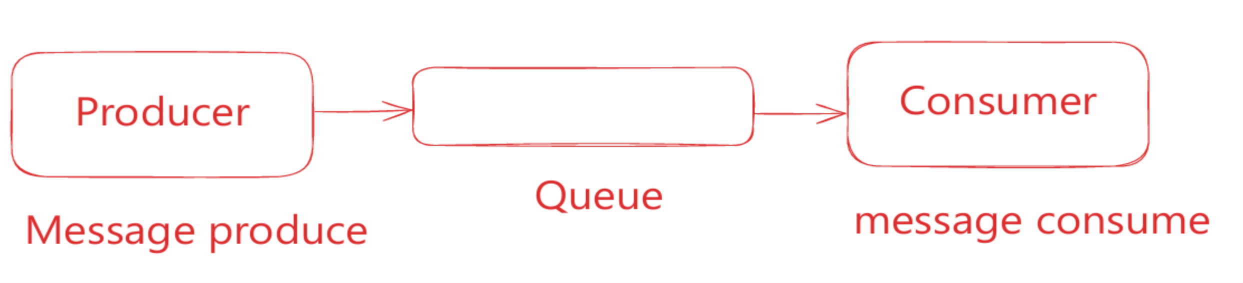
📨 Lecture 08 : MESSAGING QUEUES | KAFKA | RABBIT MQ (PART -1) 🌟

🔹 MESSAGING QUEUE FUNDAMENTALS

📌 Core Components:

- **Producer**: Message create karta hai ✍️
- **Messaging Queue**: Message store karta hai 🗂️
- **Consumer**: Message process karta hai 👥



🔸 Types of Messaging:

- **Pub/Sub (Publish/Subscribe)**
- **Queue (FIFO - First In First Out)**
- **Kafka** (Important - Not just a MQ)
- **RabbitMQ**

🔹 Basic Flow:

```
Producer → Message produce → Messaging Queue → Message consume → Consumer
```

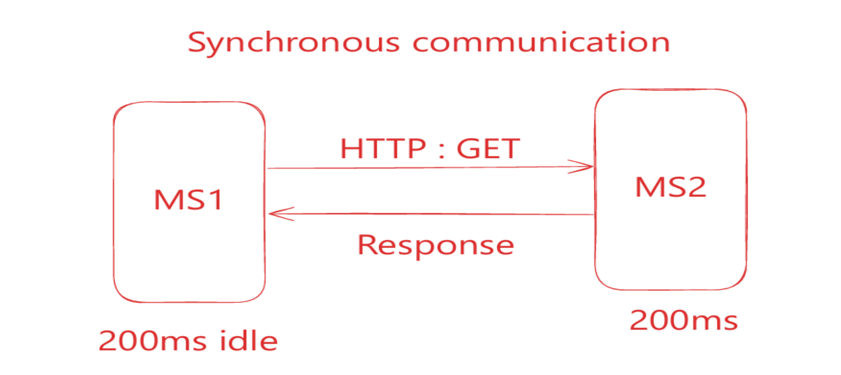
🎯 BENEFITS OF PRODUCER/CONSUMER PATTERN

📌 Key Advantages:

- ✅ Producer aur Consumer apni speed se kaam kar sakte hain
- ✅ Decoupling - Services independent hote hain
- ✅ Better resource utilization

🔄 SYNCHRONOUS vs ASYNCHRONOUS COMMUNICATION

🔹 Synchronous Communication:



