Universal Blockchain Oracle

Igor Ďurica
Slovak University of Technology in Bratislava
Faculty of informatics and information technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xduricai@stuba.sk

***Abstract*—Applications that live on the blockchain as such do not have problems accessing data stored on-chain, even in cases where the data is stored on a different chain than the app itself. However. when a decentralized application needs data that is stored off-chain, issues arise. Blockchain oracles are trusted third party sources of information and a gateway to the centralized web, that exist as the solution to this problem. This paper delves into the ecosystem of blockchain oracles and explores some of its shortcomings. Primarily in regard to the variety of data they provide, or rather, the lack thereof. Most oracles currently only operate as price feeds for various digital assets. Our project deviates from this use case and instead focuses on giving users the ability to receive data from any off-chain API. We implement a universal oracle on the Solana blockchain, complete with a web application for users as well as a working oracle node implementation.**

***Index Terms*—blockchain, solana, de-fi, decentralized finance, blockchain oracle**

I. Introduction

As the De-Fi ecosystem grows and expands, so does the need for external sources of data. One of the major downsides of decentralized applications is that they can generally only access data stored on their respective blockchains. This poses a problem because many services require outside sources of data to function, e.g. digital currency exchanges which rely on external APIs for exchange rates. This is where blockchain oracles come into play. An oracle acts as a trusted third party which collects, verifies, and aggregates data before providing it to the smart contract. A major part of this process happens off-chain, however most oracles tend to be linked to an on-chain oracle smart contract. This contract serves as an interface for other contracts on the chain to interact with and use to query data from off-chain sources. It is also important to understand that oracles themselves are never the source of the information they relay to smart contracts, rather just a method for its transmission into the blockchain. This transmission doesn't necessarily have to be inbound (data being supplied to the blockchain), it can also be outbound (data being supplied to the oracle) [8].

II. Types of Oracles

There are multiple factors by which we can categorize different oracle projects. The three that we will primarily focus on are the source of the oracle's information, the design pattern of the oracle and finally the model of trust they use.

Bachelor study programme in field: Informatics
Supervisor: Ing. Kristián Košťál, PhD., Institute of Computer Engineering and Applied Informatics, Faculty of Informatics and Information Technologies STU in Bratislava

A. Oracles by data source

Software oracles make up the majority of blockchain oracles and often operate on simple request/response or data feed models. They query information from websites, servers, or databases available on the web. An example use of a software oracle is the aforementioned checking of currency exchange rates via a web API.

Hardware oracles use sensors and trackers to gather information based on physical changes or interactions in the real world. Hardware oracles could be utilized in supply chain tracking by informing a smart contract when an item is picked up or reaches its destination. Any decentralized application designed around Internet of Things will likely rely on hardware oracles as well. [1].

B. Oracles by design pattern

Request-response oracles are used when the set of data is too large to feasibly be stored on-chain and users also do not ever need to retrieve more than small subsets of said data at a time. This is done as an alternative to storing the contents of the database in the contract and accessing it directly, which greatly improves scalability by allowing significantly more data to be stored.

Publish-subscribe oracles provide an on-chain contract with a stable feed of data that is subject to frequent change. Every time the oracles records a change in this data, it publishes the updated version to its subscribers. This type of oracles is commonly used by applications which need accurate and up to date information.

Immediate-read oracles provide data that is required to execute an operation, such as an authenticity check on a legal document or an artwork [3].

C. Oracles by trust model

Centralized oracles obtain data from a single source and transmit it directly into the on-chain contract. These solutions tend to be faster and more efficient as they only require a single node but are also more susceptible. A potential attacker only needs to compromise a single oracle to provide faulty data to the smart contract [3].

