High-Level Design (HLD) – Communication Aggregator System

Version: 1.0
Author: *abhishek pundir*

---

1. Introduction

This High-Level Design describes the architecture of the **Communication Aggregator System**, a backend solution that receives messages from clients, determines the correct communication channel (Email, SMS, WhatsApp), and forwards them to the appropriate delivery microservice. The system supports asynchronous communication, retry handling, and centralized logging for observability.

---

2. System Components Overview

The system is composed of **three independent microservices**, each running on its own port/container:

1. Task Router Service

- Provides a REST/GraphQL API (`/send-message`).
- Validates incoming payloads.
- Checks for duplicate messages using `messageId` or hashed payload.
- Applies routing logic based on the `channel` field.
- Publishes messages to RabbitMQ (or Kafka) based on channel.
- Logs key events to the Logging Service.
- Implements basic retry logic for temporary failures.

2. Delivery Services

Three independent microservices:

- **Email Delivery Service**
- **SMS Delivery Service**
- **WhatsApp Delivery Service**

Each delivery service:

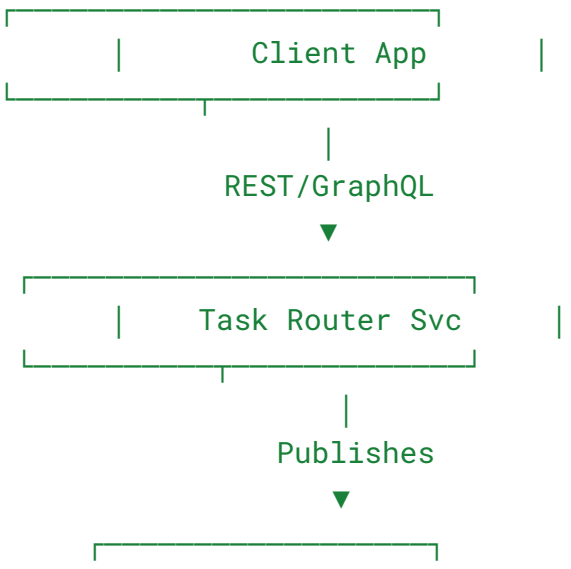
- Listens to its own queue/topic from RabbitMQ.
- Simulates message delivery (no external API required).
- Stores message status in a local SQLite/Postgres DB table.
- Emits structured logs containing `traceId`, `channel`, `status`, and timestamp.

