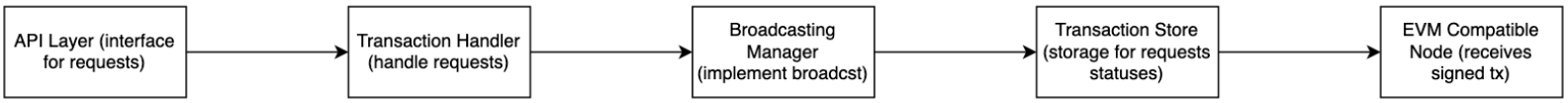
System Design

High-Level Architecture

The system I've designed consists of the following layers and components,

1. **API Layer** -- exposes the endpoint for broadcasting transactions and serves as the primary interface for requests.
2. **Transaction Handler** -- responsible for managing the lifecycle of transactions, including signing and broadcasting.
3. **Broadcast Manager** -- implements the broadcast process, including handling retries and failures.
4. **Transaction Store** -- a persistent storage to record transaction statuses and facilitate monitoring.
5. **Monitoring Dashboard** -- a user interface for admins to view transaction statuses and retry failed broadcasts.



Drawing of the System

API Layer

Response Codes

- `200 OK`: Accepted the transaction for processing.
- `400 Bad Request`: Invalid request format or missing fields.
- `500 Internal Server Error`: An unexpected error occurred while processing the request.

Transaction Handler

Upon receiving a broadcast request, the transaction handler will:

1. **Validate Input**: Ensure the request contains the necessary fields.
2. **Sign Transaction**: Use a cryptographic signing method to sign the `data` field.
3. **Store Transaction**: Save the initial transaction status (e.g., pending) in the transaction store.

-

Broadcast Manager

The broadcast manager will be responsible for the following:

1. **Broadcast Transaction**:
    - Make an RPC call to an EVM-compatible blockchain node to broadcast the signed transaction.
    - Handle the potential outcomes of the RPC call:
        - **Success (200)**: Mark the transaction as successful.
        - **Failure (400/500)**: Retry the broadcast based on exponential backoff
        - **Timeout (30+ seconds)**: Treat as a failure and retry.
    - **Retry Logic**: Implement a maximum retry limit (e.g., 3 attempts). If all retries fail, mark the transaction as failed.
2. **Asynchronous Processing**: Use asynchronous processing (e.g., message queues) to handle broadcasts without blocking the API response.

Transaction Store

The transaction store will persist transaction data to enable recovery and monitoring. Key fields include:

- `transaction_id`: Unique identifier for the transaction.
- `message_type`: The type of transaction being processed.
- `data`: The original data sent for the transaction.
- `status`: Current status (`pending`, `success`, `failed`).
- `created_at`: Timestamp of when the transaction was created.
- `updated_at`: Timestamp of the last update to the transaction status.

Monitoring Dashboard

The dashboard will provide the following features:

- Display all transactions with their statuses (pending, successful, failed).
- Allow an admin to manually retry a failed transaction at any point in time.
- Enable searching and filtering transactions by status, date, or type.

Flow of Request Handler

1. The service receives a POST request to `/transaction/broadcast`.
2. The transaction handler validates the request input.
3. The handler signs the transaction and stores it in the transaction store.
4. The broadcast manager attempts to broadcast the transaction.
5. If successful, update the transaction status to `success`.
    - If failed or timed out, initiate retries up to a predefined limit.
6. Admins can view the status (pending, successful, failed) of all transactions through the monitoring dashboard.

System Considerations

Scalability

To improve scalability:

- Use load balancers to distribute incoming requests across multiple instances of the broadcaster service.
- Deploy multiple instances of the transaction handler and broadcast manager to handle high volumes of transactions, to scale horizontally.
- If the transaction store becomes a bottleneck, consider sharding the database to distribute load.

Robustness

1. Comprehensive error handling which captures and logs unexpected issues ensure graceful degradation of service.
2. Use a reliable database for the transaction store to prevent data loss during service restarts or crashes.