Decentralized oracles collect data from multiple sources and typically rely on some type of a consensus mechanism

to determine what data should be sent to an on-chain oracle contract. Examples include a K-out-of-N system, where at least K oracles out of the total N oracles need to agree on the data for a consensus to be reached or systems where individual nodes can stake tokens to challenge the agreed upon result. While decentralized networks of oracles tend to be more secure, they are also slower due to the time it takes for a consensus to be reached. They are also more hardware intensive given the fact that they need to operate multiple nodes [5]. Decentralized oracles can further be split into two defining categories based on what trust mechanism they utilize to ensure the security of the network: voting-based and reputation based [9].

Voting-based oracle systems are designed to combat discrepancies in reported data or potential malicious misreporting by letting users certify data or vote on its validity. To participate in this process, users have to put up a stake. If the agreed upon outcome matches the one the user proposed, they are rewarded. If the opposite is true, they are penalized, typically by losing a portion of their stake. The more oracles participate in such system, the more secure it becomes [9].

In **reputation-based** systems, each oracle is assigned a reputation score which is a reflection of its performance. When oracles truthfully report data, they are rewarded and gain reputation. When they report incorrect data or do not report when expected to, they lose reputation. The rate at which reputation is gained is typically much lower than the rate at which it is and losing enough reputation leads to complete removal from the network. The primary deterrent is the opportunity cost of not being able to earn rewards via participating in the network because oracles with a higher score are more likely to be selected than those with a lower one [11].

III. Related Work

There are many commercial oracle providers on the market as well as decentralized applications operating oracles tailored to their own needs. That being said, there is currently one provider currently holds most of the market share and dwarfs all competition.

ChainLink is an oracle provider with full support for decentralized oracle networks on both the data source and oracle level, meaning it aggregates data from multiple oracles querying multiple data sources. All aggregation of data is performed by the off-chain, components of the system, meaning that only a single transaction is required to publish this data onto the blockchain. Delegating most computations to the off-chain network of oracles results in a significant decrease in overall gas fees. ChainLink utilizes the Off-Chain Reporting protocol or OCR for generation of reports based on data from third party sources. OCR is composed of 3 sub-protocols [4].

The **pacemaker protocol** serves to divide the process into time intervals referred to as epochs. Each epoch has a leader whose task is to generate a report based on data from the other oracles. All oracles have an internal timer which is reset every time the leader emits a progress event on the report. If enough oracles declare that the timer has run out, a new epoch is started[4], [2], [6].

The report **generation protocol** divides epochs into rounds of gathering observations from oracles. The frequency of these rounds is ultimately decided by the leader of a given epoch. Once a set number of observations with valid signatures is reached, the leader may generate and sign a report and submit it to the transmission protocol [2], [6].

Once a report is handed off to the **transmission protocol**, it is distributed to each oracle in the network. Oracles then verify the data and sign it if they approve of the reported values. To prevent unnecessary reporting, data can only be submitted to the smart contract if the median value of observations in the report has changed by a large enough margin since the last reported value or enough time has passed. This is done to prevent identical reports from successive rounds from being reported. If this condition is met and enough nodes have signed the report, a pseudo-random function is used to select a set of nodes. These oracles will attempt to transmit the report to the on-chain smart contract. Each transmitting oracle is assigned a delay to ensure a scheduled sending process. If an oracle manages to publish the report to the blockchain, the process ends, otherwise the next oracle in the schedule is selected [2], [6].

As far as security is concerned, ChainLink utilizes standard public key encryption and digital signatures for authenticity. Oracles themselves operate on a reputation-based model of trust, where oracles are rewarded and or penalized based on their reputation. Reputation is calculated based on multiple factors such as the total number of accepted assignments, completed assignments, the average response time as well as how many times the oracle has been penalized for misreporting in the past. If an oracle behaves maliciously, its reputation will drop resulting in an eventual removal from the network as well as the potential removal of other oracles owned by the operator of said oracle. Instances of malicious behaviour also undermine the trustworthiness of the entire network. This can potentially decrease the value of the native LINK token which is the primary method of compensation for oracle operators [3].

