Oracle System Design

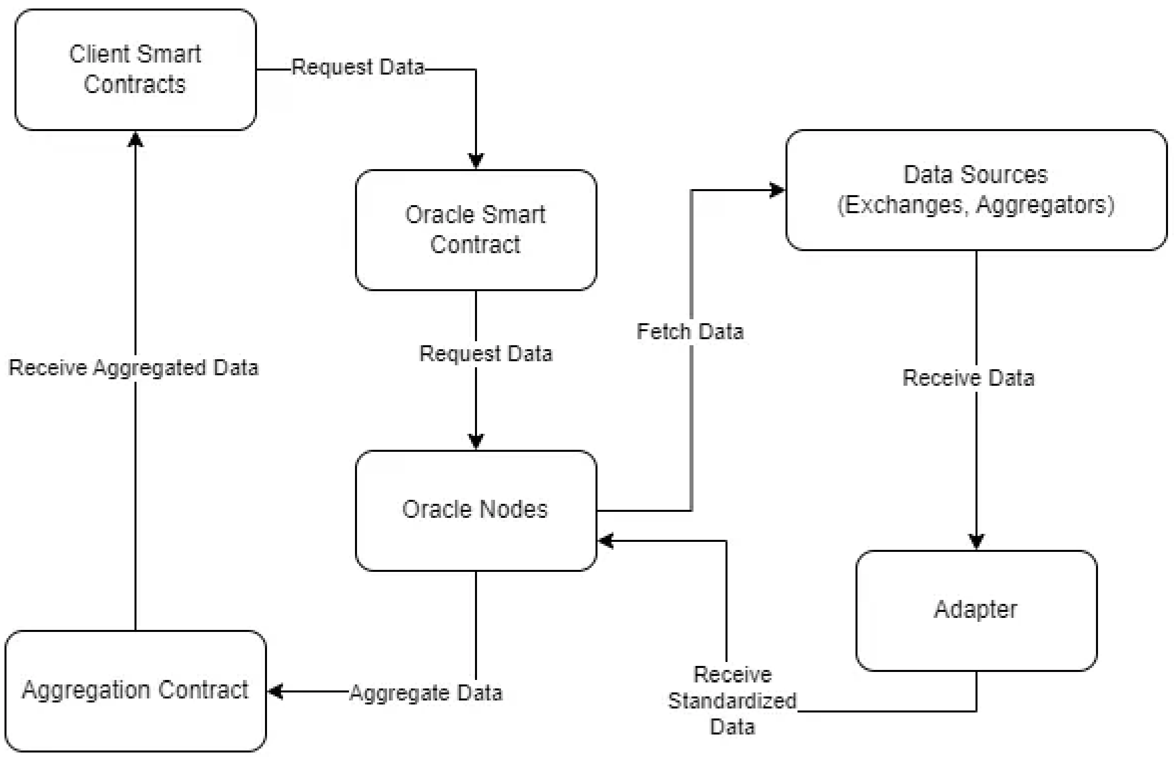
Firstly I'd have to clarify that I know little about web3 or blockchain oracle in general. So please bear with me if you spot mistakes that stem from misunderstanding. I am eager to learn more about oracle system in the job with minimal guidance. I have proven that I can learn difficult concepts in many other fields e.g. novel machine learning architecture, Calculus, computer vision, competitive programming, complex data structures and algorithms, etc. So I estimate that It will take me a few days in the job for me to acquire a good understanding of oracle system in general.

Putting that aside, here's my attempt at designing the oracle system:

High-Level Overview:

- It will be a decentralized network of nodes, similar to Chainlink, to make it reliable and secure.
- We can add new data sources easily by making the architecture modular.
- We will include a governance module to manage oracle node performance and data quality.

