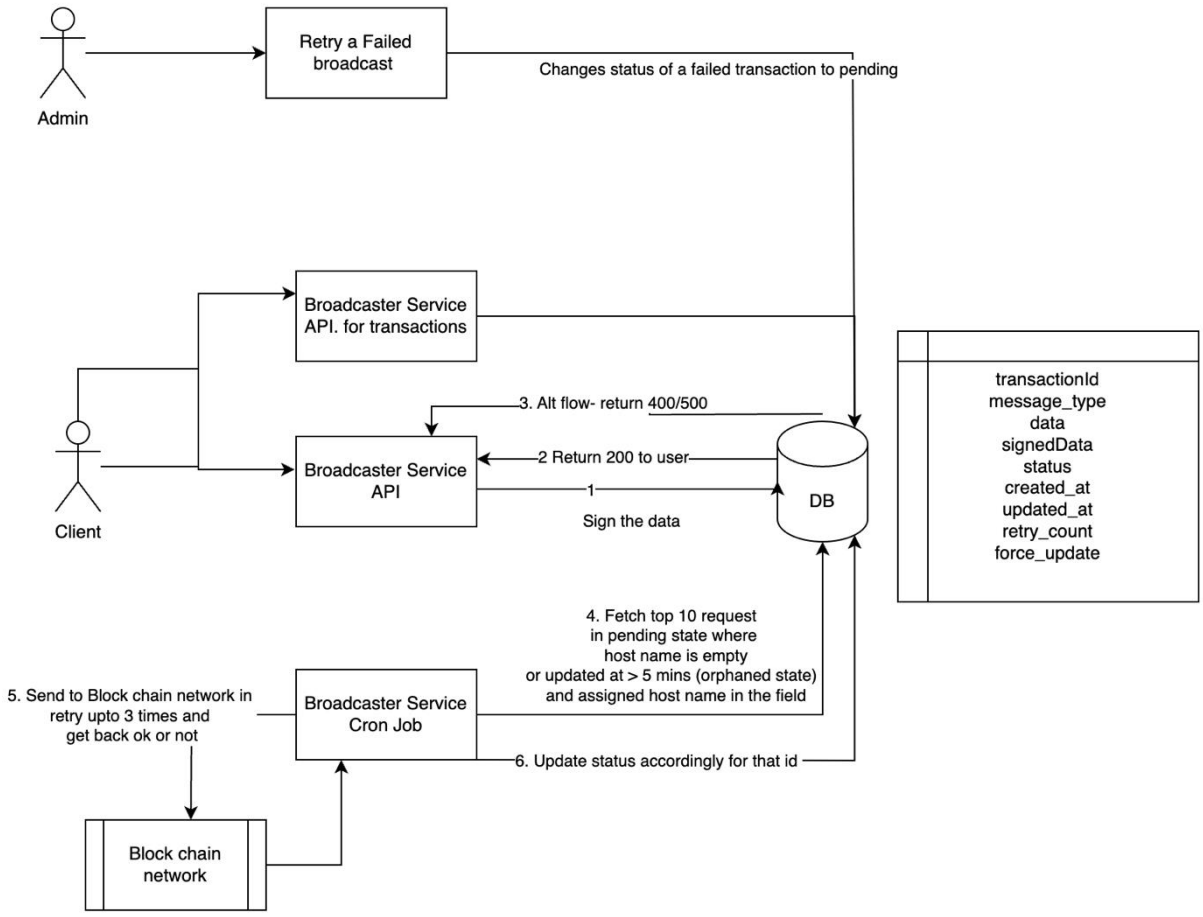
Transaction Broadcaster Service

The Transaction Broadcaster Service is designed to reliably broadcast transactions to an EVM-compatible blockchain network. The system ensures resilience, scalability, and robustness by incorporating asynchronous processing, retry mechanisms, persistence, and monitoring. Below is the comprehensive design of the service, retaining all necessary sections including **Key Architectural Choices**.

High Level Architecture Diagram



High-Level Architecture Description

The architecture consists of the following components:

1. **API Layer**

We would have 3 API i.e BroadcasterService API , Showcase Transaction API and Retry Transaction API for the admin. Logic for these APIs maintained in BroadcasterService

2. **Persistent Storage**

We would utilize DB as our persistant storage and would use it as the main source of truth.

Column Name	Data Type	Constraints	Description

transaction_id	UUID	Primary Key	Unique identifier for each transaction.
Message_type	TEXT	NOT NULL	Type of transaction/message.
data	TEXT	NOT NULL	Original transaction data.
signed_data	TEXT	NULLABLE	Signed transaction data for blockchain.
transaction_status	VARCHAR(50)	NOT NULL	Status of the transaction (PENDING, FAILED, COMPLETED).
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp of when the record was created.
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	Last updated timestamp.
host_ip_address	VARCHAR(50)	NULLABLE	IP address of the host processing the transaction.
Force_update	boolean	NOT NULL	Updated by admin flag
Retry_count	INT	DEFAULT 0	Tracks how many retry attempts have been made.