IV. Filling in the Gaps

Chainlink is the most successful oracle provider by a large margin and for a good reason. Its oracles are stable, secure, reliable and available on a wide variety of different chains. That however does not mean that it satisfies all the needs of the market. Much like most major blockchain providers, the primary purpose of Chainlink is to serve as a price data feed for de-fi applications which require accurate up-to-date prices of digital assets to operate their business. Chainlink does this very well, however their oracle services outside of that are relatively limited. While the price feed use case certainly makes the most sense in the world of De-Fi, it is not the only meaningful one. Hence, we have decided to explore other means through which blockchain oracles could provide outside information to the network. One of the use cases that appears

to be paritcularly underrepresented is that of giving users the ability to query any API of their choice. It is worth noting that Chainlink does provide users the option to do exactly that via Chainlink oracles. However, it comes in the form of a very basic feature which requires users to pay a one-time fee for a single API call. This feature is also only available for the Ethereum blockchain as of the writing of this paper. Most of the other major oracle providers don't seem to offer similar services at all. We believe this is a part of the oracle ecosystem that has a lot of room for expansion and improvement.

Chainlink of course isn't the only available solution to the oracle problem and there are a plenty of other competitors in this market. The write-up of this paper was preceded by an exhaustive review of available oracle projects, namely those covered in [9]. This further reinforced our belief that a generic oracle option for dApp developers is needed. The vast majority of solutions were built with a narrow specific use case in mind, typically that of the aforementioned price feeds. The remaining solutions also have various disadvantages. Some solutions technically do provide the functionality we are looking for but they come in the form of full-on layer-2 blockchains rather than simple on-chain solutions. Other solutions suffer from poor performance as a result of utilizing complex algorithms and or consensus mechanisms. The two solutions that stood out were Provable and Mobius. Provable is meant to serve as a platform-agnostic bridge between the blockchain and the internet, allowing decentralized applications to query external APIs [9, 10]. Provable unfortunately has runs into the problem of being fully centralized, therefore providing a single point of failure to any application that decides to use it. It must also be integrated with the contract that it will be utilized in, meaning that changes have to be made to the codebase before Provable can be used. This also means that further changes have to be made down the line in the event that the service becomes unavailable. Mobius on the other hand is a decentralized oracle relying on a proof of stake mechanism for consensus, which lets users connect to off-chain APIs. At first glance Mobius seems like the perfect solution for anyone looking for an all-purpose blockchain oracle. Unfortunately, it seems that this project has been discontinued seeing how all of the related social media accounts have ceased activity some time ago and none of the API documentation links on the official website work anymore. [9, 7]

V. Solution proposal

Based on our findings, we can conclude that there currently isn't a single existing commercial oracle that gives users the freedom to access any API of their choice without compromising performance or simplicity of integration. We want to create a solution isn't too specific in what it does and accommodates the needs of enough potential customers to warrant its existence while remaining simple and accessible to as many potential customers as possible. This is why we have decided to focus on a universal oracle, that gives users the ability to decide exactly what information they need for their decentralized application and how long they need it for.

Users will simply input a URL for their API as well as any necessary query parameters and HTTP headers, in addition to the length of the period they wish to make API calls for. After the user submits all information and authorizes the transaction, the process is complete and their subscription is created. From this point onward, until the subscription is terminated, oracles will query the relevant API. After the data is successfully relayed to the oracle smart contract, it will be subsequently forwarded to an address specified by the user. No further action beyond creating the subscription is needed on the side of the user. No changes need to be made to be made to the recipient contract either, provided it has a method that can receive and process the data.

A. Overview

Our solution will be comprised of multiple components, each with their own distinct function. The on-chain portion will be a single decentralized application whose purpose will be to process requests from clients as well as to receive data from oracle nodes. The off-chain portion includes the implementation for oracle nodes themselves and the web application that clients will interact with directly.

