nonce, falls below an upper threshold 't'.

Because a sound cryptographic hash function cannot be inverted, the only means of discovering a nonce satisfying this constraint is through brute-force search. This search process is the work and the satisfying nonce is the proof of this work. The first miner to find a proof appends the next block to the chain.

The following are the steps involved in mining a block using the proof-of-work consensus algorithm –

(1) Choose a set of pending transactions that have been received from the peers on the network but have not yet been included in any of the previous blocks and bundle these transactions as a payload p.

(2) Search for a nonce n that, when concatenated with p, produces a cryptographic hash that does not exceed a specific threshold. So we find, H(p. n) <= t for some bit string t.

(3) If some other valid block is received before n is found, append that block to the chain and return to Step 1.

(4) When the proof of work n is found, broadcast the new block which includes n, to the network. Return to Step 1.

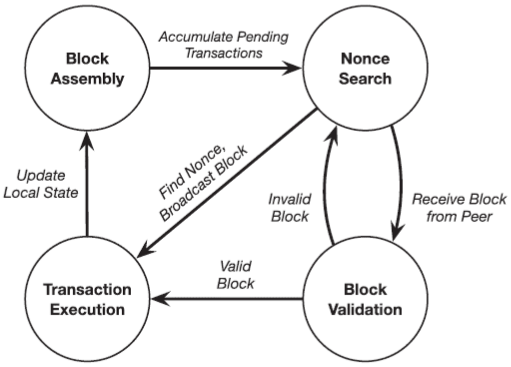


Figure 3: Proof-of-work consensus

Proof-of-work can be considered a repeated lottery that determines which miner is allowed to append the next block to the chain. All other miners validate and accept the new block before moving on to the next round of the consensus i.e., a new lottery. A miner's odds of winning a lottery round (its degree of influence over the operation of the blockchain) are proportional to the rate at which it can test nonce values in search of a valid proof of work. Therefore, a miner's influence is tied to its computing power. The tying influence of the result to the miner's computing power also gives proof-of-work consensus resilience to Sybil attacks. A Sybil attack is a technique in which an adversary disguises themselves as many users of a system to gain control over the system.

The quantity t is a bit string that represents an upper bound on the output of the hash function on the cryptographic hash of the block's contents, including the nonce used to produce the proof of work. This threshold is controlled by an adaptive and time-varying parameter known as the difficulty of the mining process. A smaller t value reduces the number of values that can serve as a valid proof of work, while a larger t value increases the number of such values, thus increasing the probability of finding a nonce. Therefore, mining difficulty can be defined as the expected number of values that must be tested before any of the network's miners succeeds in finding a proof of work. It is adjusted to keep the expected time delay between two successfully mined blocks constant even as the collective computing power of a blockchain's peer-to-peer network fluctuates over time as nodes join and leave the network. [1]

Forks

Proof-of-work consensus is analogous to a lottery system that is used to select the next block in the chain. Each new block is chosen non-deterministically and there can be situations where different miners find a nonce and append a block of transactions to the chain nearly the same time which leads to a split view of the blockchain's state among the members of the network, known as a fork in the chain (see Figure 4).

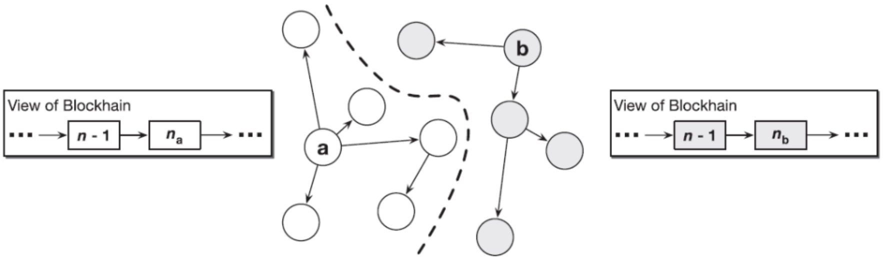


Figure 4: Fork in a blockchain

Consider for example two miners a and b in the blockchain network, and each is able to find a valid nonce that satisfies the proof-of-work consensus protocol nearly the same time. Both a and b broadcast the newly created block together with the nonce to the network. As the new blocks propagate through the blockchain's peer-to-peer network, one group of nodes accept and append a's block to their local copy while another group of nodes append b's block to their local copy. The same is the case with miners as well, where there is one group of miners that accept the block created by a and a second group of miners that accept b's block. The hashing power of the network is now split into two competing groups, thus making the network vulnerable to attacks.

This problem is resolved by adding a simple rule to the proof-of-work consensus protocol i.e., when a node observes a fork in the blockchain, it must always follow the longer of the two chains, i.e., the chain that contains a greater number of blocks. Every node should treat the longer fork as the canonical state of the chain, and every miner should append new blocks onto the head of this chain. One of the two forks will have its first block adopted by the group of nodes in the network with more hashing power. This subset of the network will then be able to mine and append new blocks at a faster rate than the nodes backing the fork with lesser hashing power. By the very fact, it will become longer over a period of time, and this disparity will only keep growing as more nodes identify the longer fork and abandon the shorter fork. Finally, it would be the longer fork that prevails.

The blockchain's structure where each block contains a hash of its previous block, and the possibility of forks to occur has led to the concept of confirmations. When a transaction is included in what appears to be the newest block on the chain, it is not yet certain that this block will become part of the canonical chain, and therefore that the transaction will finally become a part of the network. There is a probability that a block turns out to be part of an eventually abandoned fork. Moreover, the more successors a block has in the chain, the more resistant those block's transactions are to attack by adversaries. This is because an attacker must have the hashing power necessary to force the network to roll back all of a block's successors before it can alter the contents of the block itself. Therefore, many blockchain applications will not consider a transaction immutable until a sufficient number of blocks have been appended as successors to the transaction's block. Each successor reduces the probability that the transaction is manipulated by an attacker or discarded as part of an abandoned fork. [1]

2.2.3 Merkel Trees

A Merkle tree is a data structure which constitutes a hash tree where the root hash of a particular node (leaf nodes) is converged at by a sequence of hash operations along the path that converges that converges at the root node. This allows us to verify the authenticity of a particular transaction and that it has not been altered (see Figure 5). They help in building cryptographic accumulators which helps us verify whether given transaction belongs to a block in a blockchain.

Therefore, a Merkle tree can be defined as a binary tree in which the data is stored in the leaf nodes. More specifically, given a set X which contains n elements. Each of the elements 1, 2, …, n is assigned to the leaf nodes of the binary tree. Suppose a[i, j] denotes a node in the tree located at the ith level and jth position. Here, the level refers to the distance to the leaf nodes and it is evident that the leaf nodes are located at distance 0. The positions numbered starting from 0 within the level and are considered from the left-most position. For example, the leftmost node of level 1 in a tree a, is denoted by a[1, 0]. The nodes at a distance one or more from the leaf nodes are considered to be intermediate nodes and they are computed as the hash of their respective child nodes i.e., a[i+1, j] = H(a[i, 2j], a[i, 2j+1]). Given n leaves, time complexity for constructing the tree is O(n) and the time required for proving the membership of an element is O(log(n)).

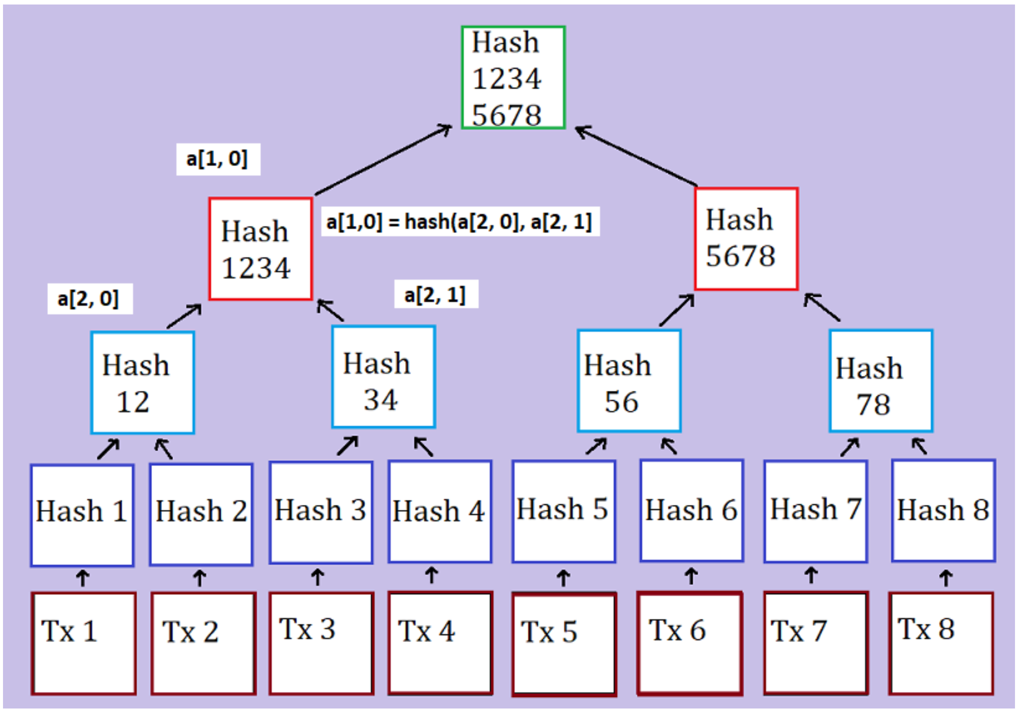


Figure 5: A Merkle tree of depth 3, accumulating transactions Tx 1 through Tx 8

The "hash" of a block is a hash function applied on the block header which contains roughly 200-byte piece of data which includes a timestamp, nonce, previous block hash and the root hash of a data structure called the Merkle tree storing all transactions in the block. In a Merkle tree, the intermediate nodes are the hash of its two children and a single root node which is also formed from the hash of its two children, representing the "top" of the tree. The purpose of the Merkle tree is to facilitate the verification of data in the leaf nodes by the using only the header of a block. Here, the hashes propagate upward, where if an adversary attempts to swap in a fake transaction into the bottom of a Merkle tree or alter its contents in any manner, this change will cause a change in the node above, all the way until the root of the tree and therefore the hash of the block, causing the protocol to register it as a completely different block. The altered block becomes invalid since the nonce value does not satisfy the hash function applied on the contents of the newly altered block. [18]

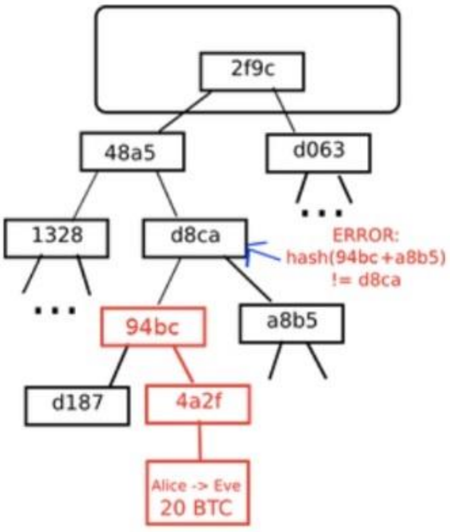


Figure 6: An attempt to change the Merkel Tree

2.3 Ethereum

Ethereum is a blockchain like bitcoin, but also has certain additional features that helps it extend its scope beyond just the management of cryptocurrencies. It shares many of the common elements such as a peer-to-peer network connecting participants, a Byzantine fault–tolerant consensus algorithm for synchronization, the use of cryptographic primitives such as digital signatures and hashes, and a digital currency (ether). Yet, the purpose of the design and construction of Ethereum is different from those of the open blockchains that preceded it, including the Bitcoin. Ethereum's purpose is not primarily to be a digital currency payment network. While the digital currency ether is necessary for the successful operation of Ethereum as a blockchain, ether is only intended as a utility currency to pay for use of the resources on the Ethereum platform such as memory and computation. [3]

Unlike Bitcoin, which has a very limited scripting language (a set of opcodes), Ethereum is designed to be a general-purpose programmable blockchain that runs a virtual machine capable of executing code of arbitrary and unbounded complexity. Bitcoin's Script language is, intentionally, constrained to simple true/false evaluation of spending conditions. Whereas, Ethereum's language is Turing complete (see section 2.3.3), meaning that Ethereum can function as a general-purpose computer that is able to execute complex code. The idea was that by using a general-purpose blockchain like Ethereum, one could develop applications on a blockchain without having to implement the underlying mechanisms of peer-to-peer networks, blockchains, consensus algorithms, etc. The Ethereum platform is designed to abstract the complex details of the blockchain and provide a secure programming platform for developing decentralized blockchain applications. Unlike the bitcoin that tracks only the state of currency ownership Ethereum tracks the state transitions of a general-purpose data store. This store that can hold any data in the form of key–value pair. A key–value data store holds values required by a program, each referenced to by its corresponding key. For example, the value “04-09-1992” is referenced by the key “DoB”. It is similar to databases used by today’s modern applications which contain a set of records that are uniquely identified by a key value. Ethereum provides memory that stores both data and code (usually used to manipulate the state of the data), and it uses the Ethereum blockchain to track how the state of the memory changes over time. Like with a general-purpose computer, Ethereum can load code into its state machine and execute that code, storing the resulting state changes in its blockchain. This helps us code logic and constraints and store them on the block chain, which can be executed by members of the network. This set of data and executable code on the blockchain is referred to as a smart contract which is discussed in detail in the coming sections. [19]

2.3.1 Accounts in Ethereum

Similar to Unspent Transaction Output Model (UTXO) in Bitcoin, in Ethereum, the state is stored in objects called "accounts" and each account has a 20-byte address and state transitions refer to transfer of Ether between accounts and change in the data stored in each of the accounts. An Ethereum account contains four fields:

- The nonce, a unique value used to ensure that each transaction is processed only once
- The account's current ether balance
- The account's contract code
- The account's storage (empty by default)

"Ether" is the crypto-fuel or cryptocurrency of Ethereum and is used to pay transaction fees. In general, there are two types of accounts in the Ethereum blockchain: externally owned accounts, controlled by private keys, and contract accounts, controlled by their contract code. An externally owned account has no code, and one can send messages from an externally owned account by creating and signing a transaction using their private key, where as in a contract account, every time the contract account receives a message its code activates, allowing it to read and write to internal storage and call other contract or even create new contracts.

Contracts in Ethereum are decentralized code that live inside the Ethereum blockchain and run on the Ethereum virtual machine and should not be considered as something that is just compiled and executed. A specific piece of code is executed when called by a message or transaction from either an externally owned account or a contract within the blockchain. The contracts have direct and full control over their own ether balance and their key-value store used to store persistent data usually required by the functionality implemented within their code. Thus, a contract is a decentralized and immutable piece of code together with data that can be used to implement a specific requirement. [18]

2.3.2 Messages and Transactions

The term "transaction" in Ethereum refers to a signed data package that stores a message, to be sent from an externally owned account. It represents a message that ideally includes data and currency. The currency either constitutes of the amount transferred to the recipient or transaction costs. A transaction contains the following parts:

- A recipient address
- A signature used to identify the sender
- An amount in ether (transferred from the sender to the recipient)
- A data field (optional)
- A STARTGAS value, represents a limit on the number of computational steps a transaction can take
- A GASPRICE value, which is a fee the sender pays for each computational step and is usually measured in bytes.

The first three are standard fields normally present in every blockchain. The data field is null by default, and Ethereum has opcodes which a contract can use to access data and the state of the contract. For example, if a contract is functioning as an on-blockchain betting service, then it may interpret the data being passed to it as containing two "fields", where the first field contains the value of an outcome and the second field is the value staked by the better. The contract reads these values from the message data and places them in the key-value data store.

The STARTGAS and GASPRICE fields are important for Ethereum's anti-denial of service model architecture. In order to prevent accidental or adversarial infinite loops and the resulting wastage of memory and computational resources, each transaction is required to set a limit on how many computational steps of code execution is allowed. The fundamental unit of computation is "gas". In Ethereum a computational step usually costs 1 gas, but some operations cost higher amounts of gas because they are more computationally expensive. There are situations where the amount of data stored as part of the state of the transaction is large and a fee of 5 gas is charged for every byte. The purpose is to discourage an adversary who is financially rational from consuming computation, bandwidth and storage on the blockchain network. Therefore, a transaction that consumes resources in the network must pay a gas fee which is proportional to the resources consumed. [18]

Messages

Contracts can send "messages" to other contracts. Messages in Ethereum refer to objects that contain data and value (in ether) which are passed between two accounts. A message is usually created when contracts interact with each other or by a transaction. A message contains the following fields:

- The sender of the message (implicit)
- The recipient’s address
- The amount of ether to be transferred
- A data field (optional)
- A STARTGAS value

We notice, a message is similar to a transaction, except that it is produced by a contract and not by an externally owned account, as is the case with transactions. A message is produced when a contract’s code executes the CALL opcode. Like with a transaction, a message thus leads to the recipient account (a contract account) running its code.