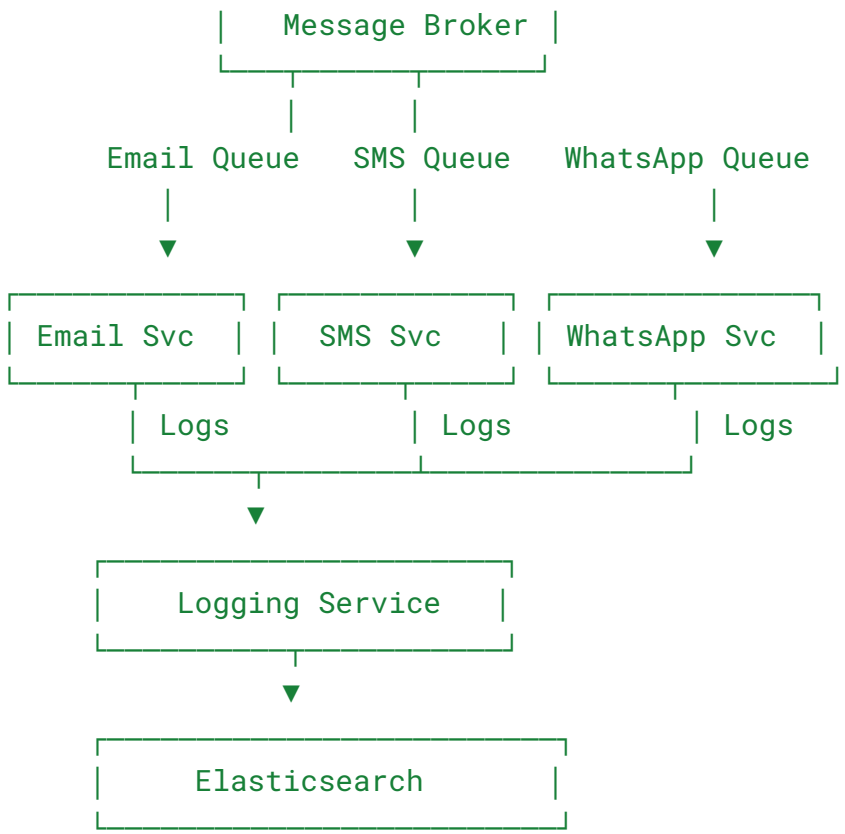
3. Logging Service

- Accepts logs from Task Router and Delivery Services (via HTTP or message queue).
- Indexes logs into **Elasticsearch**.
- Uses structured fields: `traceId`, `serviceName`, `step`, `message`, and `timestamp`.
- Supports visualization and troubleshooting via **Kibana** dashboards.

---

3. Architecture Diagram (Text Representation)





---

4. Data Flow (Step-by-Step)

Step 1 – Inbound Message

Client sends a message to the Task Router:

```
{
  "messageId": "12345",
  "channel": "email",
  "to": "user@example.com",
  "body": "Hello World"
}
```

Step 2 – Task Router Processing

- Validates payload
- Checks duplicate using `messageId`

- Generates a unique `traceId`
- Logs “message received” event
- Publishes to appropriate RabbitMQ topic (e.g., `message.email`)

Step 3 – Delivery Service Execution

- Picks message from its queue
- Simulates sending (e.g., “Email sent successfully”)
- Stores result in SQLite/Postgres DB
- Emits log to Logging Service with same `traceId`

Step 4 – Logging Service → Elasticsearch

- Receives logs from all services
- Stores logs in Elasticsearch
- Operators can search by `traceId` in Kibana to view end-to-end flow.

---

5. Communication Pattern and Justification

Pattern: Asynchronous Messaging

Reasons:

1. **Loose Coupling** — Delivery Services work independently.
2. **Fault Tolerance** — If a delivery service is down, messages remain in the queue.
3. **Scalability** — Each service scales horizontally based on traffic.
4. **Non-blocking API** — Task Router responds instantly.
5. **Easy Retry Handling** — Failed messages can be republished or moved to a DLQ.

---

6. Retry

Retry Mechanism

- Delivery Service attempts retry on temporary errors.
- Maximum retry count (default: **3 attempts**)
- If all retries fail:
    - Message recorded as FAILED in DB
    - Log sent to Logging Service
    - Message optionally moved to dead-letter queue

---

7. Logging & Observability

Trace ID Propagation

- Task Router creates a `traceId` per request
- All logs from all services include:
    - `traceId`
    - `serviceName`
    - `step`
    - `timestamp`
    - `status`

Elasticsearch + Kibana

- Logs indexed in Elasticsearch
- Kibana dashboard allows:

    - Trace-by-trace debugging
    - Filtering by channel
    - Monitoring message flow

---

8. Deployment Overview

- Each service runs on a dedicated port or container
- Recommended stack:
    - Node.js / Express / NestJS
    - Elasticsearch
    - Mongodb
    - Docker Compose for orchestration

Example port mapping:

- Task Router: 3001
- Email Service: 3002
- SMS Service: 3003
- WhatsApp Service: 3004
- Logging Service: 3005
- Elasticsearch: 9200

---

9. Summary

This HLD outlines a scalable, decoupled, and observable communication routing system designed with best practices in microservices architecture. The system fulfills all requirements including:

- Message routing
- Retry handling
- Multi-channel delivery
- Centralized logging with traceability
- Independent deployment per service