3. **Cron Job**
This would be a separate service that can be used to fetch the records from transaction DB and broadcast it to blockchain network via GRPC.

Interactions:

Client with the System
Client will have interaction with the system via 2 API:
- Post a Request : POST /transaction/broadcast
- Get the details of the transaction GET /transaction/broadcast/{transaction-id}

Client will send a post request to the system. The system will verify the request body and all its param. Incase of a bad request, 4xx response would be sent back (refer **API signatures** for details)

System will sign the data and store it in the DB with status Pending.
If the storage is successful, the system will return back the transaction-id to the client with status code 200.
Incase of any issue, the system would return back status code 5xx.

Client can also see the list of transaction done by them using the GET API as well as search the status of the transaction by the id provided to them when they posted the request.

Cron Job with the DB
The Cron job system would fetch the transaction from DB that needs to be processed. It will be fetched in short batches to ensure that cron job don't run for long time and finish quickly. Also since this system is at an initial phase, it make sense to make DB as the source of truth and not Message Queue. Reason being that MQ would attribute to additional cost and may act as an additional point of failure.
With Cron job being an independent system, we can scale it up as much as we like to handle the traffic load.

It will first fetch top 10 records from DB where status is pending and host name is empty. This is to ensure that any other instance of the cron job has not picked up the same record for processing. Also it will also check for orphan records(records where the system restarted and not processed fully) by checking if any transaction is stuck in pending state for more than 5 minutes. It will also add its instance name (host_ip_address) in that record for tracking purpose.

Once fetched, we well broadcast it to the network via a GRPC call and map the response according to the transaction it. To parallelize the process we can use threads/goroutine as well in this case. Incase of failure we will retry it for 3 times before ending the GRPC calls.

We will then update the status, updated_at for those transaction id.

Admin with the System
Admin can connect with the system to retry any transaction they like.
API : /transaction/retry
For this they will have access to an API which will just request the transaction ID for them. The API would then just update the status of that transaction to "PENDING" and then the cron job would automatically pick it up for processing.

Key Architecture Choice:
- Breaking up the APIs into 1 service and Cron job into another service so that we can independently scale it based on the request and the load.
- Using DB instead of MQ/KAFKA for cost cutting , reducing point of failure and DB can handle loads of transaction easily for this stage. If required, this can be enhanced at a later stage to add other components.
- Also Cron job can be implemented in a distributed manner thus making it quite strong , system/db can handle multiple cron jobs at once , through db read locks for instance.
- We will also use Read replicas of DB specifically for read request to separate the load of writes and read for better load handling.
- Retry count has been taken as 3 to ensure that we try multiple time but also ensure that cron job finishes quickly. If required the transaction can be manually retried. Also

since this is initial phase of system, we are not going with too much complex architecture like exponential backoff.

- Circuit breaker would be used for the GRPC transaction to ensure we don't hit broadcast the transaction when the chain is down also ensuring that cron job finish quickly.
- All fail safe options have been taken care of and error handling is done to ensure the state of transaction can be known.

Security Considerations

1. Instead of using HTTP. , use HTTPS (with TLS 1.2 or above) to ensure data in transit remains protected.
2. Also we should have rate limiting for the number of requests sent to the POST /transaction/broadcast or use a third part tools for DDOS prevention.
3. PII data / sensitive database information should be encrypted using AES-256 as per PDPA guidelines
4. MFA based login for users and admin and use of Digital Signatures can be used for tracking the user actions (in case of an attack) related to each transaction.
5. Audit trail should be maintained for proper logging using ELK(Elastic Search Logstash Kibana / Splunk) stack so that any errors/attacks can be traced and monitored.
6. **Secure Access to APIs**: Use API keys, JWT tokens, or OAuth 2.0 for authenticating API requests.

API signatures:

API Name	Endpoint	Method	Request	Response
Broadcaster Service API	/transaction/broadcast	POST	Body:	Success:
			{	{
			"message_type": "string",	"transaction_id": "string",
			"data": "string"	"status_code": 200
			}	}
				Error:
				{
				"error": "string",
				"status_code": 4xx/5xx
				}

Showcase Transaction API	/transaction/broadcast/{transaction-id}	GET	Path Parameter:	Success:
			transaction-id	{
				"transaction_id": "string",
				"message_type": "string",
				"data": "string",
				"signed_data": "string",
				"transaction_status": "string",
				"created_at": "string",
				"updated_at": "string",
				"host_ip_address": "string"
				}
				Error:
				{
				"error": "Transaction not found",
				"status_code": 404
				}
Retry Transaction API	/transaction/retry	POST	Body:	Success:
			{	{
			"transaction_id": "string"	"message": "Transaction status updated to PENDING",
			}	"status_code": 200
				}
				Error:
				{
				"error": "Transaction not found or cannot be retried",

				"status_code": 4xx
				}