Thus, contracts can be made to interact with other contracts in exactly the same way that externally owned accounts can.

The gas required when making a transaction is the total gas required by the network for the consumption of its resources, which even includes all sub-executions present within that transaction. For example, if an external actor A sends a transaction to B with 100 gas, and B consumes 60 gas before sending a message to C, and the internal execution of C consumes 30 gas before returning, then B can spend another 10 gas before running out of gas. [18]

Turing Completeness in Ethereum

The term 'Turing complete' refers to an English mathematician Alan Turing, who is considered the father of computer science. In 1936 he created a mathematical model of a computer consisting of a state machine that manipulates symbols by reading and writing them on sequential memory. Turing later provided a mathematical foundation to answer questions about universal computability, meaning whether all problems are solvable. He proved that there are classes of problems that are not computable. Specifically, he proved that the halting problem is not solvable. The Halting problem refers to the question of whether it is not possible, given an arbitrary program and its input, to ascertain whether the program will eventually stop running. Ethereum's ability to execute a stored program in a state machine called the Ethereum Virtual Machine (EVM), while reading and writing data to memory makes it a Turing-complete system and therefore a Universal Turing Machine. The EVM is able to compute any algorithm, given the limitations of finite memory and hence prone to the halting problem.

The innovation in Ethereum's is that it combines the general-purpose computing architecture of a stored-program computer with a decentralized blockchain. It thus creates the so called 'world computer' that is distributed across all the nodes in the network. In Ethereum, programs run 'everywhere' i.e., in each node, yet maintain a common state or behaviour which is enforced by the consensus rules of the blockchain. However, Turing completeness is very dangerous, particularly in open access systems like public blockchains, because of the halting problem. For example, modern printers are Turing complete and can be given files to print that send them into a frozen state after which they cannot be used to perform other operations. The fact that Ethereum is

Turing complete means that any program of any complexity can be computed by Ethereum. But that flexibility brings issues relating to security and resource management. An unresponsive printer can be restarted and continued to be used by the network. But the same is not possible with a public blockchain.

Implications

Turing proved that we cannot predict whether a program will terminate by analysing it on a computer. More specifically, we cannot predict the path of a program without running it. Turing-complete systems can run into ‘infinite loops’ and ultimately results in the usage of an enormous amount of resources. In Ethereum, this poses a challenge: a smart contract can be created such that it runs forever when an externally owned account or another contract in the blockchain attempts to invoke it. This kind of a situation is termed, a Denial of Service (DoS) attack. How does Ethereum constrain the utilization of resources by a smart contract if it cannot predict the resource usage in advance?

In order to prevent the DoS attack, Ethereum introduces a metering mechanism called gas. As the EVM executes a smart contract, it carefully accounts for every instruction (computation power, memory, etc.). Each instruction has a predetermined cost in units of gas. When a transaction triggers the execution of a smart contract, it must include an amount of gas that sets the upper limit of what can be consumed running the smart contract. The EVM will terminate execution if the amount of gas consumed by computation exceeds the gas available in the transaction. The gas to pay for computation on the Ethereum network is bought with ether which is sent along with a transaction. Any unused gas, post the execution of the transaction is refunded back to the sender of the transaction. [17]

2.4 QTUM blockchain

The Qtum blockchain is a UTXO based smart contract system with a proof-of-stake consensus model. It uses the best of both Bitcoin and Ethereum blockchains. It has adopted the UTXO model of bitcoin which is more secure and uses the Ethereum Virtual Machine as its platform for contract execution. Qtum uses an Account Abstraction Layer (AAL) which maps the UTXO-based model to an account-based structure that is present in the EVM and achieves interoperability. It implements an on-chain governance system based on the Decentralised Governance Protocol (DGP) that allows QTUM token holders to participate in the voting and negotiation of the upgrade and iteration of the blockchain network. It also introduces a way for other participants in the ecosystem, including developers, community member representatives, miners, and other multi-party participants to propose and vote for proposals. DGP manages the parameters responsible for the proper functioning of the blockchain network through smart contracts embedded within the genesis blocks.

Some of the problems that the Qtum blockchain addresses are:

1. Different blockchain platforms that exist today are not compatible with each other. For example, the Bitcoin ecosystem based on the UTXO (Unspent Transaction Output) model is not compatible with the Ethereum ecosystem based on the Account model, and the level of interoperability between these blockchains is not sufficient enough for businesses to adopt them.

2. On-chain governance of critical technical parameters that affect the system is difficult to achieve. For most decentralized platforms, once the mainnet deployment is completed, upgrade and governance of the blockchain is a major problem. Updates to Ethereum involved hard forks which are not backward compatible.

3. The Proof-of-Work consensus mechanism requires spending of large amount of energy for mining and incentives for miners. The security of the system is largely dependent on the hashing power of miners in the network, where there is a risk of centralization in mining computing power and thus the network faces the danger of a ‘51 percent’ attack. [21]

2.4.1 Account Abstraction Layer (AAL)

The EVM is stack-based with a 256-bit machine word. Smart contracts that run on Ethereum use this virtual machine for their execution. The EVM is designed for the blockchain of Ethereum and thus, assumes that all value transfers use an account-based method. Qtum is based on the blockchain design of Bitcoin and uses the UTXO-based model. To translate the UTXO-based model to an account-based interface for the EVM and decouple the value transfer layer from the contract execution layer, Qtum created the Account Abstraction Layer (AAL). This facilitates interoperability and platform independence.

Qtum developed mechanisms for conversion between smart contract operations and UTXO operations, and has designed and developed four new Bitcoin opcodes:

- OP_CREATE: create a smart contract
- OP_CALL: call smart contract (send QTUM to the contract)
- OP_SPEND: spend QTUM in smart contract
- OP_SENDER: allow an address other than contract caller to pay for Gas

OP_CREATE passes the contract bytecode to the virtual machine. OP_CALL sends data, gasPrice, gasLimit, VMversion and other key parameters required to run smart contracts through transaction scripts, and finally passes them to the virtual machine. During the block creation process, the validator’s script parses the contents in the transaction. In addition to making regular checks on transaction scripts, it also checks whether transactions contain the above-mentioned opcodes. Note that the above-mentioned opcodes are relate to operations that use funds which are stored as UTXOs. When the mentioned opcodes are encountered, those transactions are set aside to be separately processed. The contract transactions are then processed by the EVM into a special “Expected Contract Transaction List” which is executed by validator nodes. These transactions are then run on the EVM, and the resulting output is converted into a spendable Qtum transactions that support the operations related to EVM execution. Relying on this design, the Qtum x86 virtual machine can run on the blockchain in parallel with the EVM (Ethereum Virtual Machine), without the need to significantly modify the underlying protocol and retaining good functional scalability. Thus, any virtual machine based on the account model can be adapted to run on the Qtum blockchain when required.

Qtum has also adopted the concept of Gas from Ethereum which is a metering mechanism to track and restrict the use of resources belonging to the network. Use of the Gas model can prevent endless loops caused by errors and malicious attacks. They also encourage contract designers and users to make reasonable use of on-chain resources. Normally the address of the contract call sender pays the Gas, but the OP_SENDER opcode allows a third-party address, such as a distributed application service provider, to pay the Gas. As in Ethereum blockchain, there is also a state rollback for an ‘out of Gas’ scenario and a refund of remaining Gas after successful execution. [20]

2.4.2 Proof-of-Stake Algorithm

Proof of stake (PoS) is another type of consensus algorithm by which a cryptocurrency blockchain network aims to achieve distributed consensus. In PoS-based cryptocurrencies, the creator of the next block is chosen using various combinations of random selection and wealth or age. Here wealth and age refer to the stake of the validator who is responsible for the block creation and receives the transaction fees. A validator is the one who proposes the next block to be appended to the blockchain. The chance of a validator being selected is proportional to the total amount they have staked. In contrast, the algorithm of proof-of-work based blockchains such as bitcoin uses mining that involves solving computationally intensive puzzles to validate transactions and create new blocks.

Proof of stake must have a way of defining the next valid block in any blockchain. Selection by account balance would result in centralization, as the single richest member would have a permanent advantage. Instead, several different methods of selection have been devised. Certain blockchains that implement Proof-of-Stake algorithm use randomization to predict the validator of the next block, by using a formula that looks for the lowest hash value in combination with the size of the stake. Since the stakes are public, each node can predict with reasonable accuracy which account will next win the right to forge a block. [22]

Coin Aging

Some blockchains use a proof-of-stake system that combines randomization with the concept of ‘coin age’, a number derived from the product of the number of coins multiplied by the number of days the coins have been held by a validator. Older and larger sets of coins provide a miner, greater probability of mining the next block. For example, coins that have not been spent for at least 30 days begin competing for the next block. Once a stake of coins has been used to sign a block, its age is reset to zero and thus waits for 30 more days before mining another block. Also, the probability of finding the next block reaches a maximum after 90 days in order to prevent very old and very large collections of stakes from attaining a monopoly in the proof-of-stake consensus mechanism.

2.5 Smart Contracts and DApps

Until now, we have understood the concept of blockchain, different blockchains such as Bitcoin and Ethereum, Turing complete blockchains that run as Universal Turing Machines (UTM), which can execute code of unbounded complexity. The purpose of Turing complete blockchains is to execute immutable code (or programs) that run deterministically. This concept of immutable programs that execute in a deterministic fashion, facilitated by a Turing complete blockchain is referred to as a smart contract. A decentralized application or a DApp is more complex and is composed of at least two components namely, a smart contract on a blockchain and a web user interface that interacts with the contract. A DApp may also include other components such as, a decentralized storage protocol such as InterPlanetary File System (IPFS) and a decentralized messaging protocol (for example, whisper is a peer-to-peer messaging protocol).

2.5.1 Contract Components

Smart contracts are written in a high-level language such as solidity. There are different components in a contract that have to be properly understood in order to efficiently design, develop, deploy and call a contract. We have used solidity to develop smart contracts for the Ethereum blockchain. The different components in a smart contract are as follows:

- **Contract Bytecode**

Once the contract has been written they are compiled to get the bytecode. This contains a set of instructions that are to be executed by the Ethereum Virtual Machine (EVM). They are not human readable and is understood and used by the EVM.

- **Contract Address**

Once compiled, the bytecode is deployed on the Ethereum (or any other blockchain) network using a contract creation transaction. The newly created contract is identified by an Ethereum address, an outcome of the creation transaction. The address can be used in a transaction as a recipient in order to send funds to the transaction or to call one of its functions.

- **Application Binary Interface (ABI)**

An ABI is an interface between two program modules and defines how data and functions within the smart contract (bytecode) are accessed. It is the primary way to encode and decode data present as machine code on the blockchain. Ethereum uses an ABI to encode contract calls and read data from the contract by defining functions that can be invoked and their arguments. More specifically, an ABI is specified as a JSON array of data and functions defined within the contract.

- **Contract State**

The state of the contract refers to the data stored within the contract, is changed by the execution of transactions. It is only when the transaction executes successfully that the transactions are recorded globally. If an execution fails because of an error all of the changes made to the state is ‘rolled back’. A failed execution is still recorded on the blockchain as an attempt.

2.5.2 Contract Development

Solidity is the high-level programming language we will use to develop smart contracts and understand their working. Solc is the compiler that converts high-level code written in Solidity to EVM bytecode. We will use a web-based development environment called Remix IDE for implementing smart contracts.

Let us consider a simple contract ‘LateFlightReimbursement.sol’ that makes payment for claims made, when flights are delayed.

```
pragma solidity ^0.4.25;
contract LateFlightReimbursement{
   uint256 public scheduledDepartureTime;
   uint256 public actualDepartureTime;
   bool public flag;
   address owner;
   address  insured;
   uint256 public ownerBalance;
   constructor(address wallet) public payable {
       flag=false;
       owner=msg.sender;
       ownerBalance=owner.balance;
       insured=wallet;
   }
   function setScheduledDepartureTime(uint256 _scheduledDepartureTime) public {
      scheduledDepartureTime=_scheduledDepartureTime;
   }
   function setActualDepartureTime(uint256 _actualDepartureTime) public payable{
      actualDepartureTime=_actualDepartureTime;
      if(actualDepartureTime>scheduledDepartureTime){
            insured.transfer(msg.value);
            flag=true;
            ownerBalance=owner.balance;
      }
      else
         flag = false;
   }}
```

In the above example we observe that a smart contract is defined in solidity using the '*contract*' keyword. The contract consists of persistent variables (each of which consume space in the blockchain), a constructor that initializes the variables and functions. A function should be annotated with the '*payable*' keyword, in order that it may collect or receive funds in ether. The above contract accepts an 'address' value that represents the insured's wallet, which is defined in its constructor. It contains functions of setting the values of 'scheduled departure time' and 'actual departure time'. When the value of 'actual departure time' is greater than 'scheduled departure time' the function setActualDepartureTime() transfers ether to the insured's wallet using the *transfer()* method. The different data types used in the contract are *uint 256* represents unsigned int, *bool* defines a Boolean value and *address* data type is used to store and Ethereum address.

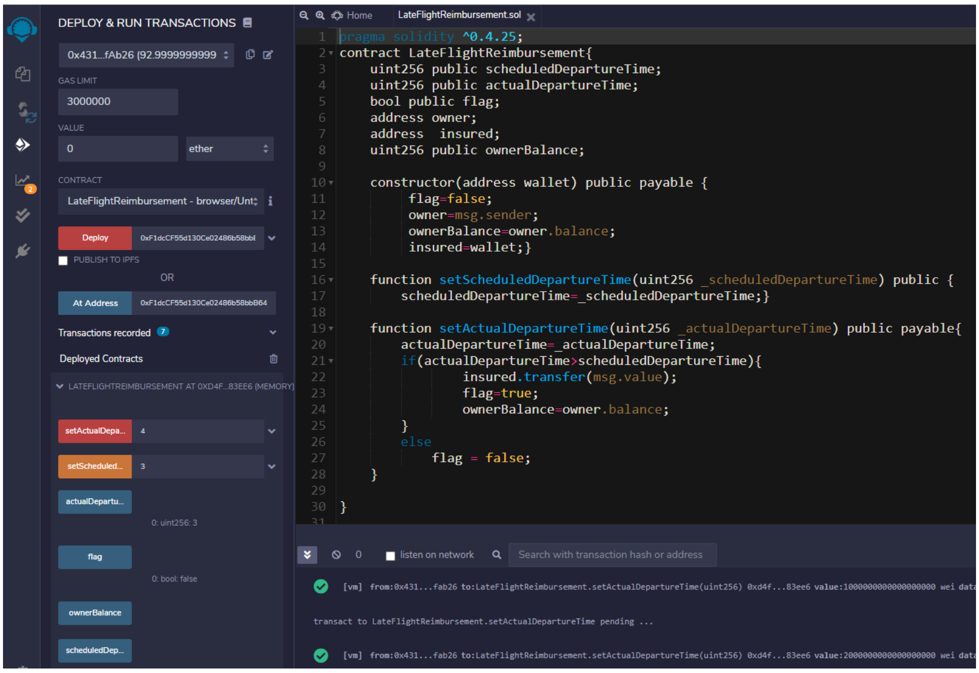
2.5.3 Contract Deployment

Before a contract is deployed, it must be compiled using a solidity compiler. We use ‘solc’ to compile solidity code and get the ABI and bytecode that are important components used to deploy and call the contract. Bytecode is the machine code that is deployed in the EVM and the ABI, which represents the structure of the contract is used to access the contract deployed in the blockchain.