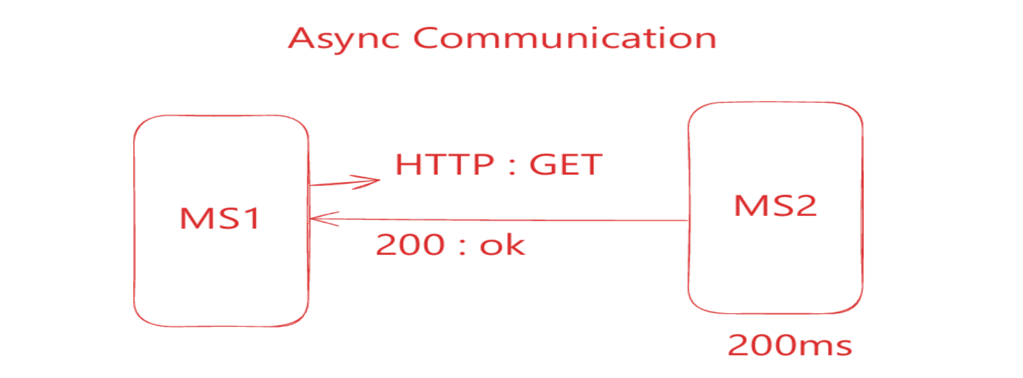
🕐 **Working:**

- Jab **MS1 (Microservice 1)** ne **MS2** ko ek request bheji (GET/POST call), tab **MS1 wait karta hai** ⏳ jab tak **MS2** apna kaam complete karke **response send nahi karta**.
- Is dauran, **MS1 kuch aur kaam nahi kar sakta** — woh idle state me rehta hai.
- **MS2** jab tak apna processing (jaise DB fetch, calculation, etc.) kar raha hota hai, **MS1** ko uska result milne tak rukna hi padta hai.

📌 **In short:**

> “Synchronous means — request bhejna aur response milne tak rukna.” 🚦

🔸 Asynchronous Communication ⚡



🧠 **Working Explanation:**

- Jab **MS1** ne **MS2** ko request bheji, tab **MS2 turant ek “Acknowledgment (200: OK)”** response bhej deta hai ki — “Request mil gayi hai, main process kar lunga.” ✅
- **MS1** ko ab **wait karne ki zarurat nahi hoti**, woh apna **next kaam continue** kar sakta hai 🚀
- **MS2** background me apna task (DB update, processing, etc.) independently complete karta hai.

💡 **Why Asynchronous?**
Because system **non-blocking** hai —
👉 **MS1 ko MS2 ke complete hone ka wait nahi karna padta.**
👉 **Throughput badhta hai** (more requests handled in less time).

📨 **Messaging Queue ka Role:**
Messaging Queues (like **Kafka**, **RabbitMQ**) iss problem ka **real-world solution** hai.

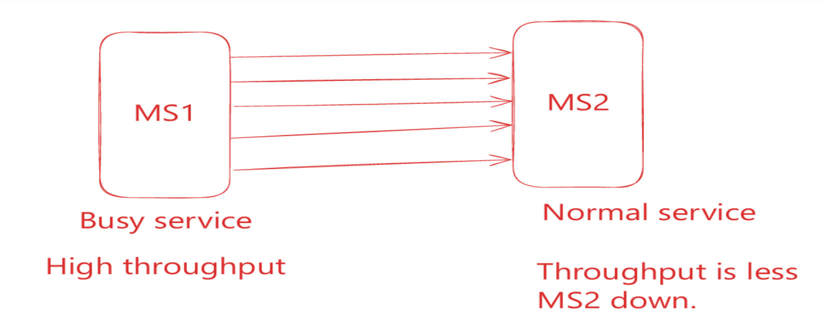
- Ye ensure karti hai ki agar MS2 busy hai 🔄,
  to bhi request **queue me store ho jaaye**.

- **MS1 free ho jaata hai**, aur **MS2** jab ready hota hai tab us request ko process karta hai.

🧩 **In short:**

> "Asynchronous Communication = No waiting ⏩ + High performance ⚙️ + Reliable message handling 📨"

📊 SERVICE STATE COMPARISON



Component	State	Throughput
🧩 **MS1**	Busy service	High throughput (zyada requests bhej raha hai)
⚙️ **MS2**	Normal service	Throughput kam hai (limited requests handle kar sakta hai)

⚠️ Throughput Imbalance Problem

Agar **MS1** bahut saare messages ek saath **MS2** ko bhejta hai**,
to question uthta hai 👇

> “Kya MS2 itne saare messages handle kar paayega?” 🤔

❌ **Nahi, hamesha nahi!**
Agar **MS2 ka throughput (capacity)** kam hai,
to woh overload hone lagta hai → **server slow ya crash** bhi ho sakta hai 💥