As previously mentioned, when oracles are a centralized, they can function as a single point of failure of the network and overall pose as a major vulnerability. To mitigate this, a mechanism needs to be put in place to help ensure the integrity and correctness of data. Hence, we have opted for a decentralized solution, utilizing a network of multiple oracle nodes, that will rely on a Proof of Stake (PoS) mechanism for consensus.

B. Oracle smart contract

The oracle smart contract represents the on-chain part of our solution. It keeps track of accounts for both the customers of the oracle service as well as the oracle nodes themselves. Customer accounts contain information about specific API's they wish to receive data from. Oracle node accounts store their respective stakes. The purpose of the smart contract is to oversee the operation of the oracle network and the data collection process. It relays information about what data to fetch to the oracle nodes and selects leaders whenever necessary. After data is successfully received from the oracle network, it is forwarded to an account specified by the customer.

C. Oracle node

Oracle nodes are responsible for the collection and aggregation of data. They communicate with the oracle smart contract as well as amongst themselves. The smart contract provides them with all necessary information in regard to what API calls to make and how often, as well as notifying oracles when a new cycle starts, and a new leader is elected. Whenever oracles query an API, they relay the information to the leader of the current cycle and await the leader's response containing the version of collected data chosen to be sent to the on-chain contract. At this point oracles can either accept or reject this data before it is sent to the oracle smart contract.