The ABI for the above-mentioned contract is of the form –

```
[
	{ "constant": false, "inputs": [{"name": "_actualDepartureTime", "type": "uint256"}], "name": "setActualDepartureTime", "outputs": [], "payable": true, "stateMutability": "payable", "type": "function"},
………,{"constant": true, "inputs": [], "name": "scheduledDepartureTime", "outputs": [{"name": "", "type": "uint256"}], "payable": false, "stateMutability": "view", "type": "function"}
]
```

Once compiled, the code is deployed onto the blockchain. In our case we have compiled and deployed the contract using Remix IDE (see figure 7). When the contract is deployed a transaction is executed on the blockchain, and a contract is created. The contract represents an account that is identified using an address obtained as the result of the contract creation transaction. After deployment, the components used to access the contract are the contract address and the ABI. An example contract address is ‘0xd4fe68a6aBC4e8BD59b9fB15f13c2A40cD883EE6’



0x431...fAb26 (92.9999999999

GAS LIMIT 3000000

VALUE 0 ether

CONTRACT LateFlightReimbursement - browser/Unt

Deploy 0xF1dcCF55d130Ce02486b58bbE

At Address 0xF1dcCF55d130Ce02486b58bbB64

LATEFLIGHTREIMBURSEMENT AT 0XD4F...83EE6 (MEMORY)

setActualDepa... 4

setScheduled... 3

actualDepartu... 0: uint256: 3

flag 0: bool: false

```
pragma solidity ^0.4.25;
contract LateFlightReimbursement{
    uint256 public scheduledDepartureTime;
    uint256 public actualDepartureTime;
    bool public flag;
    address owner;
    address  insured;
    uint256 public ownerBalance;

    constructor(address wallet) public payable {
         flag=false;
         owner=msg.sender;
         ownerBalance=owner.balance;
         insured=wallet;}

    function setScheduledDepartureTime(uint256 _scheduledDepartureTime) public {
        scheduledDepartureTime=_scheduledDepartureTime;}

    function setActualDepartureTime(uint256 _actualDepartureTime) public payable{
        actualDepartureTime=_actualDepartureTime;
        if(actualDepartureTime>scheduledDepartureTime){
                insured.transfer(msg.value);
                flag=true;
                ownerBalance=owner.balance;
        }
        else
            flag = false;
    }

}
```

```
[vm] from:0x431...fab26 to:LateFlightReimbursement.setActualDepartureTime(uint256) 0xd4f...83ee6 value:1000000000000000000 wei data

transact to LateFlightReimbursement.setActualDepartureTime pending ...

[vm] from:0x431...fab26 to:LateFlightReimbursement.setActualDepartureTime(uint256) 0xd4f...83ee6 value:2000000000000000000 wei data
```

Figure 7: Deploying a contract using Remix IDE

Chapter 3

Oracles in Blockchain

In the previous section we considered an example smart contract, which contained functions that could be called from an externally owned account (EOA) to alter the state of the contract. The vulnerabilities in the system can be exploited for financial gain and therefore it is important that calls made to the contract, and data triggering certain actions such as transfer of funds within the contract is verified. Otherwise, an adversary could either send incorrect data to the contract or hinder availability of data when required by the smart contract and its execution is stalled. So, it is critical that correct data is made available to contracts on the blockchain and at the right time. This piece of work is critical to the working of smart contracts and is implemented using an Oracle. [11]

3.1 Introduction to Oracles

An Oracle is essentially a system that answers a question that is external to the blockchain. It acts as a bridge that connects blockchains and the outside world. It system consists of off-chain and on-chain components, where off-chain components are responsible for collecting the data in a secure fashion and on-chain components comprise of oracle's contracts that are used to send data to the contract that has requested for the oracle's service. They send extrinsic information to contracts on the blockchain by executing transactions that contain the data payload. However, such data simply cannot be trusted and used to execute high value contracts. It is important to create a trustworthy model of an oracle, so that they are acceptable by businesses to execute their use cases.

All oracles in general, contain a few critical functions by definition, which include the following –

- Collect data from a source external to the blockchain.
- Transfer that data on-chain together with signature.
- Data is made available to a smart contract's storage.

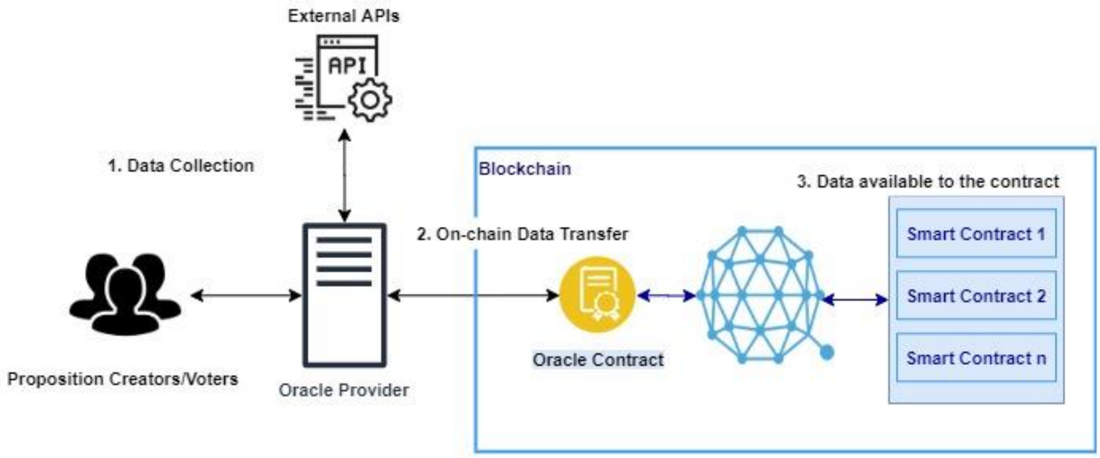


Figure 8: An Oracle System in general

It is important to note that oracles are not data sources, but a middleware that queries, verifies and authenticates data sources external to the blockchain and then makes it available to the smart contract on the blockchain that requested the information. Another key advantage of using an oracle is that it enables off-chain computation. Computation and usage of resources on the blockchain is costly, and an oracle can be used to perform large computations outside the network, with only the result being returned for use to the blockchain. There are many oracles that differ in their design and functionalities, currently in the market. How an oracle functions is dependent on the use case or purpose it is designed for.

Oracle Design Patterns

An oracle can be set-up in 3 three different ways namely - immediate-read, publish-subscribe, and request–response. Oracles having immediate-read design is implemented in situations where data is required to make an immediate decision. For example, 'Is a person above 18 years of age?'. This type of an oracle stores data only once (which may be updated later) and other contracts on the blockchain that require that information can access it using a request call. Publish-subscribe setup is where an oracle broadcasts data to the blockchain network and those who have subscribed listen to the broadcasted data.

The request-response model is used when the data space required is too huge to be made available in a smart contract and users need only a small portion of the data for computation at a time. When an externally owned account interacts with a contract in the blockchain that uses the oracle, it results in an interaction with the oracle's contract which is essentially a request to the oracle, with the associated arguments detailing the data requested. The oracle then interacts with its contract to fetch the query and the different parameter, which is used to perform the actual query of the off-chain data source. The result of the query is signed by the oracle owner and delivered in a transaction to the contract that made the request, either directly or via the oracle's contract. [19]

Although blockchains can be setup in three different ways, oracles are fundamentally classified as centralized and decentralized. A centralized oracle is controlled by a single entity, whereas a decentralized oracle is a system where data is collected from several sources and controlled by one or more entities that are equally privileged. (See sections 3.2 and 3.3)

3.2 Centralized Oracles

A centralized oracle is governed by a single entity, which is the only provider of information for the smart contract. Having only one interface can be very risky, in the sense, a reliability of a contract that is secure on the blockchain is entirely dependent on the entity controlling the oracle. A bad actor can manipulate the data being sent from the oracle to the on-chain contract, which will have a direct impact on the smart contract. Thus, the main problem that exists with centralized oracles is of a single point of failure, which makes the contracts vulnerable to attacks. Contracts of higher value act as incentives for bad actors to attack the oracle system. One of the most widely used centralized oracle is called ‘Provable’.

3.2.1 Provable (previously Oraclize)

Provable [6] is the leading oracle provider in the blockchain industry, fulfilling thousands of requests every day on various platforms such as Ethereum, Hyperledger and R3 Corda. As explained already, an oracle is an intermediary service that is introduced as a result of blockchain contracts not having the ability to make request to external services that provide data which is critical to their functioning. But, to rely on an intermediary system, would be to betray the reduced-trust and security model of the blockchain. Therefore, it is important that a reliable and trustworthy model which does not comprise the security provided by the blockchain.

Provable follows a request-response model where a request is made by a relying contract to the oracle’s contract and a response containing the required data is returned. It has developed a trust model which demonstrates that the data fetched from a source is not altered and is genuine. For this purpose, it sends data to the contract in the blockchain, together with an authenticity proof. Authenticity proofs are cryptographic guarantees that the data has not been tampered with. Provable uses TLSNotary[27] proofs that allow it to provide evidence that HTTPS web traffic occurred between the client and the server (data source provided by the contract). TLSNotary allows an auditee (client) to make a https request to a web server such that the auditor is able to verify the response before the client can access them, by temporarily withholding some information required to view the response. The provable service uses the AWS (Amazon Web Services) machine instance as the auditor which attests that the data received is the result of the client’s interaction with the specified data source. The outcome is a TLSNotary secret, which is stored safely in the AWS instance that runs the off-chain component and is used to provide honesty proofs.

A valid request from a smart contract to the provable service contains –

- A data source, such as a URL to an API
- A query
- An authenticity proof (optional)

1) Data Source

It refers to a data source that the contract delegates as its trusted source. E.g., nasdaq.com. *Provable* currently offer services for data sources such as a URL, Wolfram Alpha, IPFS, Random and Computation. AURL can invoke an API or fetch a web page. Wolfram Alpha is a computational engine that can be specified as a data source in the provable query to retrieve an answer to the query string provided. Random is used to generate a random value. InterPlanetary File System is a peer-to-peer network protocol for storing and sharing content in a distributed file system.

2) Query

A query is an array of parameters, which is used by the oraclize service to process the request. For example, in the case of the data source being specified as ‘Wolfram Alpha’, the query parameter can be ‘Flip a coin’ which generates a random value which is either a ‘heads’ or a ‘tails’.

3) Authenticity Proofs

Authenticity proof form a critical part of trust model of Provable. They refer to the type of authenticity proofs required to prove that the data supplied has not been tampered with. E.g., E.g., *proofType_TLSNotary* and *proofStorage_IPFS*. The former returns a TLSNotary Proof bytes as the proof argument in the callback transaction and the latter returns base58-decoded IPFS multihash as the proof argument. A multihash is a protocol used to uniquely identify files stored on IPFS.

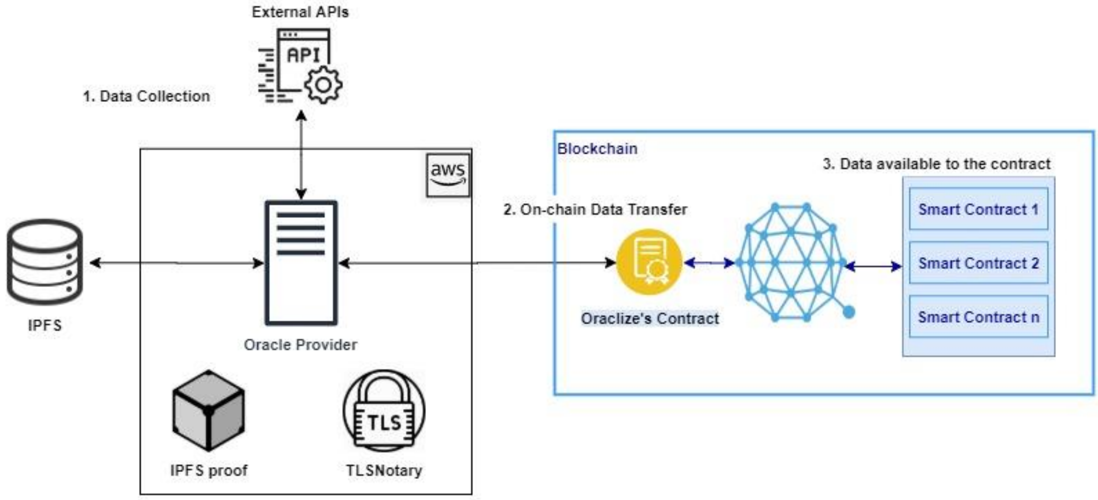


Figure 9: Oraclize - A Centralized Oracle System

Working

Let us now consider a working example of the Provable service, where we call the WolframAlpha computing engine which returns the result for a random experiment of ‘tossing a coin’. The contract transfers its funds to the bettor, if the outcome he has staked matches the result of the random experiment.

Code:

```
pragma solidity ^0.4.25;
import "github.com/provable-things/ethereum-api/provableAPI_0.4.25.sol";
contract ExampleContract is usingProvable {
   address bettor;
   string result;

   function Constructor(address _bettor, string _result) public payable{
      bettor=_bettor;
      result=_result;
   }

   function __callback(bytes32 myid, string _result) public{
      if (keccak256(abi.encodePacked(result)) ==
keccak256(abi.encodePacked(_result))) {
```

```
            bettor.transfer(address(this).balance);
        }
    }

    function updatePrice() payable public{
            provable_query(60, "WolframAlpha", "flip a coin");
    }
}
```

The above contract uses the provable service by inheriting the usingProvable class. The provable query is present in the function updatePrice(), which queries the WolframAlpha computing engine with the phrase ‘flip a coin’ recursively every sixty seconds until it runs out of gas. The provable contract stores the query, the data source and the scheduled time mentioned within the provable_query() function. E.g., *60, "WolframAlpha" and "flip a coin".* The parameter *‘60’* denotes time interval in seconds. The operator in charge of the provable service checks the provable contract for data requests made by contracts using the provable service, by calling functions that access the state of the contract, from an externally owned account (EOA). Provable maintains a centralized server to access the query requests on the blockchain and relay them to their respective data sources, in our case ‘*WolframAlpha*’. The response is returned by the server to the provable contract (on-chain), which then passes the response to the relying contract using a __callback() function. Provable uses the __callback() function defined in the contract to pass in the response to the contract. Given below is a figure that depicts the request flows of a smart contract that uses the Provable service.

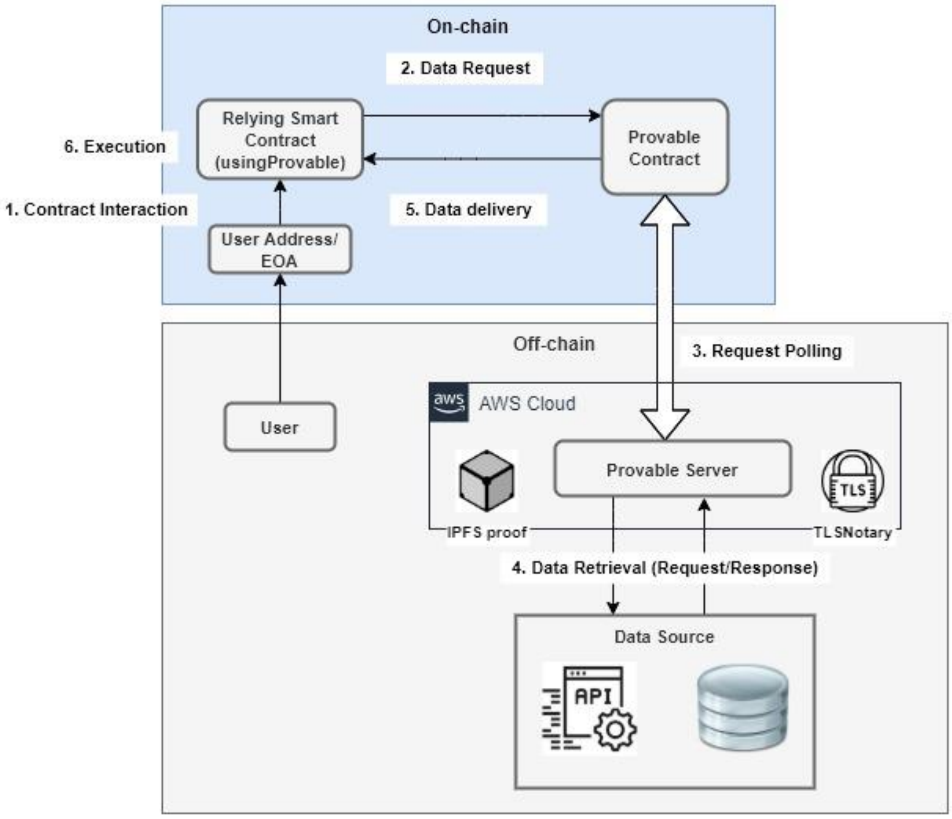


Figure 10: Request flows in Provable

3.2.2 TownCrier

Another centralized oracle system is the Town Crier. Unlike Provable which uses authentication proofs, TownCrier [8] uses a trusted execution environment to guarantee that the data received from the source is not tampered with. It provides an authenticated data feed for smart contracts on the Ethereum blockchain, by creating a secure channel between the https-enabled data source and the relying contract. It requests data from a website and sends it to the relying contract in the form of datagrams. It uses Intel Software Guard Extensions (SGX), a trusted hardware capability to execute the core functionality in an SGX enclave. The enclave protects the code which constitutes of the core functionality against malicious processes by defining private regions of memory. It also able to provide attestations or proofs to remote clients of their interaction with a secure SGX-backed instance of Town Crier code.

The three main components of Town Crier are the Town Crier contract, the Enclave and the Relay (See figure 10). The Town Crier contract resides on the blockchain, whereas the Enclave and the Relay are present within a centralized server managed by

Town Crier. The contract on the blockchain that uses Town Crier's services is referred to as the relying contract or requester. The data source is a website or a server that Town Crier communicates with, over a secure https channel.

Town Crier Contract (TC contract)

It is a smart contract which acts as an interface between the relying contract and the data source. It is designed to provide a simple API to relying contracts on the blockchain, for its requests to the Town Crier service. The TC contract accepts datagram requests from relying contracts in the blockchain and returns the response provided by the Town Crier service. The TC contract account is responsible for managing the funds of the oracle system.