💡 HLD (High-Level Design) Concept: Throughput

📌 **Throughput (OPS - Operations Per Second)**
ka matlab hota hai —

> “Koi bhi service ya server ek second me kitne operations handle kar sakta hai.” ⚙️

- Example:
  Agar ek server ka throughput = **1000 OPS**,
  aur usse **1200 requests/sec** milti hain,
  to **200 requests fail ya drop** ho jayengi 🚨

- Simple words me —
  Throughput = Handling Capacity per second

🧠 Key Takeaway:

> Jab load (requests/sec) > throughput ho jaata hai →
> **System crash ya service down ho sakti hai.** ⚠️

🚗 REAL-WORLD EXAMPLE: OLA / UBER / RAPIDO

🔹 Use Case: Real-time Driver Tracking

- Har driver apni **GPS location** server ko bhejta hai.
- **Frequency:** 1 signal per second (1 req/sec per driver) 📍

🔸 Back of the Envelope Calculation

- **Delhi:** 1 Lakh drivers
- ⇒ **1 Lakh requests/second** = 100,000 req/sec 🚀

Har request me:

- Driver ka **current location update**
- **Fare calculation**
- **Ride completion time tracking**
- **Weather condition (rain surge etc.)**

⚙️ Database Challenge

Traditional workflow 👇
Client → Server → SQL/NoSQL (Cassandra, MongoDB)

Problem:

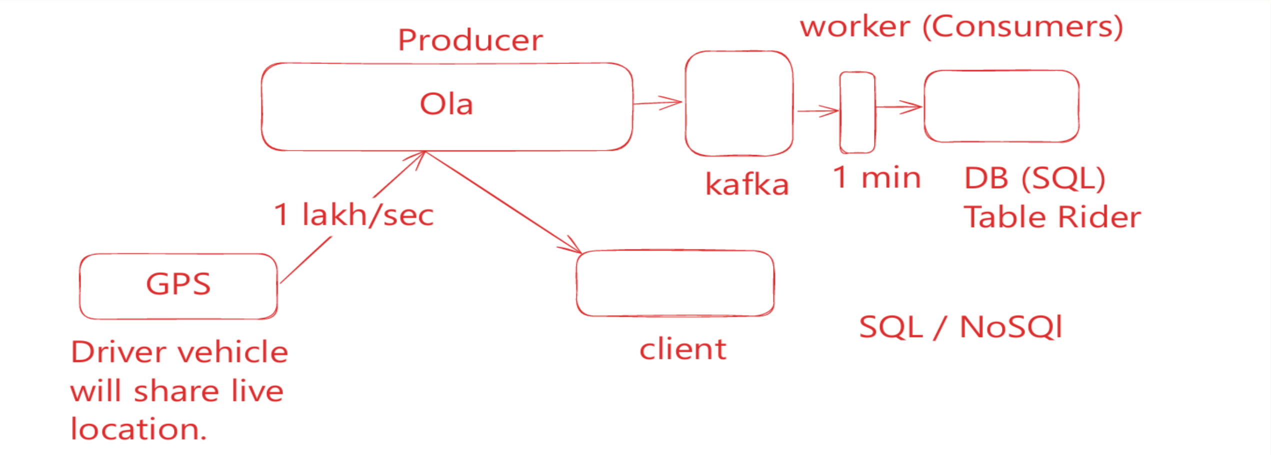
- 1 DB par **itna heavy load (1 lakh req/sec)** aa gaya ❌
- **SQL/NoSQL database** ka throughput limited hota hai
- Itna load lene par DB **slow ya crash** ho jaata hai ⚠️

💡 Solution: Messaging Queue **(Kafka)**

✅ **Why Kafka?**

- Kafka ka **throughput bahut high** hota hai (millions req/sec).
- It acts as a **buffer** between server & database.

