Here is the design a transaction broadcaster service considering all the requirements you provided. This design is constructed with scalability and robustness in mind.

High-level Overview

API Gateway: Receives client requests for broadcasting transactions and forwards them to the Broadcasting Service.

Broadcasting Service: Signs the transaction data, handles the broadcasting logic, and retries failed transactions.

Blockchain RPC Interface: Communicates with the EVM-compatible blockchain network through RPC.

Database: Stores information about the transactions (successful and failed) to provide persistence and a consistent state.

Admin Interface: Enables admins to manually retry failed broadcasts.

Monitoring & Logging: Provides visibility into the system.

Detailed Design

1. API Gateway

Receives the POST /broadcast\_transaction requests and forwards them to the Broadcasting Service. It also validates the input and handles basic authentication if needed.

2. Broadcasting Service

a. Transaction Signing

Upon receiving a request, the Broadcasting Service signs the data provided using a secure method like ECDSA. The private key for signing should be stored securely, either using an HSM or a secret management service.

b. Transaction Broadcasting

If the signing is successful, it sends an RPC request to a blockchain node to broadcast the signed transaction.

It then records the transaction in the database with a status of "Pending."

It follows the specified time-response model:

Handles non-responses within 30 seconds.

Deals with successful or failed responses within 20-30 seconds.

c. Retrying Failed Transactions

If a broadcast fails, it's marked as "Failed" in the database.

A background job is scheduled to retry the failed broadcasts. The retries can follow an exponential backoff strategy to minimize network congestion.

3. Blockchain RPC Interface

Interfaces with the EVM-compatible blockchain network.

Should support connection pooling and timeouts to handle the specific time-response behavior stated.

4. Database

Stores the transaction data, including the status ("Pending", "Successful", "Failed").

It should support transactions to ensure consistency.

It helps in resuming pending broadcasts even after unexpected restarts.

5. Admin Interface

Provides a UI for admins to see the list of transactions.

Allows admins to retry failed broadcasts manually.

6. Monitoring & Logging

Monitors the system's health and performance.

Logs crucial information for debugging.

Technology Choices

API Gateway: Nginx, AWS API Gateway, or other API management solutions.

Broadcasting Service: Implemented in a language like Go, Java, or Node.js.

Database: Relational DB like PostgreSQL or MySQL.

Final Thoughts

Implementing rate limiting at the API Gateway level would be wise to control the number of requests.

Security considerations must be made for the storage and use of private keys for signing.

Redundancy and load balancing can be implemented at the Broadcasting Service and Blockchain RPC Interface levels to handle high loads.

Regular backups of the database are vital to ensure that no transaction data is lost.

Diagram

Here's a simple illustration of the design:

A screenshot of a computer

Description automatically generated

The Admin Interface directly interacts with the Database to allow for manual retrying and monitoring of the transactions.