Enclave

The Enclave is the trusted execution environment where the core code that implements the functionality of Town Crier is run. It ingests and fulfils the requests forwarded by the TC contract. It fulfils requests by querying external data sources through secure https communication and returning the datagram as a digitally signed blockchain message, by means of a middleware called the *Relay*. The Enclave guarantees security by running code in complete isolation from all the external processes, even the host operating system.

Relay

The Relay in Town Crier is a system that enables communication between the Enclave and external data sources such as a website or an API. It serves as a bridge between the Enclave and the external entities such as the blockchain, data sources and clients. It polls the blockchain to monitor the state of the TC contract. A web server is run by the Relay to serve off-chain requests from clients such as requests for attestations. Finally, the Relay also manages the traffic between the Enclave and external data sources.

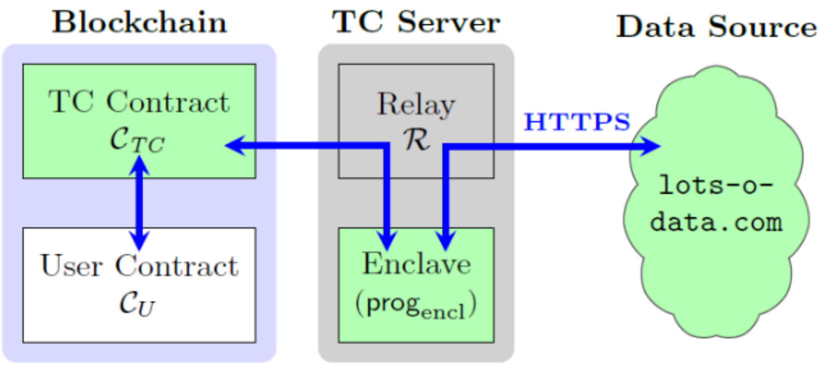


Figure 11: Town Crier Architecture

3.3 Decentralized Oracles

Unlike centralized oracles, there is a subclass of blockchain oracles that allow equal privileges to participants who report external data required by the contract, called decentralized oracles. Here, any user may participate as reporters and all participants have identical privileges (i.e., without bias). A decentralized oracle model fundamentally consists of an oracle operator, proposition creators, voters and a consensus mechanism. [5]

Oracle Operator

To achieve decentralization the consensus operation of the oracle system is implemented on a decentralized platform, such as Ethereum or Hyperledger. This entity is referred to as the operator, which maintains a list of created propositions and their votes, required to decide on the correct outcome. A user may submit new propositions to the operator, after which several users vote on an active proposition. After a certain duration the active propositions are closed the votes are no longer accepted from users. The total votes are then counted, and the rewards distributed by the operator.

Proposition Creators

Proposition creators are those entities that are concerned with the right data being provided to the smart contract on the blockchain. They choose propositions that are to be included in the oracle system, by allocating funds for the subsequent effort of transferring correct data on to the blockchain.

Voters

A decentralized oracle takes inputs from users (either humans or other data sources such as an API). These users who input data into an oracle are referred to as voters and any number of voters could join the system to provide their inputs. Suppose V = {v1, v2, v3, ..., vn} be the set of all voters who participate in the decentralized oracle protocol. For a proposition pj , each voter vi ∈ V supports an outcome Oij. Voters who collude to support a false outcome are considered adversaries and those that vote the same outcome for each and every proposition are considered lazy voters.

Consensus Mechanism

A proposition is randomly selected and assigned to all voters, who vote for a particular outcome. Finally, when the proposition is closed, all the votes and outcomes are tallied to find the outcome that has majority of the votes. The outcome with the majority of votes is reported as the correct outcome by the oracle.

Some of the decentralized oracle platforms that exist today are Astrea, Augur, Chainlink and Gnosis. All the aforementioned oracles essentially include the components – an operator, proposition creator, voters and a consensus mechanism as described previously. They differ from each other, in the steps taken by them to achieve consensus, authentication (proofs) and nash equilibrium. Nash equilibrium is a concept in game theory where no player gains when deviating from their initial strategy. [25] Each of these oracles have different trust models and each trust model is suited for different businesses or use cases. Let us now consider the design and working of various decentralized oracle.

3.3.1 Astrea

Astrea [26] is a decentralized oracle service that implements a voting-based consensus mechanism to arrive at the truth and falsity of propositions. Players of the game fall into two categories: voters and certifiers. Certifiers play a high-stakes game and voters put lesser at stake. The rewards of the game are proportional to the risk (stake) involved in the game. Parameters of the game are set such that Nash equilibrium is maintained and that all rational players behave honestly. Users of Astrea assume one of the following roles: submitters, voters and certifiers.

Submitters

Submitters add propositions to the oracle system together with a bounty, which is later distributed to all the users who were responsible for providing the correct outcome for the propositions. They are usually entities that require critical information or data, in order to trigger certain functions within their smart contracts.

Voters

They are players who have roles that accompany low-risk and low-rewards. They do not have to deposit large stakes in order to cast their vote, which is not the case with certifiers. The oracle system requires a stake to be deposited even before the voters are notified of the proposition they will be voting on. A proposition is chosen at random and is provided to the voter who has staked a deposit. The sum of votes weighted by their corresponding deposits, determines the result of the voting process. There is also a limit set on the maximum deposit a voter can stake.

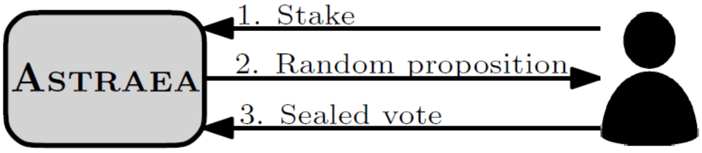


Figure 12: Voting in Astrea

Certifiers

Certifiers play a game of high risk and high rewards. Unlike voters, they get to choose a proposition from a list of propositions, by placing a large deposit to certify it as either true or false. The outcome of the process for a proposition is the sum of its certifications weighted by their deposits. The deposit size of certification is a parameter that is set to be large enough, that certifying may involve substantial risk. Not all propositions have certifications and the oracle does not compel submitters to include them either.

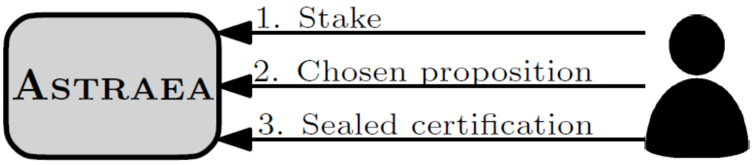


Figure 13: Certifying in Astrea

Working of Astrea

A critical element in the working of Astrea is the proposition list P, which contains a set of propositions. Each proposition in the list is denoted by $p_i \in P$ and has a truth value t_i, which is associated with a bounty B_i. The system also maintains two certifier reward pools R_T and R_F that contain rewards to be distributed to certifiers belonging to the winner universe. The purpose of using two separate reward pools is to prevent a corrupt co-ordinated strategy where both voters and certifiers support a constant outcome true or false, in order to maximize their gains without expending any effort. Each time a proposition is decided with voting outcome T, an amount $R_T/\ \tau$ is subtracted from R_T. If the certification outcome is also T, the funds are used to pay out certifier rewards. Otherwise, the funds are added to R_F. Here 'τ' denotes certification target, which is the number of certifications that the reward pool should have enough funds to pay for.

For example, consider a proposition where $R_T = 1000$, $R_F = 1000$ and $\tau = 10$. If the voting outcome was true and certifying outcome matched the voting outcome, then $R_T/\ \tau$ i.e., 100 monetary units will be distributed to certifiers who supported the winning outcome. If the certifying outcome did not match the voting outcome 100 monetary units were transferred to the R_F reward pool and the next proposition to be decided would have reward pools $R_T = 900$ and $R_F = 1100$. And, therefore 90 units are distributed to certifiers who support the false outcome and 110 units to those who support the true outcome. In this way certifiers are incentivised to support a false outcome and prevents 'lazy certification' i.e., supporting a constant outcome.

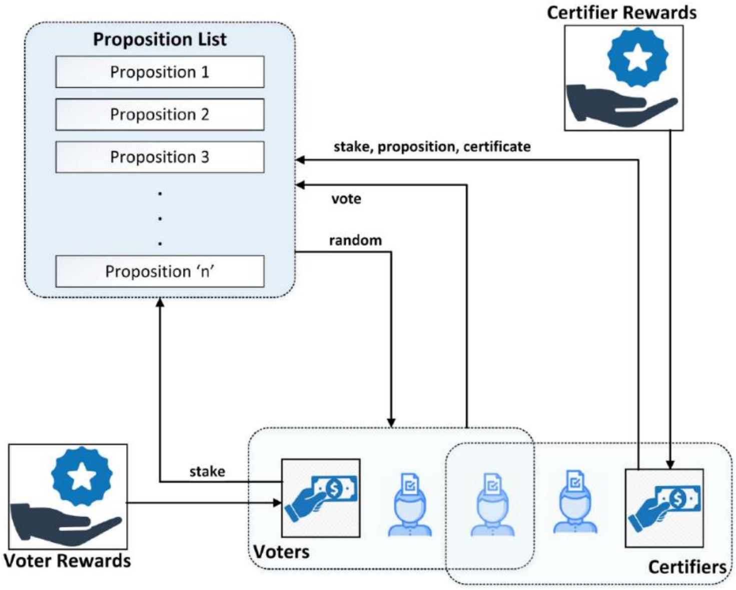
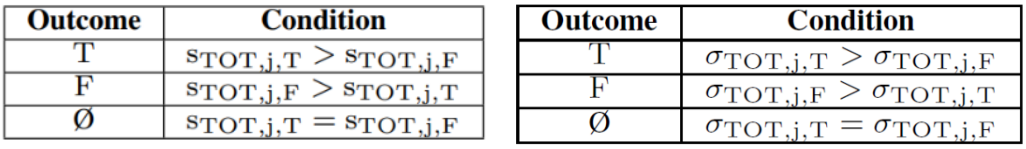


Figure 14: Architecture of Astrea

The different variables used by the oracle system are – $s_{i,\ j,\ b}$ denotes the amount staked by a player i on voting for proposition p_j and b is either true or false (T/F), s_{max} and σ_{min} are parameters that denote maximum voting stake and minimum certifying stake respectively. When voting, users stake a deposit that is less than the value of s_{max} i.e., $s_{i,\ r,\ v} <= s_{max}$ on a vote for proposition p_r , where r is not yet known to the voter. The value of r is chosen at random from the list of propositions. Users certify a proposition by submitting a stake $\sigma_{i,\ j,\ c} >= \sigma_{min}$, where c denotes a sealed certification c ∈ {T, F} for proposition p_j. At a certain time, when the proposition p_j has accumulated sufficient voting stake, it is decided and values $s_{TOT,\ j,\ T}$, $s_{TOT,\ j,\ F}$, $\sigma_{TOT,\ j,\ T}$ and $\sigma_{TOT,\ j,\ F}$ are calculated. For b ∈ {T, F}, the values are computed as follows:

$$s_{TOT,j,b} = \sum_{i=1}^{N} s_{i,j,b} \qquad \sigma_{TOT,j,b} = \sum_{i=1}^{N} \sigma_{i,j,b}$$

Once the values are computed, the voting and certifying outcomes are determined using the following logic.



Outcome	Condition
T	$s_{TOT,j,T} > s_{TOT,j,F}$
F	$s_{TOT,j,F} > s_{TOT,j,T}$
Ø	$s_{TOT,j,T} = s_{TOT,j,F}$

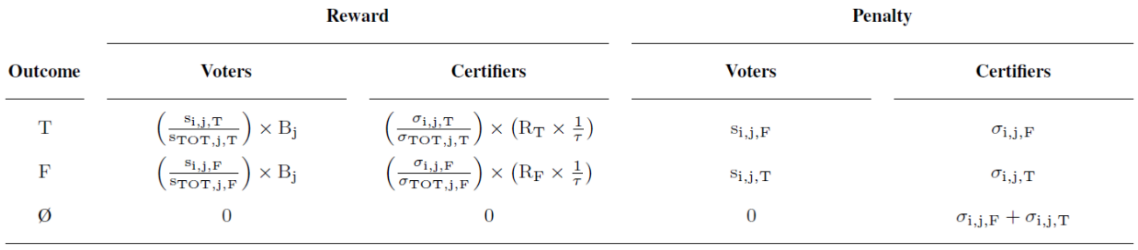
Outcome	Condition
T	$\sigma_{TOT,j,T} > \sigma_{TOT,j,F}$
F	$\sigma_{TOT,j,F} > \sigma_{TOT,j,T}$
Ø	$\sigma_{TOT,j,T} = \sigma_{TOT,j,F}$

Table 1: Voting and Certifying Outcomes

We observe in Table 1 that a simple majority determines the outcome. Alternatively, the oracle system may also require a higher majority instead of a simple majority for true and false outcomes and otherwise remain undecided (Ø). The game has 3 possible outcomes – true, false and unknown/undecided. The outcomes of propositions are used in the distribution of rewards and penalties. The suggested oracle output for a proposition p_j follows the voting outcome if it matches the certification outcome or the certification outcome is undecided. The output remains undecided in two scenarios – one where the certifying outcome is either true or false and doesn't match the voting outcome; the other is when both voting and certifying outcomes are undecided. Also, the oracle does not necessarily have to output a value which is either 0 or 1, but it can also output a value in the range [0, 1] which denotes the level of confidence in the truth or falsity of the proposition p_j. For example, a proposition can be set to have a true outcome if its level of 'truth' confidence is '0.4' or 40 percent.

Rewards and Penalties

In case of an outcome which is either true or false, a player's vote reward for a proposition p_j is the proportion of their stake times the bounty submitted for p_j. The penalty given is the amount staked for an outcome that does not correspond to the oracle's final outcome. When the outcome is undecided (Ø), voters are not penalized whereas certifiers are penalized. The rationale behind penalizing certifiers and not voters is that certifiers get to choose their propositions, while voters are given one at random. The rewards and penalties for proposition p_j are shown in table 2 below.



	Reward		Penalty	
Outcome	**Voters**	**Certifiers**	**Voters**	**Certifiers**
T	$\left(\frac{\mathrm{s_{i,j,T}}}{\mathrm{s_{TOT,j,T}}}\right) \times \mathrm{B_j}$	$\left(\frac{\sigma_{\mathrm{i,j,T}}}{\sigma_{\mathrm{TOT,j,T}}}\right) \times (\mathrm{R_T} \times \frac{1}{\tau})$	$\mathrm{s_{i,j,F}}$	$\sigma_{\mathrm{i,j,F}}$
F	$\left(\frac{\mathrm{s_{i,j,F}}}{\mathrm{s_{TOT,j,F}}}\right) \times \mathrm{B_j}$	$\left(\frac{\sigma_{\mathrm{i,j,F}}}{\sigma_{\mathrm{TOT,j,F}}}\right) \times (\mathrm{R_F} \times \frac{1}{\tau})$	$\mathrm{s_{i,j,T}}$	$\sigma_{\mathrm{i,j,T}}$
Ø	0	0	0	$\sigma_{\mathrm{i,j,F}} + \sigma_{\mathrm{i,j,T}}$

Table 2: Summary of Rewards and Penalties

The voter's reward is a proportion of their share of the voting stake times the bounty reward and their penalty is equal to their stake on the opposite outcome. Whereas the certifier's reward is a proportion of their share of certifying stake on a proposition outcome times its corresponding reward pool amount (R_T or R_F), times the reciprocal of the certification target 'τ'. The penalty imposed on certifiers is their stake on the losing outcome.

3.3.2 Chainlink

Chainlink is another decentralized oracle developed by Chainlink Ltd SEZC whose architecture basically consists of two sets of components – on-chain and off-chain components. The on-chain component makes requests on behalf of a user contract, which is also a smart contract that bridges the requesting contract with the off-chain data source. Besides being an interface, the on-chain component also implements consensus mechanisms required by a decentralized oracle system. The off-chain component consists of a network of oracle nodes that connect to the blockchain network. They harvest responses to the requests made off-chain, which is relayed to the on-chain component that aggregates them into a single response before being sent to the relying contract. Let us now discuss the architecture in detail.

On-chain Architecture

The on-chain component of Chainlink[28] comprises of three contracts: a reputation contract, an order-matching contract and an aggregating contract. The reputation contract keeps track of the performance of oracle nodes that perform off-chain requests on behalf of the on-chain contract. The order-matching contract collects a service level agreement (SLA), logs their parameters, and collects bids from the oracle nodes (data providers). An SLA consists of parameters specified by the relying contract such as the oracles to be selected, data sources etc. The order-matching contract then uses the data present in the reputation contract to select bids that best match the SLA. The aggregating contract collects the responses from oracle providers and outputs the final result of the Chainlink query. It also forwards feedback on oracle providers to the reputation contract, which is used later for bid selection. The on-chain workflow has three stages: oracle selection, data reporting and result aggregation.