Following is the diagram of the overall architecture:



Interface with Data Sources:

- The oracle system will use external adapters to call various APIs provided by decentralized exchanges (Uniswap), centralized exchanges (Binance), and price aggregators (CoinMarketCap, CoinGecko).
- Each data source will have a corresponding adapter that converts the data into the same format for the oracle system. The adapter pattern can be found here: https://refactoring.guru/design-patterns/adapter

Here is example code of how we can fetch price data from a DEX using `web3.py`:

```python
from web3 import Web3

# Connect to an Ethereum node
w3 = Web3(Web3.HTTPProvider('https://mainnet.infura.io/v3/YOUR_INFURA_API_KEY'))

# Uniswap contract address and ABI
uniswap_contract_address = '0xUniswapContractAddress'
uniswap_contract_abi = json.loads('YourContractABI')

# Instantiate the contract
```

```
contract = w3.eth.contract(address=uniswap_contract_address,
abi=uniswap_contract_abi)

# Assuming the contract has a function `getPrice` to get the latest price of a
token
# You need to replace 'getPrice' with the actual function name and parameters based
on the ABI
price = contract.functions.getPrice('0xTokenAddress').call()

print(f"The latest token price from Uniswap is: {price}")
```

Here is example code to fetch price data from CoinGecko (a price aggregator that can be queried similarly to CEXs via HTTP requests):

```
import requests

# Endpoint URL for CoinGecko API to get the price of bitcoin in USD
url = 'https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currencies=usd'

# Make a GET request to the CoinGecko API
response = requests.get(url)

# Parse the JSON response
data = response.json()

print(f"The latest Bitcoin price from CoinGecko is: {data['bitcoin']['usd']} USD")
```

Query Mechanism:

- Clients will interact with the oracle via a smart contract interface.
- Clients can specify the asset and the type of data they need, e.g., latest price, volume, etc.
- A request from a client contract will trigger an event that oracle nodes listen to.

Data Format:

- We will provide data in JSON format, which is widely used and can be understood by almost every programming language.
- The JSON object will contain the asset name, price, timestamp, volume, and source identifier.

Here's an example of the standardized JSON data we can use:

```
{
  "data": {
    "asset": "ETH",
    "price": "1578.45",
    "volume_24h": "450000.1234",
    "change_24h": "3.45",
    "last_updated": "2024-03-07T14:52:00Z",
    "source": "Binance"
  },
  "timestamp": "2024-03-07T14:53:00Z",
  "status": "success"
}
```

Transmitting Prices:

- Once the data is retrieved and aggregated by the oracle nodes, it will be transmitted back to the client's smart contract.
- A consensus mechanism will ensure that only verified data is transmitted to prevent data manipulation by any single point.

Data Submission by Clients:

- Clients will have the ability to submit data requests through a smart contract method call.
- This submission process will require the use of the system's native token to prevent spam and allocate network resources efficiently.

Accuracy and Consistency:

- We will use multiple nodes to fetch the same data and then aggregate the results. This should make it more accurate.
- We can use outlier detection algorithms to identify and remove phishy data points e.g. IQR method.
- Data sources will be weighted based on historical performance and reliability. We will need to track variables such as response time, uptime, standard deviation of its price from the mean price of all sources over time. Then we need some kind of simple heuristic to compute the weighted score.

Robustness and Resilience Strategies:

- The system will have redundancy built-in by using multiple nodes for each data request.
- There will be a fallback mechanism for data sources, so if one is dead, the system can automatically switch to another source. If a node detects that the primary source is dead or provides data that diverges a lot from other sources, it automatically queries the secondary source.

Documentation:

- We will provide A complete API documentation, detailing request and response formats, usage examples, and common troubleshooting. We will need to use tools like Swagger UI, Sphinx, or Postman.
- Architectural diagrams will be included to illustrate the flow of data and interactions between components. Use tools like Lucidchart, Draw.io, or Microsoft Visio. If we want to keep diagrams versioned along with our code, we could use a simple tool like Graphviz, PlantUML, or Mermaid. We will be able to write a diagram and commit it into a Git repository.
- The documentation will be versioned and updated regularly.