⚙️ How Kafka Solves the Problem



`Client (Driver App) → OLA Server (Producer) → Kafka Queue (Message Broker) → Worker/Consumer → Database (Cassandra/MongoDB)`

🔸 **Working:**

1. **Server (Producer):**
   - Drivers se request receive karta hai.
   - Sabhi requests **Kafka server** me push kar deta hai.
2. **Kafka (Buffer Queue):**
   - Crores of requests ko temporarily store karta hai.
   - High throughput maintain karta hai ⚡
3. **Workers (Consumers):**
   - Kafka se messages **batch me** uthate hain (e.g., every 10–15 min).
   - **Bulk upload** karte hain database me efficiently.
4. **Database:**
   - Ab par request-by-request load nahi,
     balki **batch-wise processing** hoti hai → system stable 💪

🧠 Job Roles in System

Component	Role
🌐 **Web Server**	Takes request, processes it, sends data to Kafka
📦 **Kafka (Messaging Queue)**	Buffers requests, ensures no data loss
👷 **Worker / Consumer**	Pulls messages from Kafka, uploads in bulk to DB
🗄️ **Database**	Stores final processed data (comparatively slower)

🔥 Summary

🧩 *Without Kafka → DB overload, system crash*
⚡ *With Kafka → High throughput, smooth async communication, scalable system*

💡 WHY KAFKA? KEY BENEFITS

🔹 Web Server Limitations:

- Server processes request → DB slower → Latency higher

🔸 Kafka Advantages:

1. **Pace Matching**

```
Producer ────────── Kafka ────────── Consumer

┌────────────┐      ┌────────────────┐      ┌────────────┐
│ PRODUCER │ ───▶ │   KAFKA 📨    │ ───▶    │ CONSUMER │
└────────────┘      └────────────────┘      └────────────┘

        🔹 Producer → High Throughput 🚀
        🔹 Consumer → Normal Throughput ⚙️
        🔹 Kafka → Balances speed mismatch (Pace Matching) ⚖️
```

2. **Async Communication (Notification System)**

```
⚡ Async Communication (Notification System)

📤 Producer ───▶ 🗄️ Kafka ───▶ 📥 Consumer
                                │
                                ▼
🔔 Notifications (APN | FCM | MailChimp)
```

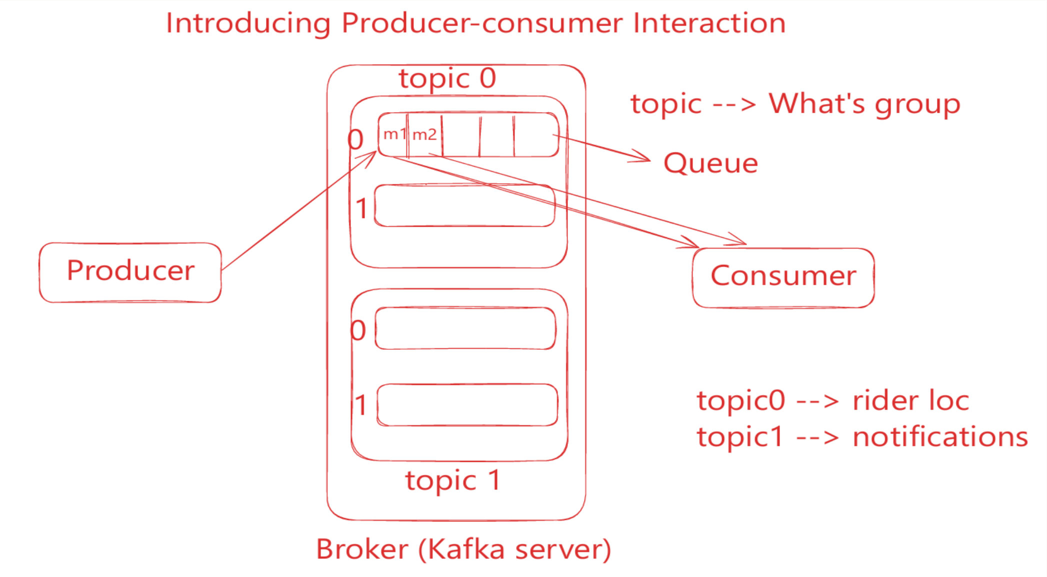
Explanation:

- 📤 Producer messages bhejta hai → 🗄️ Kafka me queue hoti hai → 📥 Consumer asynchronously fetch karta hai.
- 🔔 Notifications automatically APN, FCM, ya MailChimp ke through deliver hoti hain.
- ⚡ Producer ko wait nahi karna padta, kaam fast aur non-blocking hota hai.