1) Oracle Selection

The relying contract that purchases services from the oracle specifies requirements that constitute the SLA proposal. The proposal has details such as query parameters and the number of oracles needed by the purchaser. It also specifies the reputation and aggregating contracts to be used by the agreement. Using logs that are maintained on-chain, purchasers are able to view, filter and select oracle providers based on their reputation, via an off-chain interface.

Once the purchaser submits the SLA, it is forwarded to the order-matching contract. This triggers a log that oracle providers can use to view, filter and select currently available SLAs based on their capabilities. Oracle service providers (Chainlink nodes) view the SLAs and bid on an order. They commit to an order by attaching a penalty amount that would be lost if they did not stick to the conditions laid down in the SLA. The bidding window is closed when sufficient number of qualified bids have been received. The requested number of oracles is then selected from the pool of bids. And, penalty payments of oracles that were not selected are returned to oracle providers (or nodes).

2) Data Reporting

The off-chain oracle providers execute the requests using the parameters mentioned in the SLA and relay the response results of the request to the oracle contract on-chain.

3) Result Aggregation

After the oracle providers have relayed their results to the oracle’s on-chain contract, they are in turn fed to the aggregating contract. The aggregating contract collects results to output a weighted answer, which is sent to the requesting contract. Also, the validity of the data provided by each off-chain oracle is forwarded to the reputation contract. Chainlink also allows purchasers to use customized contracts for aggregation, provided they follow the required standards.

Off-chain Architecture

Off-chain components consists of a network of oracle nodes that connect to the blockchain. They make requests on behalf of the requesting contracts and forward the results on-chain. The Chainlink oracle nodes run a standard open source core implementation capable of handling standard blockchain interactions and making requests to external resources such as an API or a database. Node operators may also choose to add custom code via software extensions called external adapters. Let us now discuss each of them in detail.

1) Chainlink Core

The core node-software of Chainlink, equips the oracle providers (nodes) with functionality required to interact with the blockchain, schedule and balance workload across its services. The work performed by the nodes is divided into sub-tasks and is processed as a pipeline to reach a final result. The core node-software comes with built-in subtasks such as making http requests, JSON parsing, and conversion of data to blockchain formats.

2) External Adapters

Custom sub-tasks can be defined by node operators by creating adapters. External adapters refer to services provided by a simple API. A complex interaction with multi-step APIs can be divided into simpler sub-tasks with parameters, using adapters.

3) Subtask Schema

With different developers maintaining varied types of adapters, it is important that they follow some type of standard to ensure compatibility. Chainlink implements a JSON schema which specifies what inputs an adapter needs and how they need to be formatted. In addition, adapters also specify a schema that describes the output format of each subtask.

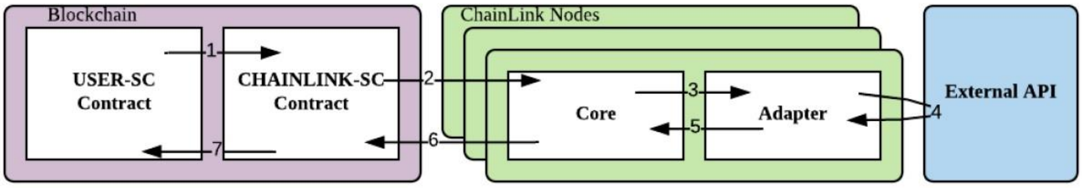


Figure 15: Workflow of Chainlink

Request Workflow

When a relying contract ‘USER-SC’ makes an on-chain request, it is forwarded to CHAINLINK-SC which logs an event, which is then viewed by oracle providers. The core code present on the oracle nodes retrieves those events and forwards them to the adapter. The ChainLink adapter makes a request to an external API using the parameters specified in the event forwarded by the adapter. The ChainLink adapter then processes the response and sends it back to the core, which relays the data to the on-chain oracle contract ‘CHAINLINK-SC’. The responses provided by the oracle nodes are combined into a single response and sent to USER-SC.

Consider the following example –

```
pragma solidity 0.4.24;
import "@chainlink/contracts/src/v0.4/ChainlinkClient.sol";

contract MyContract is ChainlinkClient, Ownable {
        function createRequestTo( address _oracle, bytes32 _jobId, uint256 _payment,
        string memory _url, string memory _path, int256 _times)
        public onlyOwner returns (bytes32 requestId){
                Chainlink.Request memory req = buildChainlinkRequest(_jobId,
                address(this), this.fulfill.selector);
                req.add("url", _url);
                req.add("path", _path);
                req.addInt("times", _times);
                requestId = sendChainlinkRequestTo(_oracle, req, _payment);
        }

}
```

The above function *createRequestTo()* uses function *sendChainlinkRequestTo()* to log an event containing the details of the request such as the *jobId*, url, *path*, *times*, *oracle* and *payment*. The *jobId* is used to track the requests, the *url* contains the URL to fetch data, *path* is the dot-delimited path to parse the response and *times* is the number to multiply the result by. The *sendChainlinkRequestTo()* function takes parameters - *oracle*(the address of the oracle), *req*(the request body) and *payment*(payment for gas and the service), which logs an event in the TownCrier contract. The core code present in each oracle collects the events that specify the oracle's address in the '*oracle'* field and forwards the request to the adapter which makes an API request. The adapter processes the response and sends it to the core, which then relays the data to the oracle contract. If more than one request is made by the user contract, then different responses are aggregated into a single response by the oracle contract and sent to the user contract.

Decentralization in Chainlink

Chainlink creates a trust model by following three approaches – distribution of data sources, distribution of oracles and use of trusted hardware. The oracle achieves decentralization by including several data sources and oracle providers.

1) Distribution of Data Sources

A single data source makes the oracle service less reliable and becomes a single point of failure. The solution to a single faulty data source is to collect data from multiple sources and aggregate them into a single response. An oracle provider may query a collection of data source s_1, s_2, …, s_k to obtain responses r_1, r_2, … r_k and aggregate them into a single answer 'A'. Aggregation can be done by checking if the majority of sources return an identical value for a particular request. The function which aggregates the responses, could be modified to make the system robust against erroneous data. For example, calculating the mean after discarding outliers or implementing an n out of k scheme where, at least n out of k providers submit identical value for a particular request.

2) Distribution of Oracles

As the use of several data sources increase the robustness of the oracle system, so does the concept of distribution of oracles. Instead of using a single oracle to report to the oracle contract on-chain, the purchaser can require a set of oracles to collect data from an external source and pass them to the blockchain. Each oracle contacts its own set of data sources, which may or may not overlap with those of other oracles. The responses returned by all the oracles are aggregated into a single authoritative value by the oracle's aggregating contract, which is present on the blockchain.

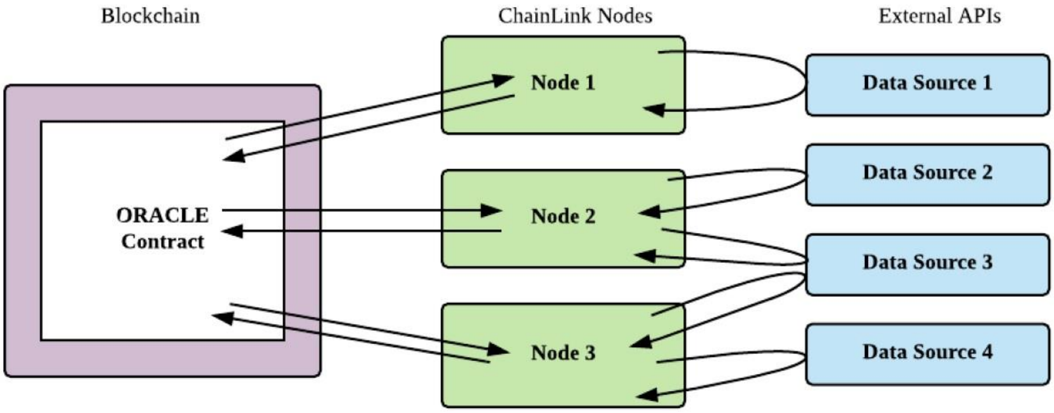


Figure 16: Request Distribution (among oracles and data sources)

Reputation System

As discussed, the reputation contract stores details related to the performance of each oracle. Purchasers of the oracle system are able to view the performance metrics of oracle providers and take a well-informed decision. The reputation system essentially monitors the oracles for availability and correctness. It records the failures of an oracle to respond to the request in a timely manner and incorrect responses measured by deviations in responses from those of its peers. The reputation system logs the following metrics: total number of assigned requests, total number of completed requests, total number of accepted requests, average time to respond and amount of penalty payments. It incentivizes oracle providers to behave correctly and also ensure high availability. Both negative user feedback and penalties significantly affect the brand value of the oracle system. Thus, it creates a mechanism where well-functioning oracles secure good ratings and consequently are chosen by several relying smart contracts to be their data provider.

3.3.3 Augur

Augur [29] is a decentralized oracle and prediction market platform where individuals speculate on the outcome of certain events. The participants who predict the outcome correctly earn rewards while others lose them. An Augur market is resolved in four stages: creation, trading, reporting, and settlement. Each phase of progression in the Augur market is critical to the security and performance of the system. The platform has a token called REP (reputation), which is used by market creators and reporters when they create markets and report on the outcomes. Augur contracts are completely automated and redistributes the REP staked by reporters who supported the losing outcome (or outcomes) to those who reported supporting the winning outcome. The reward received is proportional to the amount of REP staked by reporters.

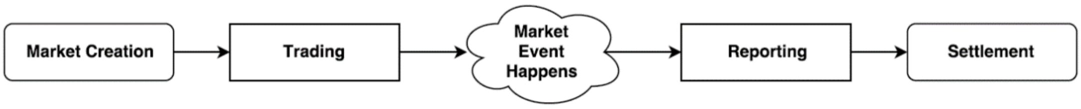


Figure 17: Different Phases in Augur

1) Market Creation

Any user can create a market in Augur for an upcoming event. When creating a market, the user stakes a deposit and sets several parameters. The market creator sets the event end time and a designated reporter who is responsible for reporting the outcome of the event. The market creator also chooses a resolution source, that reporters must use to settle on the outcome. E.g., bbc.com. Besides this, market creators also post two bonds: the validity bond and the designated reporter no-show bond. The validity bond is staked in Ether, which incentivizes market creators to create well defined markets that have unambiguous outcomes. The size of the validity bond is determined dynamically by Augur, which is essentially a proportion of the invalid outcomes in recent markets. On the other hand, the no-show bond encourages market creators to choose a reliable designated reporter. It is also used to cover the gas costs of the first public reporter, provided they report correctly.

2) Trading

In this phase, users trade shares of market outcomes so as to predict the outcome of events. The automated matching engine of the Augur contract fulfils the trading requests. Request are fulfilled as complete sets by either issuing new complete sets or closing out existing complete sets. A complete set of shares is a collection of shares that consists of one share of each possible valid outcome of the event. For example, consider a market that has two possible outcomes A and B, and Alice and Bob are traders pay a sum of 0.3 ETH and 0.7 ETH to purchase shares for outcomes A and B respectively. Augur matches the orders placed by Alice and Bob, and collects a sum of 1ETH from each of them to create a complete set of shares, giving Alice the share of A and Bob the share of B. Orders are never executed at a worse price than the minimum price set by the seller and unsuccessful trades can be withdrawn by participants at any time.

3) Reporting

When an event belonging to a market has occurred, the outcome of the event must be determined. The outcome is decided using a decentralized oracle system, which consists of profit motivated reporters who report the actual outcome of the event. Reporters who report correctly are awarded i.e., reports that are consistent with the final outcome and others are penalized. The reporting process can be divided into 6 different stages: designated reporting, open reporting, waiting for the next fee window, dispute round, fork and finalized.

Designated Reporting

Once the event end time has passed, the market enters the designated reporting phase, where the designated reporter specified during market creation has up to three days to report on the outcome of the market’s underlying event. If the designated reporter fails to report within the three-day period, the market creator forfeits his/her no-show bond and the market enters the open-reporting phase. And, if the designated reporter reports on time, the no-show bond is returned to the market creator. When reporting, the designated reporter is required to stake a deposit which they forfeit if the market finalizes to an outcome different from the one, they reported. After the designated reporter submits a report, the market enters the ‘waiting for the next fee window’ phase.

Open Reporting

If the designated reporter fails to report at the right time, the market enters the open reporting phase and any user can report on the outcome. Also, the market creator forfeits the no-show bond, because the designated reporter was not reliable. The user who first reports on the outcome is called the market's first public reporter, who receives the no-show bond as a reward for the report they submitted. They may claim the reward only when the market's final outcome coincides with the outcome they reported. The first public reporter does not have to stake any REP to report on an outcome, which incentivizes participants to report quickly, after the market has entered the open reporting phase.

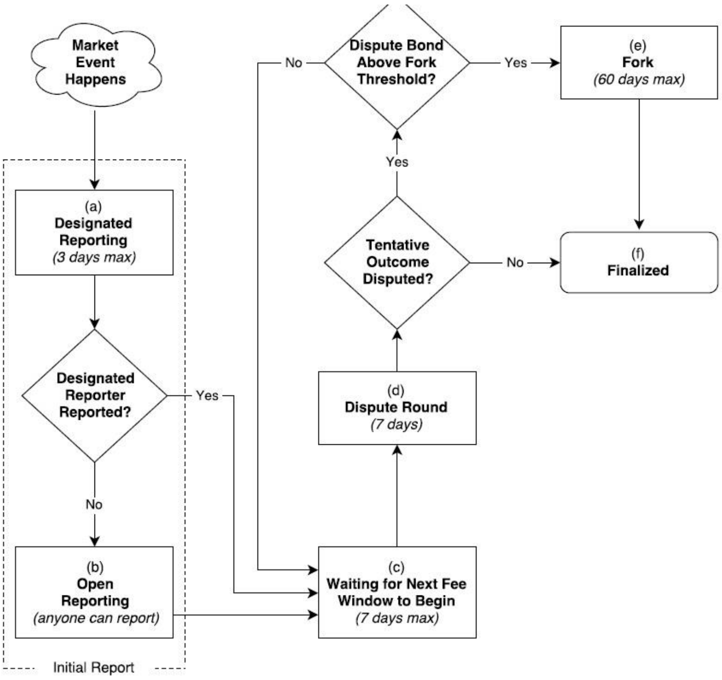

Market Event Happens
(a) Designated Reporting (3 days max)
Designated Reporter Reported?
Yes
No
(b) Open Reporting (anyone can report)
Initial Report
(c) Waiting for Next Fee Window to Begin (7 days max)
(d) Dispute Round (7 days)
Tentative Outcome Disputed?
No
Yes
Dispute Bond Above Fork Threshold?
No
Yes
(e) Fork (60 days max)
(f) Finalized

Figure 18: s's Reporting Flowchart

Waiting for the Next Fee Window

Once the first report has been submitted, the reporting process is on hold until the end of the current fee window. When the current window period ends, the disputing phase begins.

Dispute Round

The dispute round is a 7-day interval during which users holding REP tokens have the privilege to dispute the market's tentative outcome. The tentative outcome of the market becomes the final outcome if it is not successfully disputed in the 7-day period. A successful dispute is one where a REP token holder deposits a stake on an outcome other than the market's tentative outcome. The size of the deposit to be staked in order to successfully dispute the tentative outcome in favour of a new outcome w during round n is denoted by B(w, n) and is given by:

$$B(w, n) = 2A_n – 3S(w, n)$$

Where A_n denotes the total stake on all the market's outcomes at the beginning of round n and S(w, n) is the total stake on outcome w at the beginning of round n. The size of the deposit is chosen this way to ensure a fixed ROI of 50 percent for reporters who successfully dispute false outcomes. For example

Augur also permits collaboration among participants to stake deposits and dispute an outcome. If an outcome is successfully disputed, then the market either enters another dispute round or a fork state. The market enters a fork state if the total size of the dispute bond is greater than 2.5 percent of all the REP available in the system. If the 7-day dispute round ends without a successful dispute, then the market enters the finalized state. The final outcome of a market is the tentative outcome that clears the dispute round without a successful dispute, or the outcome arrived at after a fork. Based on the final outcome, the augur contract distributes the rewards to the reporters who supported the final outcome.

Fork

The fork state lasts for a total of 60 days and occurs when a market is successfully disputed with a stake of more than 2.5 percent of all the available REP. Forking creates a new child universe for each outcome that is successfully disputed. E.g., a binary forking market has three 3 child universes: universe A, universe B and universe Invalid. A fork permanently locks the parent universe and there after no new markets are allowed to be created until the underlying market has resolved. During the fork period holders of REP token migrate to the universe of their choice. Also, reporters who staked for a particular outcome in the parent universe cannot alter change positions during migration. The universe that receives the most migrated REP at the end of the forking interval becomes the winning universe and its corresponding outcome becomes the final outcome of the market. Unfinalized markets are only allowed to migrate to the winning universe. The system provides 5 percent additional tokens to all token holders who migrated their REP during the forking period, to encourage greater participation.

