Sunday, 25 February 2024

System Design: Transaction Broadcast Service

Preface

The primary objective of this service is to accept broadcast requests, sign and broadcast transactions to an EVM-compatible blockchain network, and handle failures with retries. This design will focus on the software abstractions and components necessary to meet these requirements.

System Components

1. **API Gateway:** The entry point for all incoming requests. It authenticates and routes requests to the appropriate service components, ensuring that only valid requests reach the core system.
2. **Broadcast Request Handler:** This component processes all incoming broadcast requests via HTTP requests. It validates the request payload and transforms it transforms it into a domain-specific model for the rest of the system to understand.
3. **Transaction Signer:** This module takes the validated request data and uses a secure method to sign the transaction. The signing process involves generating a cryptographic signature based on the transaction's "data", ensuring the transaction's integrity and authenticity.
4. **Transaction Broadcaster:** After signing, this component will broadcast the signed transaction to the blockchain network by making RPC requests to a blockchain node and handling the various response scenarios as defined (success, failure, timeout).
5. **Retry Mechanism:** Integrated within the Transaction Broadcaster, this mechanism handles automatic retries for failed transactions. It can use the exponential backoff strategy to manage retry attempts, to avoid overwhelming the network.
6. **Persistence Layer:** Stores details about each transaction request, including its status (pending, success, failed), the transaction data, and any relevant timestamps. This enables the service to recover from failures and provides data for the admin interface.

7. **Admin Interface:** A web interface or API that allows administrators to view the status of transactions and manually retry failed broadcasts. This component interacts with the Persistence Layer to fetch transaction data and with the Transaction Broadcaster for retry operations.
8. **Monitoring and Logging:** This system monitors and logs the operations, errors, and performance metrics to help with diagnosing issues and ultimately making informed decisions about scaling and optimisation.

Workflow

Request Reception:

The API Gateway receives a POST /transaction/broadcast request and forwards it to the Broadcast Request Handler.

Validation and Transformation:

The Broadcast Request Handler validates the request payload and transforms it into a transaction model. If validation fails, it returns an HTTP 4xx response.

Signing:

The Transaction Signer signs the transaction, ensuring it's ready for broadcast.

Broadcasting:

The Transaction Broadcaster attempts to send the signed transaction to the blockchain. Since a success code is received within 30 seconds 95% of the time, the service will await a response for 30000ms before a timeout is called. It handles the various response scenarios accordingly:

BROADCAST SUCCESSFUL	BROADCAST FAIL / TIMEOUT
- updates the transaction status in Persistence Layer - returns HTTP 200 OK to the caller	- trigger Retry Mechanism

