Transaction Broadcaster Service System Design

System Requirements

Functional Requirements

- Expose an API endpoint to receive broadcast requests
- Sign transactions using a private key
- Broadcast signed transactions to a blockchain RPC node
- Handle transaction failures with an automated retry mechanism
- Persist transaction states to ensure reliability in case of service restart
- Admin dashboard to monitor and manually retry failures

Non-Functional Requirements

- High Availability: The service should be operational at all times
- Scalability: Should be able to handle large amount of transactions
- Fault tolerance: Transaction status should not be lost even if broadcaster service restarts

High Level Overview

The system is designed to handle the signing and broadcasting of transactions to an EVM-compatible blockchain network in a scalable, robust, and durable manner. To ensure all these prongs are handled, the system is decoupled into four main components:

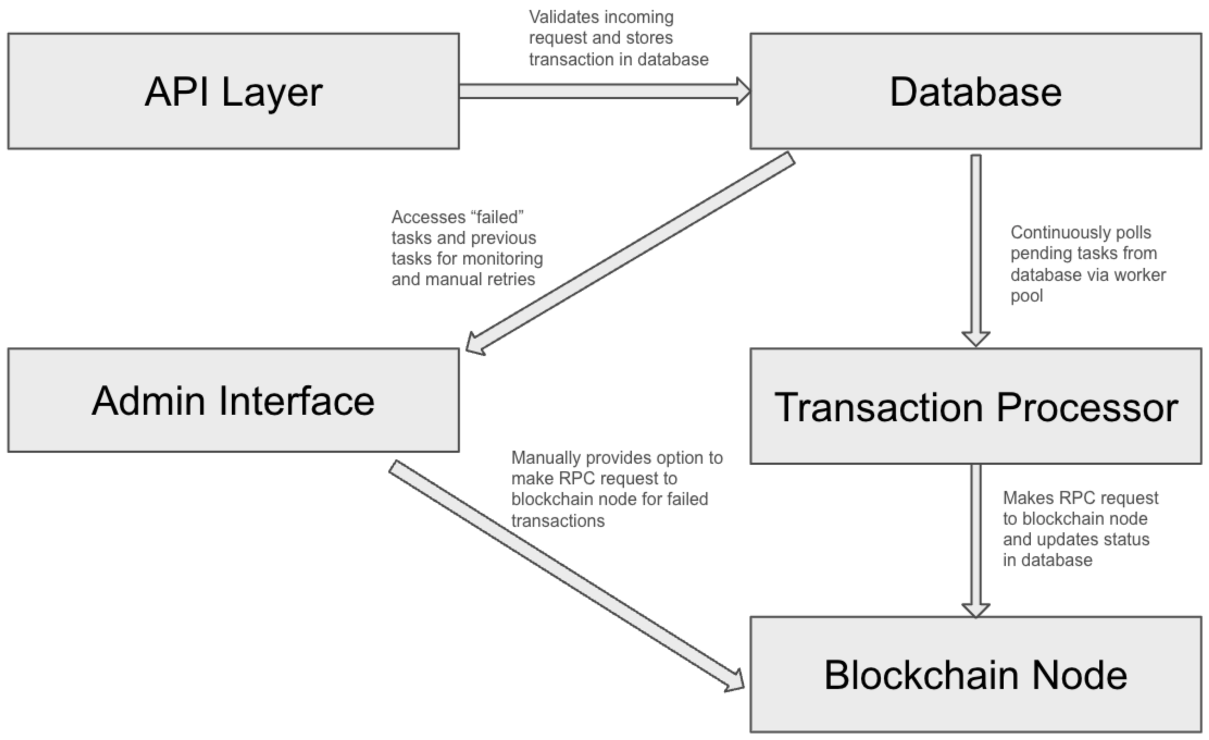
API Layer: Handles incoming requests, validates them, and stores transactions as objects in the database.

Database: Persists transaction states, ensuring durability and enabling recovery after restarts.

Transaction Processor: Processes transactions from database asynchronously, signing and broadcasting them to the blockchain network. Works independently from the API layer.

Admin Interface: Provides a UI/API for monitoring and managing transactions, including manual retries for failed transactions.