Finalized

A market enters the finalized state if it passes through a 7-day dispute round without a successful dispute or on the completion of a fork. The outcome of a fork is considered final and cannot be further disputed. The traders settle their positions after their markets enter the finalized state.

4) Market Settlement

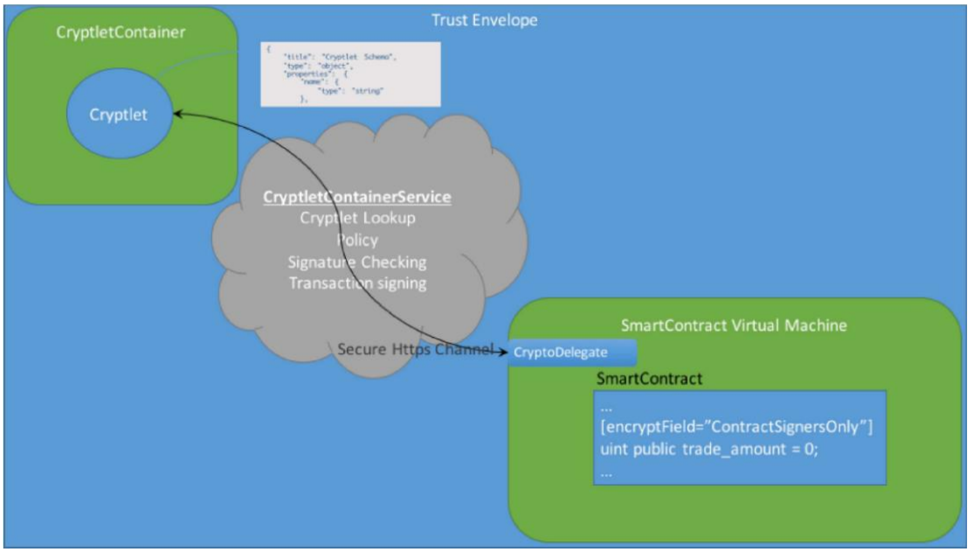
In this phase, traders close their positions by either selling their shares to another trader in exchange for currency or by settling their shares in the market. Augur levies fees such as the creator fee and the reporting fee, on participants during settlement. All the REP staked on an outcome other than the final outcome is forfeited and redistributed to participants who supported the final outcome, in proportion to the amount of REP they deposited.

Let us now consider an example where Charlie creates a market ‘Who will be the 2020 Republican Nominee for president?’ on 14-07-2020 and deposits a validity bond and a no-show bond. He specifies the end time to be 14:00 hours, 16-07-2020, deposits a validity bond of size 5 REP as required by Augur, a no-show bond of 5 REP and a designated reporter ‘R_1’. Traders begin trading shares on possible outcomes. E.g., Bernie Sanders or Donald Trump. The current share price of an outcome is the probability of that outcome when compared to other outcomes. Assume, the current probability of Bernie Sanders winning the election is 0.3 Alice wants to purchase a share of the outcome ‘Bernie Sanders winning the election’ and Bob wants to purchase a share of ‘Donald Trump winning the election’. Alice and Bob would have to pay 0.3 ETH and 0.7 ETH respectively to make the purchase. Similarly, a total of 100 complete sets of shares were purchased by various participants, i.e., value of shares traded was 100 ETH. After the end time has passed the market moves to the designated reporting phase and waits for a period of 3 days for D to report on the actual outcome of the event. R_1 reports Bernie Sanders as the actual outcome within 3 days by staking a deposit of 2 REP. After 3 days the market waits for the 7-day dispute round to begin. During the dispute round, reporter R_2 reports ‘Bernie Sanders’ with a stake of 1 REP. The market’s tentative outcome remains ‘Bernie Sanders’, as reported by D and passes through the dispute round without being successfully disputed and becomes the final outcome. Traders settle their positions directly with the market. In our case Alice has a share of the winning outcome ‘Bernie Sanders’ and wants to cash it in. She sends her share to Augur and in return receives 1 ETH minus the reporting and market creator fees. Reporting fees are set dynamically by Augur. For convenience sake let us assume a reporting fee of 5 percent of the traded value. Since our market has traded 100 ETH worth of shares the reporting fee pool has 5 ETH. The reward R_1 receives is 2/3 times the total amount in the fee pool, i.e., 3.3 ETH approximately and R_2 receives one-third of the total amount, i.e., 1.3 ETH approximately.

3.4 A Summary of Oracles

In addition to the above-mentioned oracles, we also have Gnosis, MS Bletchley and Corda. Gnosis is a prediction market platform that leverages the capability of blockchain oracles using features of both centralized and decentralized oracles. It is a flexible service that provides options to use data sources or human inputs as oracles providers, which can also be disputed by participants. It implements a reputation system which rates oracle providers based on their performance (E.g., correctness of the data provided). ‘Ultimate Oracle’ is decentralized consensus mechanism adopted by Gnosis as a last resort, which is essentially creating a prediction market for the outcome of the prediction market. But the ultimate oracle may only be triggered if a participant deposited a high stake of 100 ETH. [24]

Both MS Bletchley and Corda [32] use a trusted hardware to fetch data from an external trusted execution environment. But the difference is in that MS Bletchley uses multiple oracles to fetch data from several sources, whilst Corda implements only a single oracle. Multiple oracles and sources make the oracle system more reliable than those with single oracle mechanisms. They are implemented in MS Bletchley [34] using Cryptlets, called by CryptoDelegate which is a function 'hook' within the smart contract that uses the oracle service. Cryptlets are full delegation engines that function on behalf of the smart contract off the chain. They are bound to their smart contract and created on the fly when the smart contract is deployed to the blockchain. Multiple oracles prevent single points of failure. We also find a similarity between Chainlink and MS Bletchley in that both use multiple oracle providers. However, MS Bletchley uses automated redundant oracles to fetch data while Chainlink uses redundant human oracles.



```
{
    "title": "Cryptlet Schema",
    "type": "object",
    "properties": {
        "name": {
            "type": "string"
        },
```

```
...
[encryptField="ContractSignersOnly"]
uint public trade_amount = 0;
...
```

Figure 19: Cryptlets in MS Bletchley

Oracles are implemented in Corda [31] using commands or attachments. Commands encapsulate a specific fact (e.g., price of ETH in dollars) and lists an oracle as the required signer. If a smart contract wishes to use a given fact for its execution, they request or query a command asserting that fact from the oracle. The oracle then sends the command together with an attestation (or signature) on-chain. Only the commands that require a signature of the oracle are made visible to the oracle entity and the rest are not. Implementation of attachments consist of establishing facts by sending a query to the oracle and a response is received that contains a signed attachment. An attachment is a separate object and is referred to by a hash. They can be reused in other transactions without involving the oracle again. The below table provides a summary of blockchain oracles.

Platform	Oracles	Consensus	Trust Model / features	Data Source(s)	Latency	Types of Oracles
Provable	single	N/A	TLS Notary, IPFS multihash	single	low	Provable Contract
TownCrier	single	N/A	Intel SGX, proofs	single	low	TownCrier Contract
Corda	single	N/A	Intel SGX	single	low	Corda code
MS Bletchley	multiple	N/A	Secure Container, Intel SGX	multiple	low	Off-chain code
Astrea	multiple	Voting/ Certification	Reward Pools, Penalties,	multiple	high	Certifiers/ Voters
ChainLink	multiple	n-of-k multi-signature	Off-chain aggregation, Reputation	multiple	high	Reporters
Augur	multiple	Voting	Reputation, Dispute windows, Forks	multiple	high	Voters
Gnosis	multiple	Voting	Centralized oracle, Ultimate oracle	multiple	high	Voters

Table 3: A Summary of Blockchain Oracle Mechanisms

The selection of oracles always involves a trade-off between reliability and response time. Different use cases require different oracle mechanisms to be implemented. Decentralized oracles that use multiple oracles and data sources need to aggregate the responses into a single output/outcome, which is a time-consuming process. Having a greater number of oracles reporting an outcome would make the system more reliable but might incur a higher cost to be paid for all oracles to work. [30] Although a centralized oracle is faster, it becomes a single point of failure and thus less reliable. Using several mechanisms to ensure security often makes the process lengthy and slows down consensus. Oracles that use humans to report data may be less reliable than those that use data sources such as an API or a website. Gnosis [33] uses a combination of centralized and decentralized mechanisms, which could take a lot of time to reach consensus. The use of a trusted execution environments involves trusting a third-party hardware or software which diminishes the value created by the trustless service, i.e., the blockchain. Therefore, it is imperative that oracles be designed to be both efficient and reliable. [23]

3.5 Use Cases

Oracles are required for many real-world use cases that use blockchain. Examples where data may be provided on-chain using oracles are:

- Random numbers E.g., to select a winner in a lottery smart contract
- Data related to natural hazards: e.g., to trigger catastrophe bond smart contracts
- Exchange rate data: e.g., for accurate conversion of cryptocurrencies to fiat currency
- Capital markets data: e.g., pricing baskets of tokenized assets/securities
- Benchmark reference data: e.g., incorporating interest rates into smart financial derivatives
- Time and interval data; for event triggers grounded in precise time measurements
- Weather data; e.g., insurance premium calculations based on weather forecasts
- Political events; for prediction market resolution
- Sporting events; e.g., betting applications
- Geolocation data; e.g., used in supply chain tracking systems
- Damage verification data for insurance contracts
- Ether market price; e.g., for fiat gas price oracles
- Flight statistics; e.g., used for flight ticket pooling

Although all oracles work on providing external data to the blockchain, it is important to understand, certain use cases rely on real-time data while others do not. Also, some smart contracts have more monetary value attached to them than others and therefore adversaries have more to gain if the oracles sending data to them are tampered with. Oracle platforms vary by their response time, security mechanisms in place and the types of oracles used to supply data on-chain. So, it is important to choose oracles based on business needs. For example, the above-mentioned use case of ‘exchange rate data’, for conversion of cryptocurrencies to fiat currency, it is important that we get real-time accurate data. Otherwise, large value conversions could result in a loss for either of the parties involved in the transaction. Same is the case with ‘time and interval data’ a slight difference could greatly impact contract conditions. On the contrary, consensus for betting applications requires more reliable data as betting contract are of large monetary value and time may not be a critical factor. Therefore, it is imperative that we design and implement an oracle that suites many of the real-world use cases.

Chapter 4

Design

Until now, we have discussed various concepts of blockchain and several blockchain oracles that currently exist. This section presents a flexible oracle that accommodates both centralized and decentralized mechanisms. The centralized oracle polls the blockchain for requests, makes request to APIs or a website and relays back the response on-chain. The decentralized component on the other hand aggregates data from designated human oracles and sends it to the blockchain. Also, if there is only a single designated reporter, then the data provided is allowed to be disputed before it is sent on-chain to the relying contract. Thus, relying contracts are given an option to choose a centralized or a decentralized oracle.

4.1 Architecture

We can divide our oracle into two components – on-chain and off-chain. The on-chain component contains an oracle contract that stores the requests made by relying contracts. The off-chain component polls requests from the oracle contract, initiates requests on behalf of the relying contract and returns the response back to the blockchain. The different parts of the oracle platform are discussed below.

Oracle Contract

The oracle contract stores the details of the requests by creating events. The events contain an array of URLs or an array of designated reporters, attribute that is to be fetched, a scheduled time and a proof type. The array may either contain URLs of type string or addresses of designated reporters. The attribute contains the field that is required by the relying contract. The scheduled time is the time interval in milliseconds, after which the request is initiated by the oracle contract. Proof type is optional and is used to generate proofs of the type specified within the request.

Relay Server

The relay server polls events generated by the oracle contract, stores the events as requests within the database and IPFS. For requests that require a centralized oracle, the server initiates a request to the URL or website and returns the response to the blockchain. If the contract requests for decentralized reporting with more than one designated reporter, then it passes the request to a private blockchain which in turn aggregates data reported and sends them to the server which relays them on-chain.

Oracle^2 (Private Blockchain)

Oracle^2 is a private blockchain maintained to store only the decentralized oracle requests and manage the consensus mechanism of the oracle. It stores the state of decentralized oracle requests by generating events and, aggregates and stores the data reported by various oracles (human oracles or any other infrastructure). Primarily, we use a private blockchain to reduce costs of computation and maintain transparency in the reporting process.

Designated Reporter

The designated reporter is either a human oracle or any other infrastructure that reports data for the requests present in the private blockchain. If the request contains more than one designated reporter and if k out of n reporters provide identical data for the requests, then the system reaches consensus and the resultant data is transferred to the relying contract. Otherwise, the outcome remains undecided and the request enters a 1-week dispute round.

REP Tokens

They are tokens of monetary value, staked by participants when reporting data. Reporters are required to back their reports with tokens, which they forfeit if their reports do not coincide with the final outcome. The Qtum blockchain provides the facility for accounts to create tokens easily.

Voters

Voters are participants of the decentralized oracle who stake a REP token deposit and vote on a particular outcome. The outcome that has the majority stake is used to decide the final outcome. Voters play a low risk/reward role, where the they stake only a small deposit for a proportionate reward. The voters are provided requests at random after they stake a deposit, i.e., they do not get to choose the requests they vote for. The oracle system sets the minimum number of voters (V_{min}) required for each request.

Certifiers

Certifiers like voters, also stake a deposit and certify an outcome. But, unlike a voter, a certifier stakes a larger deposit and plays a game of high risk and high reward. The request reaches a consensus if the outcome that has the highest certification stake follows the outcome that has the majority of voting stake. The oracle system sets the minimum number of certifiers (C_{min}) required for each request.

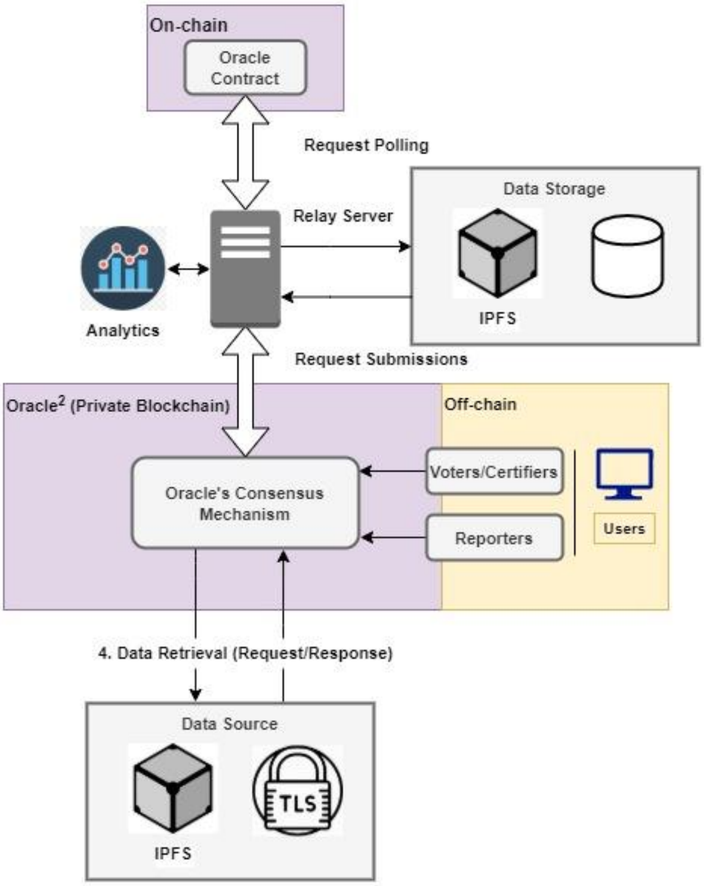


Figure 20: Architecture of the Oracle platform

4.2 Working

Request Creation

When a relying contract creates a request, it emits an event within the oracle contract which contains details such as the URL, attribute to fetch, schedule time and proof type. The relay server polls the oracle contract for events and stores the details in IPFS, a distributed file system and a stores the file’s hash in the database. In this way the contents pertaining to a request can be accessed from the IPFS using the hash stored in the database.

The format of the request is –

oracleQuery(boolean flag = true, string[ ] URL, string attribute, uint scheduledTime, string proof)

or

oracleQuery(boolean flag = false, address[ ] designatedReporter, string URL, string attribute, uint scheduledTime)

If the parameter *flag* is true then the request is for the centralized oracle and if *false*, represents a request to a decentralized oracle. In case of centralized oracles, the server initiates a request to the URL(s) or website after the time period specified within the parameter ‘*scheduledTime*’ and returns the aggregated response to the blockchain together with a proof of its interaction with the source URL at a given time. Before sending the response it also stores the data on IPFS, and its hash in the database. If the request is made to a decentralized oracle and the number of designated reporters specified in the request are greater than zero, then the request enters a designated reporting phase. The *URL* field specifies the data source to be used for reporting purpose.