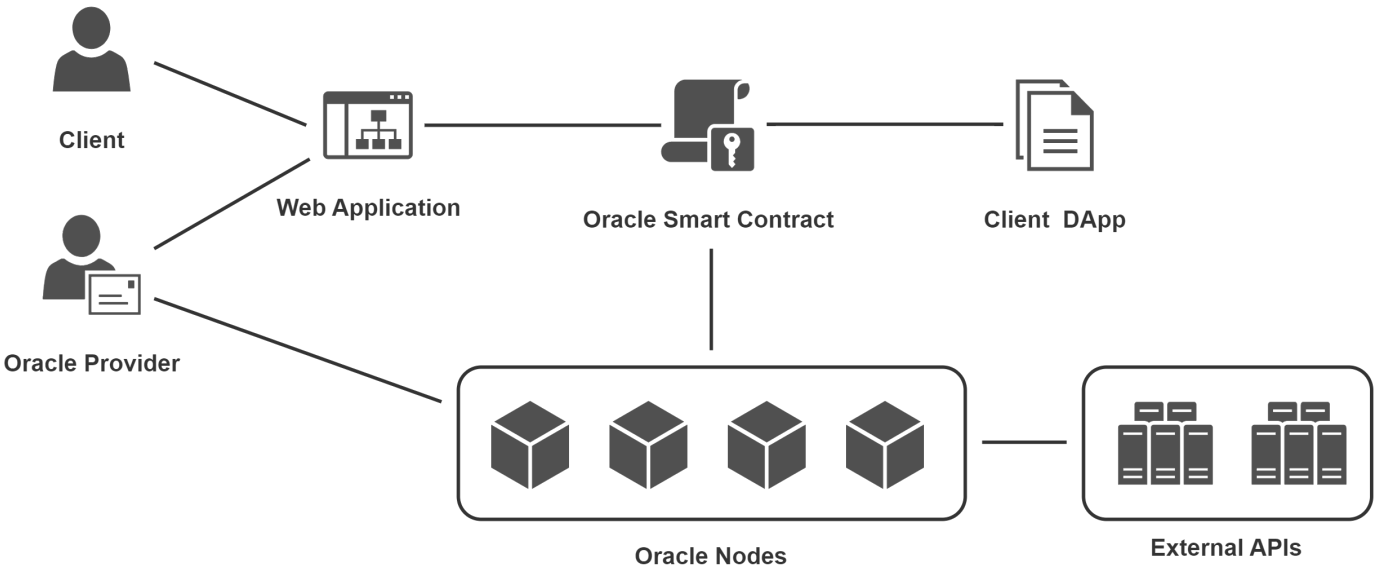

Fig. 1. Solution structure visualization

D. Web application

The part of the solution that a customer will interact with to manage their subscriptions (a subscription refers to a series of rounds of data collection from an API by the oracle network on behalf of the customer). A simple web application containing a form page and an overview page. The overview page will contain a list of all ongoing and expired subscriptions for a given user. Each entry will also contain some basic information such as the name of the API, the frequency of API calls and the date of expiration for each subscription. The form page will allow users to purchase new subscriptions. First users will fill out a form containing fields for all necessary information. This page will also contain a display with the price of the subscription, calculated based on the total number of calls. Upon entering all the necessary information users will be able to finalize their subscription by paying. The subscription becomes active immediately after payment is received by the oracle smart contract.

The following input fields will be present on the form page:

Address - single text input field, a Solana address that the data will be sent to upon being received by the oracle smart contract

Subscription length - single numeric input field for the total number of API calls a customer wishes to make, calls will be made once per minute

API URL - single text input filed for the URL of the API that will be queried by the oracle nodes

Parameters - single text area input that accepts any valid JSON object containing various parameters required for querying the API specified above.

VI. Oracle smart contract implementation

When deciding on which platform to develop our solution for, we looked for a blockchain that best aligns with the core principles of simplicity and performance that we envisioned for the project. Its high throughput, fast transaction speeds and minimal transaction fees make Solana the ideal candidate. Additionally, using Solana also gives us the ability to write our contract in the Rust programming language, known for its high performance and memory-safety.

We have also opted to use the **Anchor framework** to streamline the development process of our smart contract. Anchor provides a higher level of abstraction and a more intuitive API, which lets us bypass a lot of the more tedious aspects of writing code for Solana. It is also generally more robust and secure, decreasing the likelihood of errors and vulnerabilities occurring in our project. Lastly, Anchor automatically handles more mundane tasks such as serialization and deserialization under the hood, making the code far more clear and readable.

A. Accounts

Solana smart contracts, more commonly referred to as programs are technically stateless. That being said, we can still store state inside of accounts. Each account serves as a self-contained pocket of data with its own internal state and can be accessed via its public key. Each account is owned by a specific program and only that program is allowed to execute write operations on it. Other programs are only allowed to read the contents of the account.

Our contract implements two types of accounts. First, a central state account, that will hold all the relevant information about the application, The state account will handle all the necessary information about oracles and the status of the oracle network as a whole. All off-chain calls to our application will access the same state account. Second, we implement

a subscription account which holds all of the details of a given subscription. Each subscription is allocated its separate account upon being created.

B. Instructions

All write operations called by off-chain APIs are handled via program instructions. We can think of each instruction as a method inside the program that can be called by an outside API to perform an action. Instructions can be used to create and modify accounts, as well as to transfer funds.

Our program will contain the following instructions:

Initialize - only called once at the very start to initialize the smart contract and create a state account

AddSubscription - called whenever a user registers a new subscription, takes in the subscription parameters as well as the recipient's address and stores them in a new subscription account

AddOracle - registers a new oracle inside of the state account

ReportData - called by an oracle to report data to the program, data is subsequently parsed and distributed to subscribers

C. Consensus

As mentioned before, our solution will utilize a decentralized network of oracles with the proof of stake mechanism for consensus. Whenever a new oracle enters the network, it must also deposit a stake. The network will operate in cycles and at the start of each one a leader is selected out of the available oracles. The chance of being selected is proportional to the size of their stake. Each cycle will contain a single series of API calls made by each oracle - (each oracle will make one call per active subscription). Oracles will then report their data to the leader, who will aggregate them into a single report. This report is subsequently distributed to the other oracles who can choose whether to sign off on it or not. The report is then submitted back to the contract. The contract then evaluates the report and if the report received signatures from at least two thirds of the network, the report is accepted and the leader is rewarded, otherwise the leader's stake is slashed and the report is rejected. Too many rejections will result in the oracle being removed from the network. After this, the next cycle begins and a new leader is selected.