🏗️ KAFKA IN-DEPTH ARCHITECTURE

📌 Core Components:

- **Cluster** - Multiple Kafka servers
- **Broker** - Individual Kafka server
- **Topic** - Category for messages
- **Partition** - Topic subdivision
- **Offset** - Message position in partition
- **Zookeeper / Kraft** - Coordination services



🗄️ Kafka Server = Broker

- Kafka me har server ko **Broker** kehte hain.

- 🏷️ **Topic**: Category of messages

  - Example:

    - `topic0` → Rider locations 🚖
    - `topic1` → Notifications 🔔

- 📦 **Partition**: Topic ke andar queue ki tarah, jahan messages sequentially store hote hain
- 🔢 **Offset**: Partition ke andar message ka index

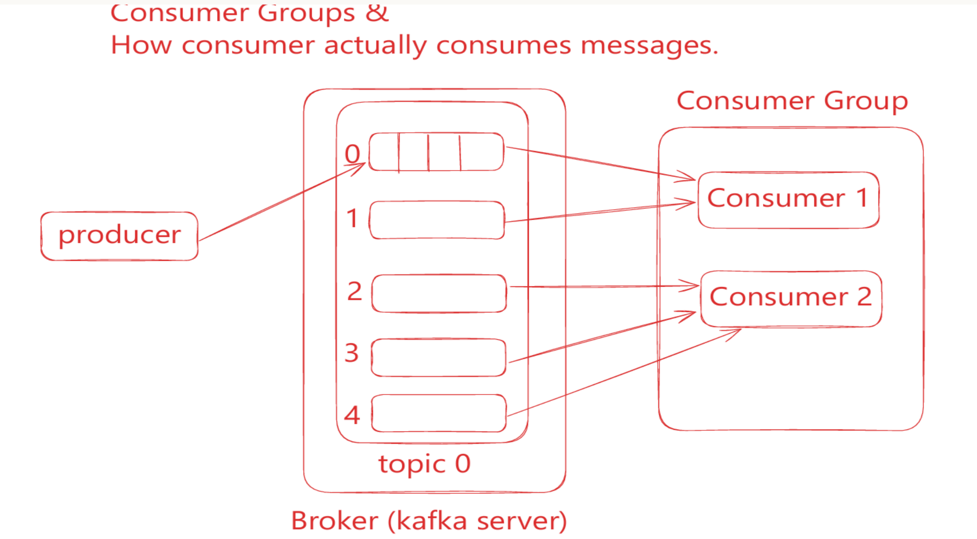
  - Example:

    - `m1` → offset 0
    - `m2` → offset 1

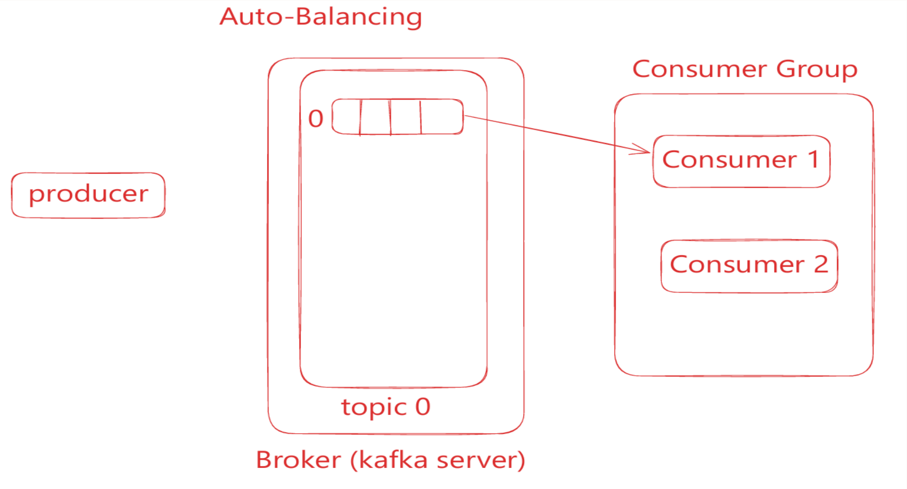
- 💡 Offset se Kafka easily track karta hai ki kaunsa message consume hua aur kaunsa pending hai.