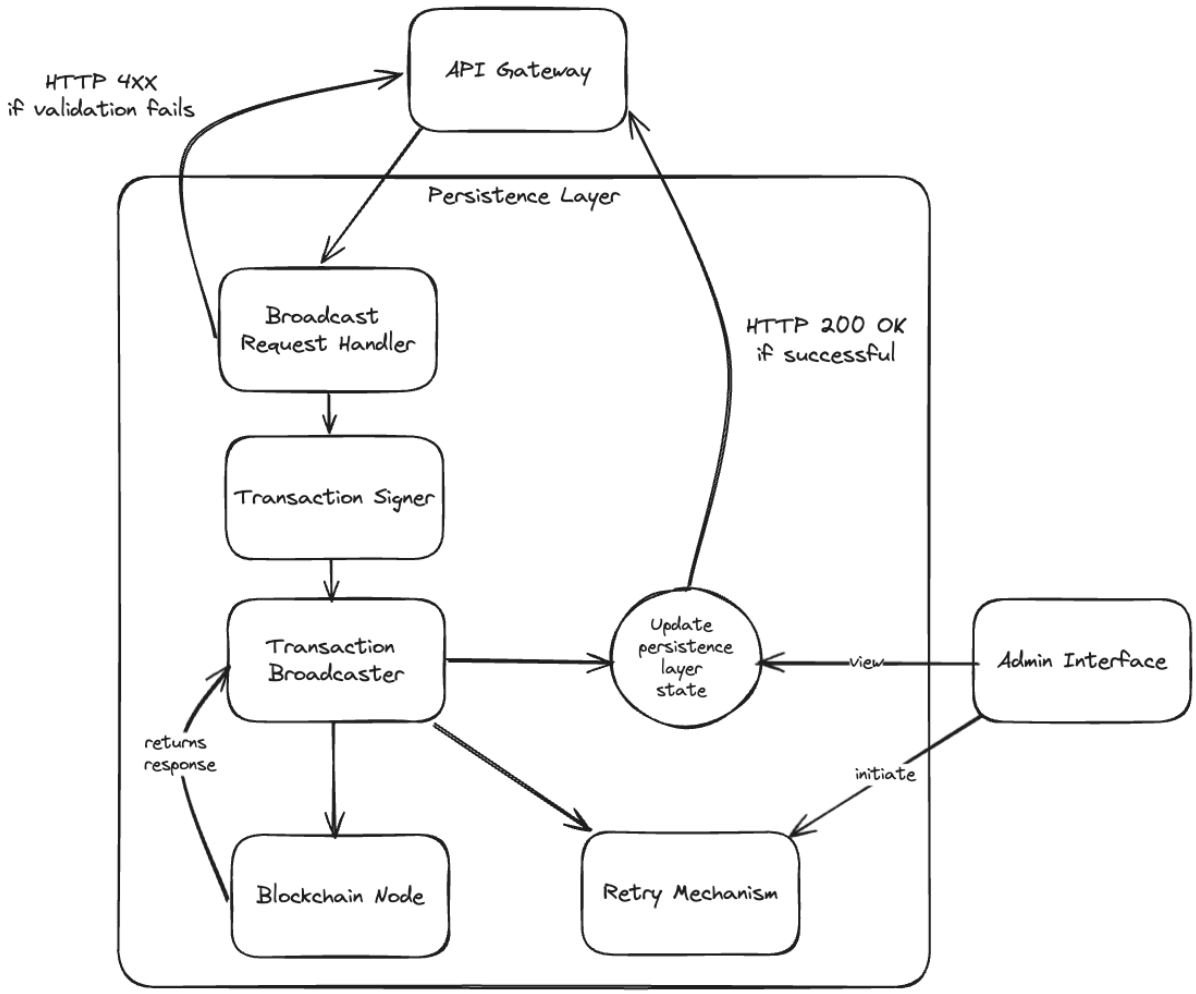
Retry Mechanism: For failed transactions, the Retry Mechanism schedules retries using an exponential backoff strategy. It will keep retrying until the transaction succeeds or a maximum retry limit is reached.

↕

Admin Interface: The admin can **view** transaction statuses and **manually initiate retries** for failed transactions. This interface will interact with the Persistence Layer for data retrieval and the Transaction Broadcaster for executing retries.

Persistence and Recovery: All transactions are persisted with their current status. In the event of a service restart, the system should recover and continue processing transactions based on their persisted states. (ie. should the broadcaster service restart unexpectedly, it should still fulfil the broadcasting of those transactions that have yet to been broadcasted).

Sample Workflow Diagram



Considerations - Key Management

1. **Secure Storage:** Private keys used for signing transactions must be stored securely, using hardware security modules (HSMs) or secure enclave technologies to provide tamper resistance, ensuring that the private keys cannot be extracted or misused.
2. **Access Control:** Strict access controls should be implemented to limit access to private keys. This includes both physical access to the hardware where keys are stored and logical access through software interfaces.

Considerations - Signing Process

1. **Algorithm Choice:** The cryptographic algorithm for signing transactions is determined by the blockchain network. For example, Ethereum uses the Elliptic Curve Digital Signature Algorithm (ECDSA) for its transactions. The algorithm of choice needs to provide strong security guarantees to be resistant to any possible attacks.
2. **Detached Signatures:** The signing mechanism should also generate detached signatures, so that the signature is separate from the transaction data. This allows for the verification of the transaction without needing to access the private key used for signing.

Considerations - Security

1. **Multi-Sig Transactions:** For added security, especially for high-value transactions, implementing multi-sig (multi-signature) transactions can provide an additional layer of security as the transaction would require signatures from multiple private keys, before it could be broadcasted. This reduces the risk of unauthorised transactions due to a compromised single private key.
2. **Rate Limiting:** Implement rate limiting on signing operations to prevent abuse and potential denial-of-service attacks. Rate limiting can also help in detecting unusual patterns that may indicate a security issue.

Considerations - Scalability

1. **Caching:** In some scenarios, similar transactions may be signed repeatedly. Caching the transaction templates for these operations may improve performance as it reduces the need to perform computationally intensive operations for each transaction over again.
2. **Parallel Processing:** The signing mechanism should be designed to handle multiple requests in parallel, leveraging multi-core processors to meet high demand without significant delays.
3. **Dynamic Load Distribution:** Implement load balancers to distribute incoming requests evenly across instances of the service components. This can help prevent any single instance from becoming a bottleneck.