VII. Web application implementation

Our web application is implemented in the JavaScript framework Vue 3, also utilizing TypeScript for its numerous benefits such better readability as well as the ease of development and debugging it provides. Much like with our oracle nodes, TypeScript was the obvious choice thanks to its compatibility with the development kits for Solana and the Anchor framework. This is vital because all interaction with the smart contract happens via Solana RPC (Remote procedural call) which is provided by the Anchor SDK. Additionally, there is also a Solana wallet adapter available for the Vue framework, which we have decided to take full advantage of as it makes the overall experience more straight-forward and intuitive for our users.

The following conditions must be met for the web application to be able to function:

1) The oracle smart contract must already be deployed on-chain.
2) The IDL of the contract containing all of its methods and metadata must be provided to the web application in the form of JSON.
3) The IDL must contain a valid Solana address belonging to the currently deployed version of the contract.
4) A connection to the Solana network must be established.
5) A Solana wallet must be installed in the browser that is being used to access the web application.

A. Home page

The home page is displayed when the user opens the app. They are then prompted to connect their Solana wallet. Thanks to the aforementioned wallet adapter, the wallet functionality is seamlessly integrated into our web application. The only requirement is for the user to have a Solana wallet within the browser they are using to access our app. It should be noted that Phantom Wallet is currently the only wallet our app supports. Once connected, the user gains access to the other parts of the application.

B. New subscription page

The "New Subscription" page contains form fields for all the information necessary for creating a subscription. Once a user fills out the form and submits the form is properly validated to make sure that as much of the information as possible is correct. If contents of the form pass the validation a pop-up window is opened and the user is prompted to pay for the cost of the subscription. If the user has enough funds in his account and the transaction is successful the oracle smart contract registers the subscription. From this point onward, the never has to interact with the web application again and their dApp will continue receiving data from off-chain APIs for the remainder of the subscription's duration.

C. All subscriptions page

The "All Subscriptions" page contains a list of all active and finished subscriptions belonging to a certain user. As the page is loaded, the app makes a call to the oracle smart contract and fetches all subscription accounts linked to the public key of the currently selected wallet. All results, assuming there are any are then neatly displayed in a table. Each entry represents a single subscription and contains all relevant information such as the recipient's address as well as the remaining duration of the subscription. Finished subscriptions are still listed and can be identified by their duration being at 0.

VIII. Oracle node implementation

Oracle nodes will be implemented in TypeScript and will run on the node.js runtime. TypeScirpt was chosen for many