👥 CONSUMER GROUPS & LOAD BALANCING | HOW CONSUMER ACTUALLY CONSUME THE MESSAGE

🔹 Multiple Consumers:



🔸 Auto-Balancing Mechanism:



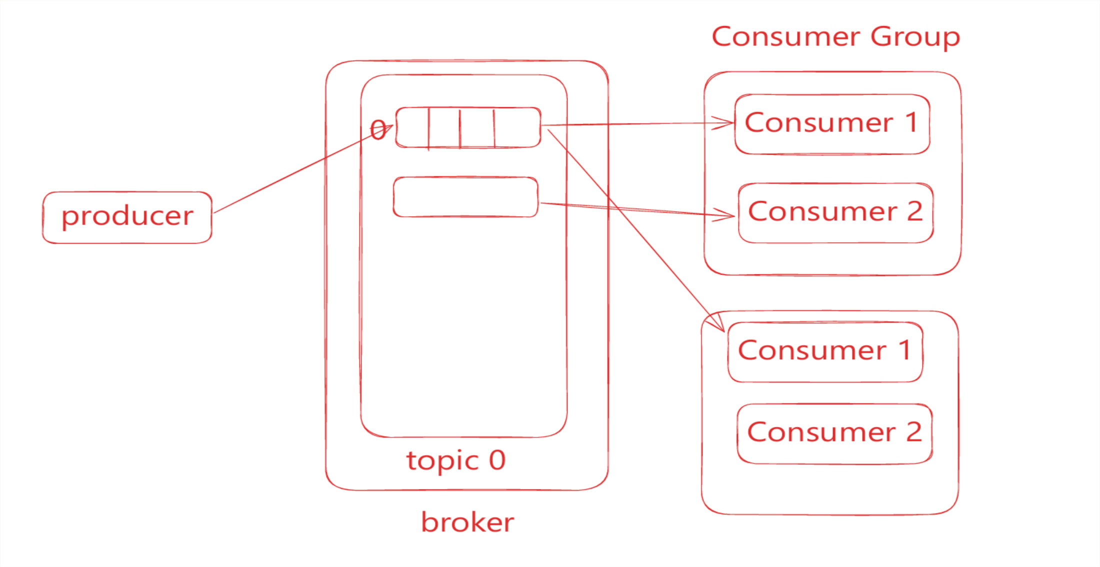
- Partitions auto-assigned to consumers

🔸 **Auto-Balancing Mechanism** ⚖️
Partitions auto-assigned to consumers 🔄

Content 📝:
Partition ko kaun sunega? Consumer1️⃣, Consumer2️⃣... bola mujhe bhi kisi queue ko listen karna hai, broker bolega ❌. Broker ek queue ka access2️⃣consumers ko nahi dega.

Rules 📌:
1️⃣If number of partitions < number of consumers → some consumers will remain idle ⏸️
2️⃣consumer can consume multiple partitions 🔄
3️⃣ partition is consumed by only1️⃣consumer 🔒

🎯 Introducing Multiple Consumer Groups :



Producer → Consumers

- Ek **partition** ek **consumer** padh sakta hai, par **same consumer group** me 🎧
- **Different consumer groups** me alag-alag consumer ek **partition** ko listen kar sakta hai ✅

Why Kafka does this?