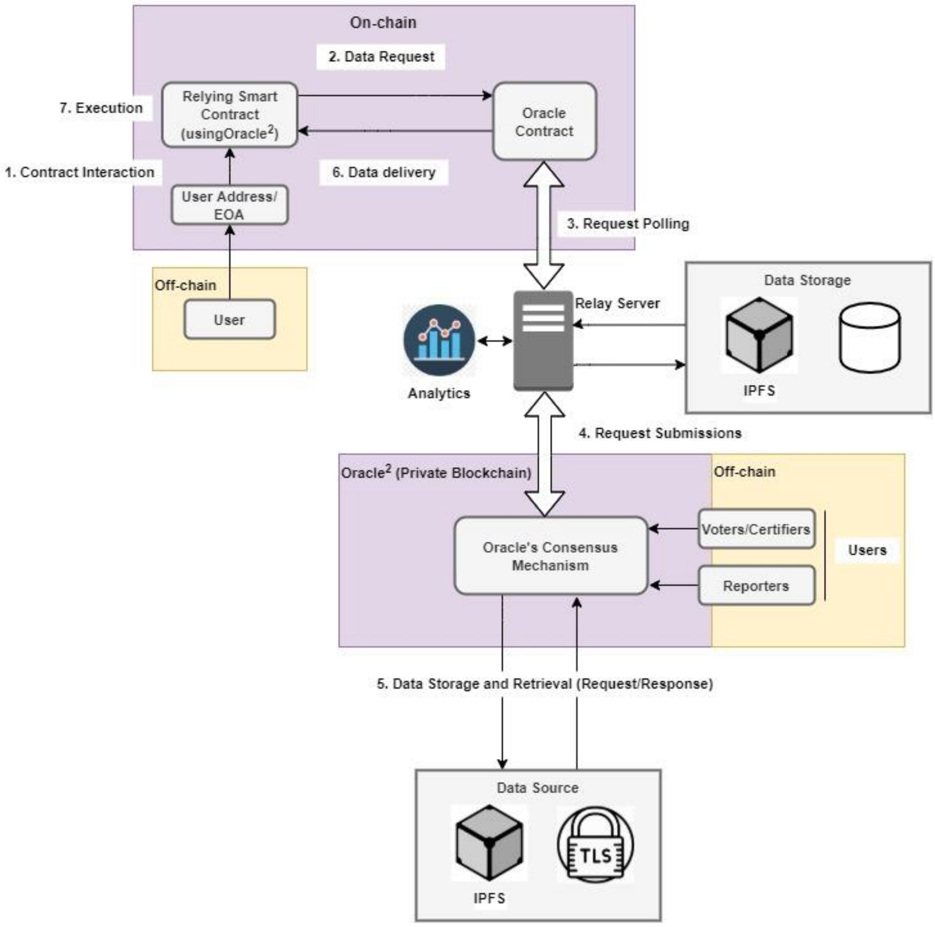


Figure 21: Working of the Oracle

Designated Reporting

This is a 3-day interval, where the request waits for its designated reporter (or reporters). If the number of designated reporters is greater than one and if k out of n reporters submit identical outcomes, then the outcome is decided, and the result is sent to the relying contract on the blockchain. If the number of reporters is only one, then the request transitions into the Voting/Dispute phase. Also, if the request is undecided i.e., if the reporting process has not helped reach consensus, then the request transitions into the Voting phase. The designated reporters are required to stake a REP token deposit, which they forfeit if the data they reported did not coincide with the final outcome.

Voting/Dispute

If none of the reporters report within the 3-day interval or the number of designated reporters is one, and has reported, then the request moves into the voting phase. The voting/dispute phase is a 7-day period allowed for voting/certification or disputing. For requests containing a single designated reporter, if this phase passes by without a successful dispute, then the data provided by the designated reporter becomes the final outcome. For scenarios where the outcome remains undecided even after reporting by more than one reporter, the request moves into a voting/certification phase where the participants of the oracle system vote and certify the request. The request reaches a consensus if the outcome that has the highest certification stake follows the outcome that has the majority of voting stake. C_{min} and V_{min} are variables that refer to the minimum number of certifiers and voters required by each request respectively.

Rewards

The rewards distributed to voters and certifiers are stake weighted sums of the bounty provided by the requesting contract. All the reporters, voters and certifiers who have reported false data are penalized. The penalties collected from participants who have reported incorrectly are distributed as rewards to those who have reported data correctly, i.e., those who have reported the same data as the final outcome.

Consider an example where the bounty submitted with a request is 2 ETH and, the participants are –

- Designated reporter D1 who stakes a deposit of 2 REP
- Voters – V1, V2 and V3 who stake deposits 4 REP, 3 REP and 2 REP respectively,
- Certifiers – C1 and C2 who deposits 10 REP and 15 REP respectively

The total REP staked as a deposit is 36 REP. Suppose voter V1 and certifier C1 report false outcomes, then they are penalized, and their deposit is transferred to the reward pool. At this point the reward pool holds a value of 2 ETH and 14 REP. Once the final outcome is decided, the deposits made by participants who reported correct outcomes are returned together with a proportionate reward. In our case voter V2 would receive a reward (3/22) * (2 ETH + 14 REP).

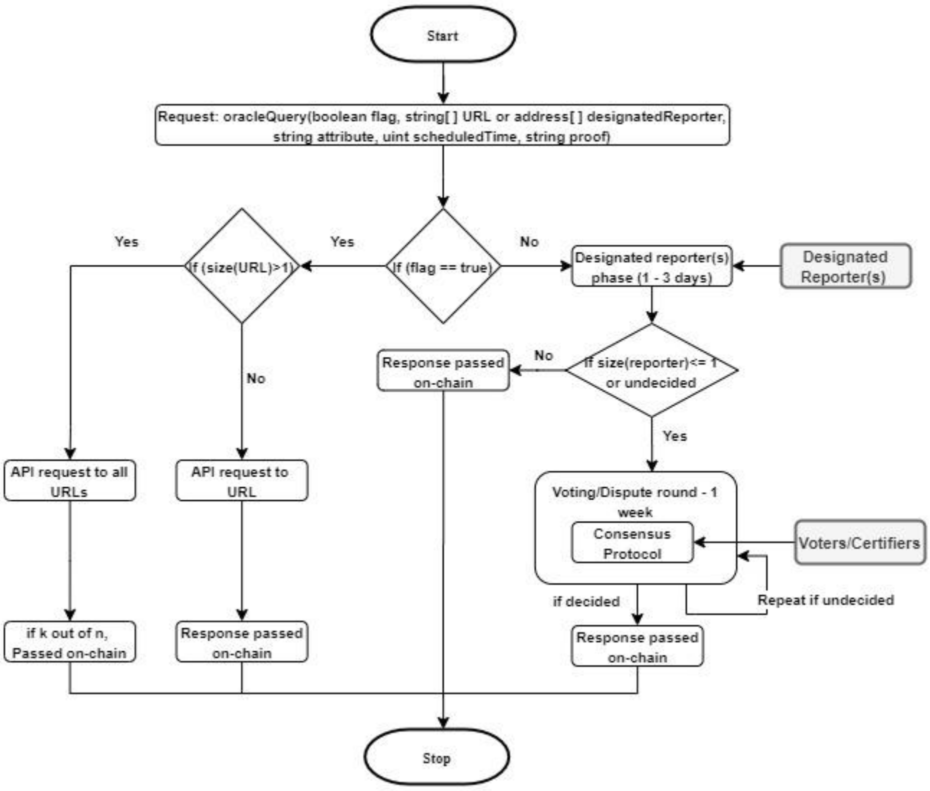

Start
Request: oracleQuery(boolean flag, string[ ] URL or address[ ] designatedReporter, string attribute, uint scheduledTime, string proof)
Yes
If (size(URL)>1)
Yes
If (flag == true)
No
Designated reporter(s) phase (1 - 3 days)
Designated Reporter(s)
No
Response passed on-chain
No
If size(reporter)<= 1 or undecided
Yes
API request to all URLs
API request to URL
Voting/Dispute round - 1 week
Consensus Protocol
Voters/Certifiers
if decided
Repeat if undecided
if k out of n, Passed on-chain
Response passed on-chain
Response passed on-chain
Stop

Figure 22: Detailed Workflow of the Oracle Request

Chapter 5

Implementation

Our implementation consists of an on-chain component which essentially constitutes the Oracle Contract, which is written in *Solidity*. The contract contains a data structure to store the requests made by relying contracts and code to manage the consensus mechanism of the oracle. In our case we use the ‘k out of n’ scheme to reach consensus i.e., if k out of a total of n oracles report identical data, then we pass the final outcome to the relying contract. The off-chain component is implemented using the node.js and the web3 interface. The web3 interface is used to access the events triggered by the contract and create signed transactions, and node.js is used to code the logic for the centralized off-chain oracle that uses a database server to store the details of the oracle and data reported by them. Let us now discuss the implementation in a more detailed manner.

5.1 On-chain Implementation

The on-chain implementation stores the requests made by the relying contracts that choose to use the oracle and implements the logic to arrive at final outcome, that is reported to the requesting/relying contract. The data structure used by the oracle contract is –

```
struct Request {
    bool api;                    //centralised or decentralised service
    uint id;                     //request id
    string source;               //source url
    string attribute;            //json attribute (key) to retrieve in the response
    address contractAddress;          // address of the contract to return the value
    string finalOutcome;              //value from key
    mapping(uint => string) outcomes;   //data provided by the oracles
    mapping(address => uint) qtum;  //0=oracle hasn't voted, 1=oracle has voted
   }
```

The boolean value ‘api’, if set to false, uses a decentralized service and if set to true uses a centralized oracle. The integer ‘*id*’ is used to uniquely identify a request, ‘*source*’ is a query or a URL to a data source, ‘*attribute*’ denotes a key in the response that is used to access one of the many values present in the response, ‘*contractAddress*’ stores the address of the requesting contract, ‘*finalOutcome*’ is the value that is reported to the requesting contract, ‘*outcomes*’ is a list of values that store data reported by oracles as key-value pairs and ‘*qtum*’, which is also a list of values that stores details of whether an oracle has voted i.e., if an oracle has not voted, the values is set ‘0’ and if an oracle has voted the value is set to ‘1’.

The contract contains three functions – createRequest(), which creates a request on behalf of the relying contract and adds it to the list of requests, the updateCentralizedRequest() method, which is used by the centralized service to report data to the blockchain, for each request and the updateDecentralizedRequest() method, which is used by decentralized oracles to submit data on-chain. The createRequest() function accepts parameters ‘*apiFlag*’ – used to check whether the request is for a centralized or a decentralized service, ‘*source*’ – which is a query or a URL to a data source and ‘*attribute*’ – which is the key in the response (E.g., a JSON). The updateCentralizedRequest() and updateDecentralizedRequest() functions accept an ‘*id*’ – used to uniquely identify a request and an ‘*outcome*’ – data reported by the oracle. Given below is the oracle contract.

Oracle Contract

```
pragma solidity ^0.4.25;

contract Oracle {
  Request[] requests; //list of requests made to the contract
  uint currentId = 0; //increasing request id
  uint minQtum = 2; //minimum number of responses to receive before declaring final result
  uint totalOracleCount = 3; // Hardcoded oracle count
  event NewRequest (bool apiFlag, uint id, string source, string attribute);
  event UpdatedRequest(uint id, string source, string attribute, string agreedValue);

  // defines a general api request
    struct Request {
      bool api;
      uint id;                    //request id
      string source;              //API url
      string attribute;         //json attribute (key) to retrieve in the response
      address contractAddress;        // address of the contract to return the value
      string agreedValue;           //value from key
      mapping(uint => string) outcomes;   //data provided by the oracles
      mapping(address => uint) qtum;  //0=oracle hasn't voted, 1=oracle has voted
    }

    function callback(uint id, string result) payable public{

    }
    function createRequest (bool _apiFlag, string memory _source, string memory _attribute)
public payable{

      uint length = requests.push(Request(_apiFlag, currentId, _source, _attribute,
msg.sender,""));
      Request storage r = requests[length-1];

      // Hardcoded oracles address
      r.qtum[address(0x06517087a60E27621C67FE9221CF0327c0d057f0)] = 0;
      r.qtum[address(0x67f4A64D18B549f5C218fb7A0Ce358af34304510)] = 0;
      r.qtum[address(0x46c18280FBc55ba665177416f316ae151f1baEba)] = 0;
      // launch an event to be detected by oracle outside of blockchain
      emit NewRequest (_apiFlag, currentId, _source, _attribute);
      // increase request id
      currentId++;
    }

    function updateDecentralizedRequest (uint _id, string memory _valueRetrieved) public {
      Request storage currRequest = requests[_id];
      //check if oracle is in the list of trusted oracles
      //and if the oracle hasn't voted yet
      if(currRequest.qtum[address(msg.sender)] == 0){
       //marking that this address has voted
       currRequest.qtum[msg.sender] = 1;
       //iterate through "array" of outcomes until a position if free and save the retrieved value
       uint count = 0;
       bool found = false;
       while(!found) {
        //find first empty slot
        if(bytes(currRequest.outcomes[count]).length == 0){
         found = true;
         currRequest.outcomes[count] = _valueRetrieved;
        }
        count++;
       }
       uint currentQtum = 0;
       //iterate through oracle list and check if enough oracles(minimum qtum)
       //have voted the same answer as the current one
       for(uint i = 0; i < totalOracleCount; i++){
        bytes memory a = bytes(currRequest.outcomes[i]);
        bytes memory b = bytes(_valueRetrieved);

        if(keccak256(a) == keccak256(b)){
         currentQtum++;
         if(currentQtum >= minQtum){
          currRequest.agreedValue = _valueRetrieved;
          emit UpdatedRequest (
           currRequest.id,
           currRequest.source,
           currRequest.attribute,
           currRequest.agreedValue
          );
          //require(currRequest.contractAddress.call(bytes4(keccak256("__callback(uint,
string)")), currRequest.id, currRequest.agreedValue));
          Oracle relyingContract = Oracle(currRequest.contractAddress);
          relyingContract.callback(currRequest.id, currRequest.agreedValue);
         } } }
    }}

    function updateCentralizedRequest (uint _id, string memory _valueRetrieved) public {
      Request storage currRequest = requests[_id];
      //check if oracle is in the list of trusted oracles
      //and if the oracle hasn't voted yet
      if(currRequest.qtum[address(msg.sender)] == 0){
         //marking that this address has voted
         currRequest.qtum[address(0x06517087a60E27621C67FE9221CF0327c0d057f0)]    =
1;
         currRequest.qtum[address(0x67f4A64D18B549f5C218fb7A0Ce358af34304510)] = 1;
         currRequest.qtum[address(0x46c18280FBc55ba665177416f316ae151f1baEba)] = 1;
         currRequest.agreedValue = _valueRetrieved;
         emit UpdatedRequest (
           currRequest.id,
           currRequest.source,
           currRequest.attribute,
           currRequest.agreedValue
          );
         Oracle relyingContract = Oracle(currRequest.contractAddress);
         relyingContract.callback(currRequest.id, currRequest.agreedValue);
}}}
```

5.2 Off-chain Implementation

The off-chain implementation consists of a set of components used to retrieve the events generated by the oracle contract, which contains a source URL and another set of components used to send a response to the relying contract in the form of a signed transaction. The set of components that send data or response to the blockchain are divided into two – centralized and decentralized. The centralized component consists of an infinite loop that poll the events generated by the oracle contract from the blockchain, stores newly created requests in the database, generates an API request and relays the response on-chain. The decentralized component consists of independent oracles who submit data for each of the requests from their respective accounts. *Web3.js* interface is used to access the events emitted by the oracle contract and create signed transactions on the blockchain by using externally owned accounts i.e., using an address and a key. *Node.js* is used by the centralized oracle in order to create an infinite loop, make API requests and access the database to insert and update information.