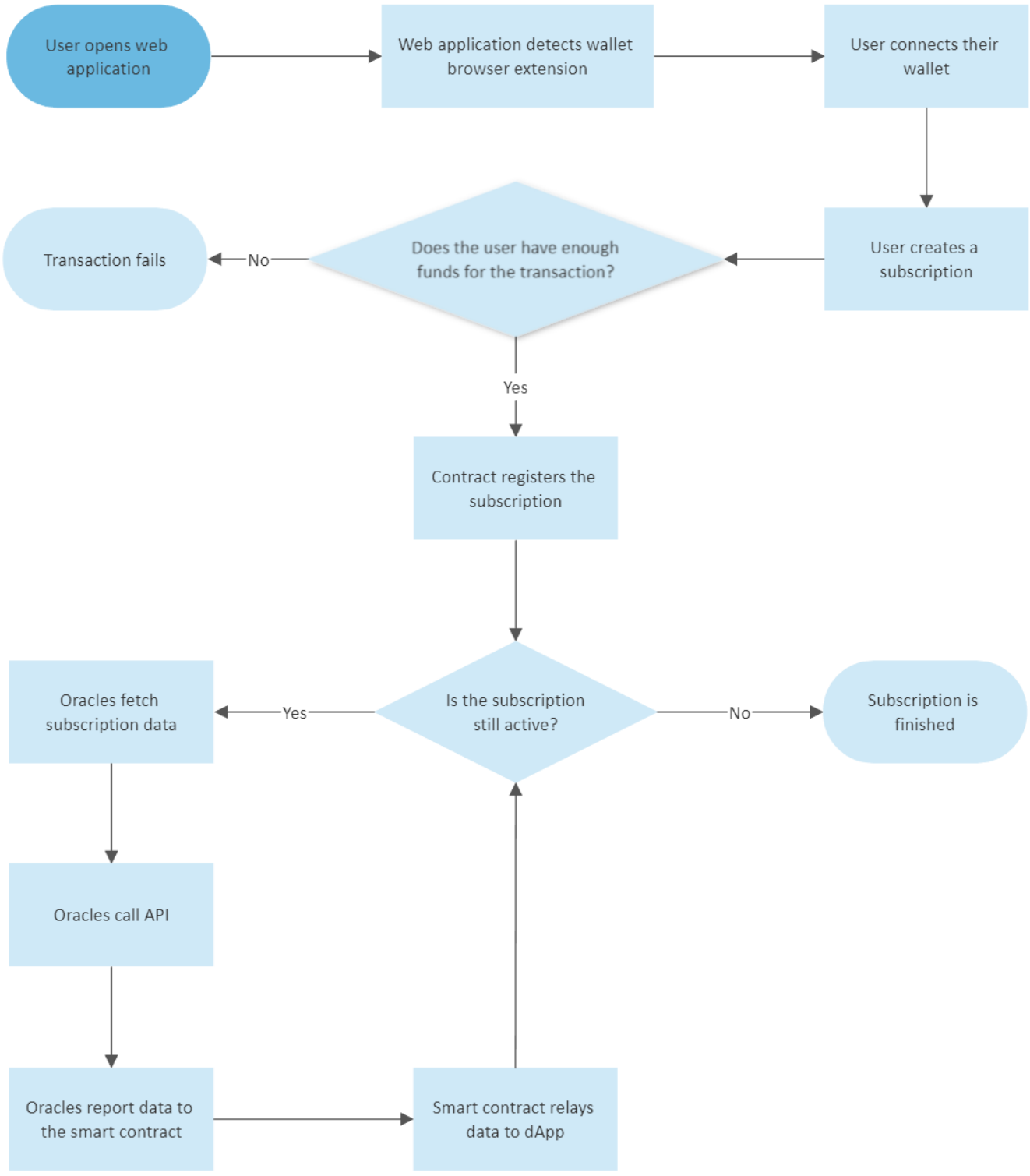

Fig. 2. Solution control flow visualization

of the same reasons that were outlined in the section about the implementation of the web application. Oracle nodes will also use the Solana RPC in the same way as our web application to communicate with the smart contract. Additionally they will also use the promise-based HTTP client Axios to make API calls. We have opted to use Axios instead of the built-in fetch API because of its robustness and extra features which may be useful in the future.

The role of the oracle nodes is relatively simple. At the start of each cycle they fetch all subscriptions from the smart contract and call endpoints for all active subscriptions. Afterwards they report their data to the leader. After the leader sends them the final version of the report they compare it to their own data and decide on whether to accept it and sign it with their public key or reject it. Regardless, they then send the data back to the leader afterwards. The leader aggregates all signatures and sends the report to the smart contract.