This decoupling allows the system to scale horizontally, as multiple instances of the Transaction Processor can work concurrently. Furthermore, it ensures that even if the API layer is down or restarts unexpectedly, the transactions will still be eventually broadcasted successfully as it would be handled by the transaction processor. The database ensures that no transactions are lost, even if the service restarts unexpectedly, while the admin interface provides visibility and control for administrators. All these come together to ensure the scalability and robustness of the system


API Layer
Validates incoming request and stores transaction in database
Database
Accesses “failed” tasks and previous tasks for monitoring and manual retries
Continuously polls pending tasks from database via worker pool
Admin Interface
Transaction Processor
Manually provides option to make RPC request to blockchain node for failed transactions
Makes RPC request to blockchain node and updates status in database
Blockchain Node

API LAYER

Endpoint: POST /transaction/broadcast
Input: JSON payload containing message_type and data.

Response:
HTTP 200 OK with a transaction_id if the request is accepted.
HTTP 4xx for invalid requests.
HTTP 5xx for internal server errors.

Example Request

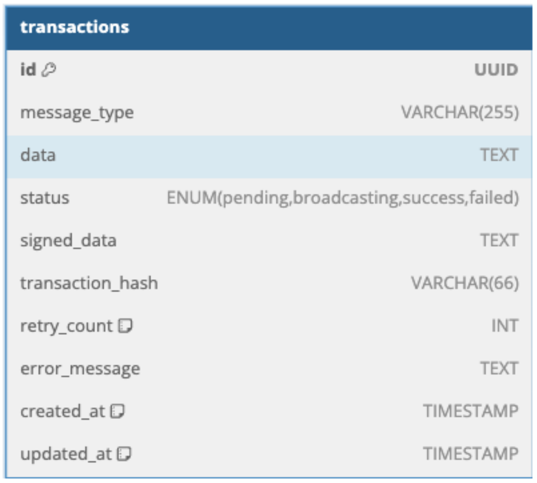
```
{
  "message_type": "add_weight(address _addr, uint256 _weight)",
  "data":
"0xd71363280000000000000000000000005eb715d601c2f27f83cb554b6b36e047822fb70a00000000000000000000000000000000000000000000000000000000000000fa"
}
```

Example Response

```
{
  "transaction_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "pending"
}
```

DATABASE

Purpose: Persists transaction states for durability and recovery.



transactions	
id	**UUID**
message_type	VARCHAR(255)
data	TEXT
status	ENUM(pending,broadcasting,success,failed)
signed_data	TEXT
transaction_hash	VARCHAR(66)
retry_count	INT
error_message	TEXT
created_at	TIMESTAMP
updated_at	TIMESTAMP

Schema:

- id: Unique identifier for the transaction.
- message_type: Type of the transaction.
- data: Original transaction data.
- status: Current state (pending, broadcasting, success, failed).
- signed_data: transaction after transaction processor signs data
- retry_count: Number of retry attempts.
- transaction_hash: to be updated after successful broadcasting to blockchain network
- Error_message: latest error message encountered for this transaction
- created_at: Timestamp of creation.
- updated_at: Timestamp of last update.

TRANSACTION PROCESSOR

- Purpose: Processes transactions and broadcasts them to the blockchain network.
- Implementation:
  - Continuously polls the database for pending transactions.
  - For each transaction:
    - Update the status to broadcasting.
    - Sign the transaction data (if not already signed).

        - Send the signed transaction to the blockchain node via RPC.
        - Handle the RPC response:
            - Success: Update the status to success and store the transaction hash.
            - Timeout or Failure: Increment the retry_count and update the status to pending (for retries) or failed (if max retries are reached).

ADMIN INTERFACE

- Purpose: Allows administrators to monitor and manage transactions.
- Implementation:
    - UI: A web-based dashboard showing a list of transactions with filters for status, retry count, etc.
    - API: Endpoints for retrying failed transactions and querying transaction details.

EXAMPLE WORKFLOW

- Request: A service sends a POST /transaction/broadcast request.
- Validation: The API layer validates the request and stores it in the database with a pending status.
- Transaction Processor: Polls the database for pending transactions, signs the data, and broadcasts it to the blockchain network.
- Retry: If the broadcast fails, the transaction is retried automatically.
- Persistence: The transaction state is updated in the database.
- Monitoring: The admin interface displays the transaction status