Centralized Oracle

The centralized oracle polls the oracle contract for newly created events(only those that have the *flag* value set as true) every 5 seconds and inserts or updates the events in the database. It connects to the database by creating a client. The code to connect to the database is –

```
const {Client} = require('pg')
let client = new Client({
  user: 'admin',
  host: 'localhost',
  database: 'oraclesquare',
  password: 'admin',
  port: 5432,
});
client.connect();
```

In the above code, we connect to the database at port number *5432* using user ‘*admin’* and password ‘*admin*’. The database name is ‘*oraclesquare*’ and the host URL is the current computer. After the connection has been established, API requests are made for events where data has not yet been sent on-chain by the oracle. The responses to those requests are stored in the database and sent to the oracle contract in the form of a signed transaction which calls the updateQuery() function with parameters ‘*id*’ – used to uniquely identify a request and an ‘*outcome*’ – which is the API response. The code used to create an API request is –

```
const https = require('https');
https.get(request_list[i][1], (res) => {
        responseStatus = res.statusCode;
        console.log('statusCode:', res.statusCode);
        res.on('data', function(body){
                    apiData='';
                    if(responseStatus==200)
                      apiData += body;
                 });
        // The whole response has been received. Print out the result.
        res.on('end', () => {
                    console.log('explanation: ');
                    //var response = JSON.stringify(apiData);
                    if(responseStatus==200){
                      console.log((JSON.parse(apiData))[apiDataField]);
                      oracleData = (JSON.parse(apiData))[apiDataField];
                    }
                });
        }).on("error", (err) => {
                 console.log("Error: " + err.message);
                 });
```

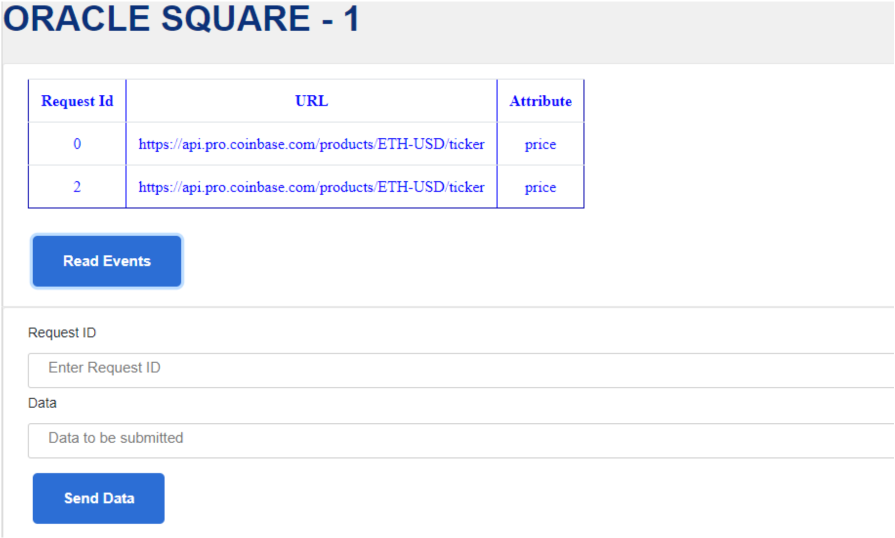
The *https.get()* statement is used to make an API request to the URL passed as the first parameter of the get() function. Once a valid response to an API request is received the, data received is stored in the database before sending them to the blockchain. The data is sent to the blockchain using a signed transaction as follows –

```
var txCount = web3.eth.getTransactionCount(sender);
web3.eth.getTransactionCount(sender, (err, txCount) => {
        const txObject1 = {
                nonce:    web3.utils.toHex(txCount),
                gasLimit: web3.utils.toHex(800000),
                gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
                to: oracleContracAddress,
                data: oracleContract.methods.updateRequest(id, oracleData).encodeABI()
                }
        const tx = new Tx(txObject1);
        tx.sign(senderPrivateKey);
        const serializedTx = tx.serialize()
        const raw = '0x' + serializedTx.toString('hex')
        web3.eth.sendSignedTransaction(raw, (err, txHash) => {
                console.log('err:', err, 'txHash:', txHash)
                })
        })
```

The transaction object '*txObject1*' contains the transaction details such as the *gasLimit* and *gasPrice* which is used to specify the expendable gas to be used for the transaction, *to* – the recipient of the transaction, and *data* which constitutes the ABI i.e., the JSON structure of the method *updateRequest()* together with parameters *id* and *oracleData.* The transaction object is then signed with the private key using the *tx.sign()* method. The signed transaction is then serialized into a hexadecimal format and sent to the blockchain using the *web3.eth.sendSignedTransaction()* method. Note that contract method calls that change the state of the contract require ETH to be passed along with the transaction, to purchase the gas required for transaction execution.

Decentralized Oracle

The decentralized oracle polls the oracle contract for newly created events(only those that have the *flag* value set as false) and displays them on the user interface of each oracle provider. The oracle provider then sends the data for each of the requests present in the list as a signed transaction to the blockchain. The code to send a signed transaction is similar to the one used in the centralized oracle. Given below is the user interface for the Oracle.



ORACLE SQUARE - 1

Request Id	URL	Attribute
0	https://api.pro.coinbase.com/products/ETH-USD/ticker	price
2	https://api.pro.coinbase.com/products/ETH-USD/ticker	price

Request ID

Enter Request ID

Data

Data to be submitted

Figure 23: User Interface for the Decentralized Oracle

Chapter 6

Evaluation

In this chapter we consider a working example of the design, compare the novel design with the state of the art and discuss the novel aspects in the design.

6.1 Working Example

Consider a request made by a relying contract –

```
pragma solidity ^0.4.21;
import
"https://github.com/georgejohnchavady/georacle/blob/master/QtumOracleContractv
1.1.sol";
contract ExampleContract is Oracle{
    function updatePrice() payable public{
        //send 12 wei while creating contract instance
        address[] public designatedReporter = [0x36eaf79c12e96a3dc6f53426c,
0xf235aa56dd96bda02acfb361e];
        address oracle_addr  = 0xc359a1E80459589CB3BF18818E3daF19ABD65950;
        Oracle oracle = Oracle(oracle_addr);
        string memory url = "https://api.pro.coinbase.com/products/ETH-USD/ticker";
        string memory attribute = "price";
        oracle.createRequest(false, designatedReporter, url, attribute, 1000 * 60 * 60 *
24 *3);
    }

    function callback(uint id, string result) payable public{
        emit LogNewOracleResponse(id, result);
    }
}
```

The relying contract The request ‘*oracle.createRequest(false, designatedReporter, url, attribute, 1000 * 60 * 60 * 24 *3)*’ has a false value passed as a parameter which indicates that the request requires a decentralized service. The request passes 2 addresses as designated reporters, ‘*https://api.pro.coinbase.com/products/ETH-USD/ticker*’ as the data source, ‘*price*’ as the attribute and schedule time as 3 days. This statement basically creates an event containing the data source, attribute, designated reporters, and schedule time. Once the interval specified in the scheduled time has passed, the request enters the designated reporting phase. During this phase, if k out of n reporters submit identical data – in our case the value of n equals 2. The first reporter ‘*0x06517087a60E27621C67FE9221CF0327c0d057f0*’ reports the value ‘*233*’ and the second reporter ‘*0x67f4A64D18B549f5C218fb7A0Ce358af34304510*’ reports the value ‘*233*’. Our consensus mechanism checks whether k out of n reporters have submitted identical values. Assume that the value of k is set to 2. In our case both the reporters have reported the value ‘*233*’ and satisfies the condition that 2 out of 2 reporters submit identical data. And, thus the value ‘*233*’ is passed to the relying contract by calling the callback() function. If both reporters hadn’t submitted identical data, then the request would remain undecided and move to the voting/certification phase where voters and certifiers stake a deposit supporting a particular outcome. The outcome that has the majority stake is decided as the final outcome and the same is reported on-chain.

Another scenario would be a state where the first reporter '0x06517087a60E27621C67FE9221CF0327c0d057f0' reports the value '233', the second reporter '0x67f4A64D18B549f5C218fb7A0Ce358af34304510' reports the value '234' and the third reporter '0x46c18280FBc55ba665177416f316ae151f1baEba' reports a value of '235'. Since no outcome has majority votes, the request moves into the voting and certification phase where the outcome is decided, if the outcome that has the maximum certification stake follows the outcome that has the maximum voting stake. In our case, let us suppose the voting stakes on outcomes '233', '234' and '235' are 3 REP, 6 REP and 2 REP respectively and the certification stakes are 10 REP, 15 REP and 0 REP respectively. The outcome '234' has majority stakes for both voting and certification i.e., 6 REP and 15 REP respectively and therefore becomes the final outcome which is passed to the blockchain.

6.2 Comparison

The state-of-the-art oracles constitute of either decentralized or centralized oracles, but none provide a choice – to choose a centralized or a decentralized service. For example, the centralized oracle Provable provides makes a request to its centralized server to access data from the outside world. It does not provide an option of using a decentralized reporting. Decentralized oracles such as Augur have a designated reporting phase (a 3-day period) where a single reporter submits data for a request and later moves into the dispute phase of 7 days. This process often takes a long time to even service simple requests. Our oracle divides the decentralized reporting process into two – micro-decentralization and macro-decentralization. Micro-decentralization is one where k out of n designated reporters submit identical values for a request and the final outcome is decided. Situations where there are fewer number of reporters or when the outcome is undecided (i.e., k out n reporters do not submit identical data) the request moves into a dispute or voting/certification round where participants stake value on a particular outcome and play a risk/reward game, until the values is decided.

6.3 Innovation

This section discusses the various aspects of innovation in the design of the oracle. The oracle offers features such as flexibility and decreased turn-around time.

Flexibility

It is important that smart contracts are able to choose between a centralized and a decentralized oracle service. A centralized service has lesser turn around time when compared to a decentralized oracle. When oracle contracts require data that is easily available from a data source such as an API, it would be unwise to use a decentralized service. On the contrary, if an oracle contract of very high value relies on external data source such as an API, the service becomes a single point of failure. It would be wise for such contracts to use a decentralized oracle for its execution. Our oracle makes it possible for oracle contracts to choose between a centralized and a decentralized service.

Turn-around Time

Another important aspect in oracles is their turn-around time. Together with flexibility comes an opportunity to improve the turn-around time of responses to requests. Also, the techniques of micro-decentralization and macro-decentralization helps reduce the response time by making it possible for multiple reporters to submit data during the designated reporter phase, to reach consensus. The system thus helps improve the process of reporting data by being flexible and exhibits better performance.

Private Blockchain

The design uses a private blockchain to implement the decentralized oracle platform. This is done to reduce costs of computation and maintain transparency in the reporting process. Further, analytics can be used to rate the performance of oracles using the historical data stored on the blockchain.

Qtum Tokens

The Qtum blockchain provides a platform for easy creation of tokens, which can be used by participants of the decentralized oracle to stake a deposit during the voting/dispute round.

Chapter 7

Conclusion

7.1 Summary

Through this work we have been able to study the blockchain, and the various oracles that currently exist. It is important that a highly secure platform such as the blockchain does not fall prey to an unsecure oracle system. The aspects of security, latency and flexibility have been considered in the newly proposed design. The security aspect deals with data availability and data integrity. Obstruction of data from being provided to a contract on-chain, at the right time, results in the contract not being executed on time. Also, it is important that correct data be passed on-chain, lest adversaries or wrong parties benefit from the execution of the contract. Using a decentralized platform helps reduce the risk of incorrect data being reported. But the reporting time required by centralized oracles is much lesser when compared to decentralized oracles. A well-informed choice is imperative to optimal performance. Our oracle design provides an option to choose between a centralized and a decentralized platform which is best suited for real world use cases.

7.2 Future Works

The scope for improvement lies in the decentralized oracle system, where historical data of oracles can be used to better secure the system. The historical data includes details of reporting i.e., reporting correct or incorrect data, REP token age. It may also be possible to detect threats using machine learning models on historical data.

REP token aging

The REP tokens held by participants for reporting can be used to rank or weight them. An oracle that has held REP for a longer period is more trustworthy than an oracle who has recently joined the platform and has only held their REP for a short period.

Analytics dashboard

Implementation of a dashboard to view the performance of oracles with respect to reporting can help contracts choose designated reporters wisely. For example, ranking various oracles based on their historical data of reporting or providing the list of oracles who have reported incorrect data in the past. A similar option is provided by Chainlink, which uses a reputation contract to store the feedback received from relying contracts on the performance of oracles.

Threat Detection

Once sufficient amount of historical data is available, machine learning models can be implemented to detect threats in the system. E.g., reporting anomalies. Oracles may also be classified into trustworthy and untrustworthy. When threats are detected, the reports provided by oracles that are more trustworthy (from experience i.e., historical data) can be given a higher weightage than those deemed to be unreliable.

Bibliography

[1] Kolb, J., Abdelbaky, M., Katz, R. H., & Culler, D. E. (2020). Core Concepts, Challenges, and Future Directions in Blockchain: A Centralized Tutorial. ACM Computing Surveys, 53(1), 1–39. https://doi-org.elib.tcd.ie/10.1145/3366370

[2] Satoshi Nakamoto. (2009). Bitcoin: A Peer–to–Peer Electronic Cash System. Retrieved from: https://bitcoin.org/bitcoin.pdf .

[3] Vitalik Buterin. (2014). A Next-Generation Smart Contract and Decentralized Application Platform. Retrieved from: https://github.com/ethereum/wiki/wiki/White-Paper .

[4] Feng, T., Yu, X., Chai, Y. and Liu, Y. (2019). Smart contract model for complex reality transaction. International Journal of Crowd Science, Vol. 3, pp. 184-197.

[5] M. Merlini, N. Veira, R. Berryhill and A. Veneris. (2019). On Public Decentralized Ledger Oracles via a Paired-Question Protocol. IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Seoul, Korea (South), pp. 337-344.

[6] Provable [Online]. Available: https://docs.provable.xyz/

[7] Karame, G.O., & Audroulaki, E. (2016). Bitcoin and Blockchain Security.

[8] Zhang, Fan & Cecchetti, Ethan & Croman, Kyle & Juels, Ari & Shi, Elaine. (2016). Town Crier: An Authenticated Data Feed for Smart Contracts. Retrieved from: https://eprint.iacr.org/2016/168.pdf

[9] V. Costan, S. Devadas. (2016). Intel sgx explained. IACR Cryptology ePrint Archive. Vol. 86.

[10] Berberich, M.; Steiner, M. (2016). Blockchain technology and the gdpr how to reconcile privacy and distributed ledgers. European Data Protection Law Review (EDPL), 2(3), 422-426.

[11] Blockchain Oracles Explained. Available: https://www.binance.vision/blockchain/blockchain-oracles-explained

[12] Bitcoin's UTXO Set Explained. Available: https://www.mycryptopedia.com/bitcoin-utxo-unspent-transaction-output-set-explained/

[13] Full Node. Available: https://en.bitcoin.it/wiki/Full_node

[14] Unspent transaction output. Available: https://en.wikipedia.org/wiki/Unspent_transaction_output

[15] Domain Name System. Available: https://en.wikipedia.org/wiki/Domain_Name_System

[16] Remote Procedure Call. Available: https://en.wikipedia.org/wiki/Remote_procedure_call

[17] Dr. Gavin Wood. (2019). Ethereum: A secure decentralised generalised transaction ledger byzantium version. Retrieved from: https://ethereum.github.io/yellowpaper/paper.pdf

[18] Vitalik Buterin. (2013). Ethereum Whitepaper. Retrieved from: https://ethereum.org/whitepaper/

[19] Antonopoulos, A. M., & Wood, G. (2018). Mastering Ethereum: Building smart contracts and DApps.

[20] Patrick Dai, Neil Mahi, Jordan Earls, Alex Norta. Smart-Contract Value-Transfer Protocols on a Distributed Mobile Application Platform. Retrieved from: https://old.qtum.org/en/white-papers

[21] QTUM Blockchain New Whitepaper. (2020). Retrieved from: https://qtum.org/en/developer

[22] Proof of stake. Retrieved from: https://en.wikipedia.org/wiki/Proof_of_stake

[23] Sin Kuang Lo, Xiwei Xu, Mark Staples, Lina Yao. (2020). Reliability analysis for blockchain oracles. Volume 83.

[24] Abdeljalil Beniiche. (2020). A Study of Blockchain Oracles. Retrieved from: https://deepai.org/publication/a-study-of-blockchain-oracles

[25] Nash Equilibrium. Retrieved from: https://en.wikipedia.org/wiki/Nash_equilibrium

[26] John Adler, Ryan Berryhill, Andreas Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. (2018). ASTRAEA: A Decentralized Blockchain Oracle. Retrieved from: https://www.eecg.utoronto.ca/~veneris/18chain1.pdf

[27] TLSnotary - a mechanism for independently audited https sessions. (2014). Retrieved from: https://tlsnotary.org/TLSNotary.pdf.

[28] Steve Ellis, Ari Juelsy, Sergey Nazarov. (2017). ChainLink - A Decentralized Oracle Network. Retrieved from: https://link.smartcontract.com/whitepaper

[29] Jack Peterson, Joseph Krug, Micah Zoltu, Austin K. Williams, Stephanie Alexander. (2018). Augur: a Decentralized Oracle and Prediction Market Platform (v2.0). Retrieved from: https://www.augur.net/whitepaper.pdf

[30] Sin Kuang Lo, Xiwei Xu, Mark Staples, Lina Yao. (2020). Reliability analysis for blockchain oracles. Volume 83.

[31] Writing oracle services. Retrieved from: https://docs.corda.net/docs/corda-os/4.5/oracles.html

[32] Richard Gendal Brown. (2018). The Corda Platform: An Introduction. Retrieved from: https://www.corda.net/content/corda-platform-whitepaper.pdf

[33] Gnosis. Retrieved from: https://gnosis.io/pdf/gnosis-whitepaper.pdf

[34] Microsoft. (2016). Introducing Project “Bletchley”. Retrieved from: https://github.com/Azure/azure-blockchain-projects/blob/master/bletchley/bletchley-whitepaper.md