IX. Evaluation

Our solution offers a service that currently has very little representation in the blockchain oracle ecosystem in more ways than one. Our primary contribution to the ecosystem is the providing users the ability to set up subscriptions to any off-chain API of their choice. This feature currently has very limited availability across all blockchains with only 2 oracle platforms providing it in a form similar to ours. First being Chainlink which only offers API calls on Ethereum and second being Provable which does not offer decentralization. Neither solution offers the ability to set up a long-term subscription either, with both only giving users the ability to make a single API call at a time with a separate transaction being required to for each additional call. Our solution on the other hand is fully decentralized and lets users create a subscription which will continue supplying off-chain data to their decentralized application without any further action on their part being needed. Additionally, our soultion is also incredibly easy to integrate with existing Solana dApps and requires very few changes to existing codebases to be made.

It is also worth noting that there are currently only 2 successful commercial oracle providers on the Solana blockchain, Chainlink and Pyth. Both of which only cover the standard price feed use case at the moment. Not only are we expanding the somewhat limited range of oracle options on Solana, we are also providing developers a tool that no one else is offering.

X. Conclusion and future work

The need for trusted sources of outside information will only continue to grow as the adoption of blockchain technology and decentralized finance become more widespread. We believe that our universal oracle will help fill in some of the gaps that are currently present in the ecosystem and satisfy a unique demand in the market that no other oracle provider is currently focusing on.

The implementation detailed in the previous sections is fully functional, however there is still room for additional features and improvements that have yet to be implemented. The next logical step would be to build upon the existing web application to add a page which would allow oracle providers to register their own deployed oracle nodes into the network. Additional functionality within the oracle smart contract would of course be required to fully support this new feature. This could then be expanded on even further in the future with the release of a fully-fledged software development kit that would improve developer experience as well as provide a degree of standardization for oracle node implementation. Finally, further efforts could be made towards improving performance and overall scalability of the oracle network, once all features are fully implemented and thoroughly tested.

References

[1] Abdeljalil Beniiche. “A Study of Blockchain Oracles”. In: *CoRR* 2004.07140 (2020). DOI: 10.48550/ARXIV.2004.07140.

[2] Breidenbach. “Chainlink Off-chain Reporting Protocol”. In: (Feb. 2021). [online] cited 31.3.2023. URL: https://research.chain.link/ocr.pdf.

[3] Hamda Al-Breiki et al. “Trustworthy Blockchain Oracles: Review, Comparison, and Open Research Challenges”. In: *IEEE Access* 8 (2020), pp. 85675–85685. DOI: 10.1109/ACCESS.2020.2992698.

[4] FERRULLI. *On demand decentralized oracles for blockchain: a new Chainlink based architecture*. [online] cited 31.3.2023. Feb. 2022. URL: https://etd.adm.unipi.it/t/etd-02062022-124127/.

[5] Sin Kuang Lo et al. “Reliability analysis for blockchain oracles”. In: *Computers & Electrical Engineering* 83 (2020), p. 106582. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2020.106582.

[6] Léonard Lys and Maria Potop-Butucaru. “Distributed Blockchain Price Oracle”. Research Report. working paper or preprint. Apr. 2022. URL: https://hal.archives-ouvertes.fr/hal-03620931.

[7] *Mobius Network*. [online] cited 31.3.2023. URL: https://mobius.network/;.

[8] Amirmohammad Pasdar, Zhongli Dong, and Young Choon Lee. “Blockchain Oracle Design Patterns”. In: *CoRR* 2106.09349 (2021). DOI: 10.48550/ARXIV.2106.09349.

[9] Amirmohammad Pasdar, Young Choon Lee, and Zhongli Dong. “Connect API with Blockchain: A Survey on Blockchain Oracle Implementation”. In: *ACM Comput. Surv.* (Oct. 2022). ISSN: 0360-0300. DOI: 10.1145/3567582.

[10] *Provable*. [online] cited 31.3.2023. URL: https://provable.xyz/.

[11] Yinjie Zhao et al. “Towards Trustworthy DeFi Oracles: Past,Present and Future”. In: *CoRR* 2201.02358 (2022). DOI: 10.48550/ARXIV.2201.02358.