- Consumer group concept **completely new** in Kafka
- Messaging queue serve karta hai **2 purposes**:
   1. **Queue (FIFO)** → 1 to 1 mapping 🛒
   2. **Pub/Sub model** → 1 to many 📢

🔹 Messaging Queue Models

1. Normal Queue (1 to 1)

- **Flow:** `Producer → QUEUE → Consumer`
- Normal queue me ek producer par 2 consumer allow nahi ❌
- Example: OLA → UPI signal → sirf ek consumer

🔹 Queue Use Case (Single Consumer)

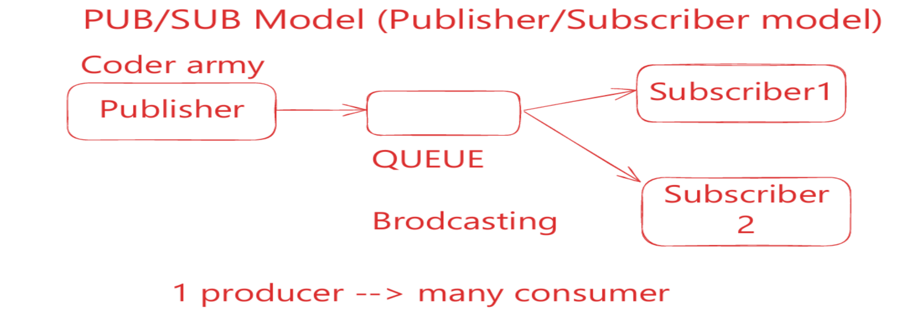
- **Goal:** Message produce ho → **ek hi consumer** consume kare ✅

  Example: OLA Rider GPS signal 🚖 → Sirf ride book karne wale driver ka app consume kare

- Queue ensures **1 message = 1 consumer**
- Useful for **rides, transactions, single-user events** ⚡

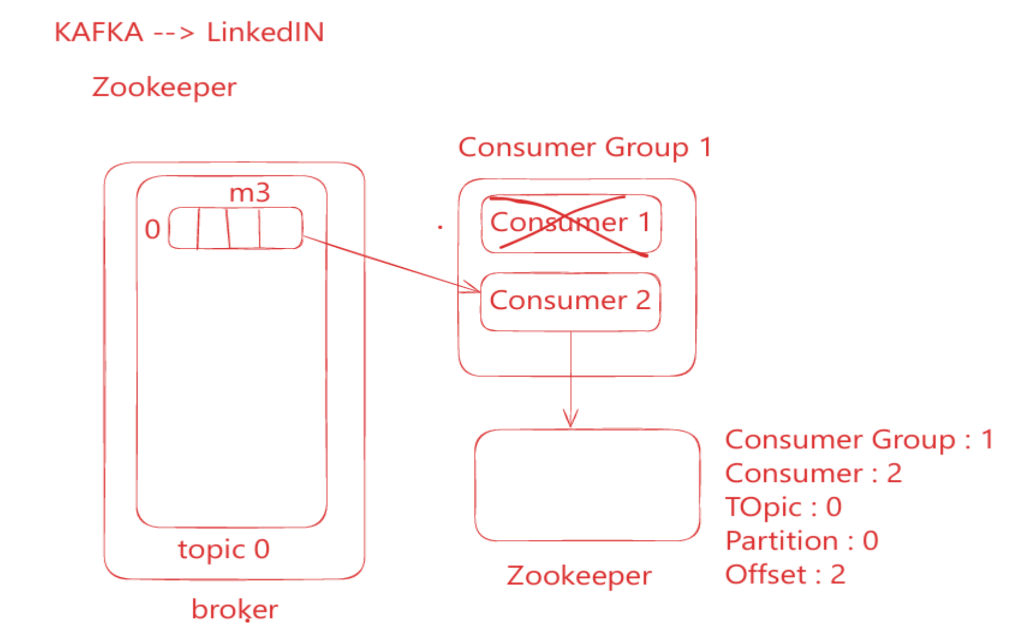
2. Pub/Sub Model (1 to Many)

- **Flow:**


PUB/SUB Model (Publisher/Subscriber model)
Coder army
Publisher
QUEUE
Brodcasting
Subscriber1
Subscriber 2
1 producer --> many consumer

- **Broadcasting:** 1 producer → many consumers 💬
- Use Case: Jab user channel par video upload kare → saare subscribers ko notification jaye ✅

🔹 Zookeeper Role in Kafka 🐘



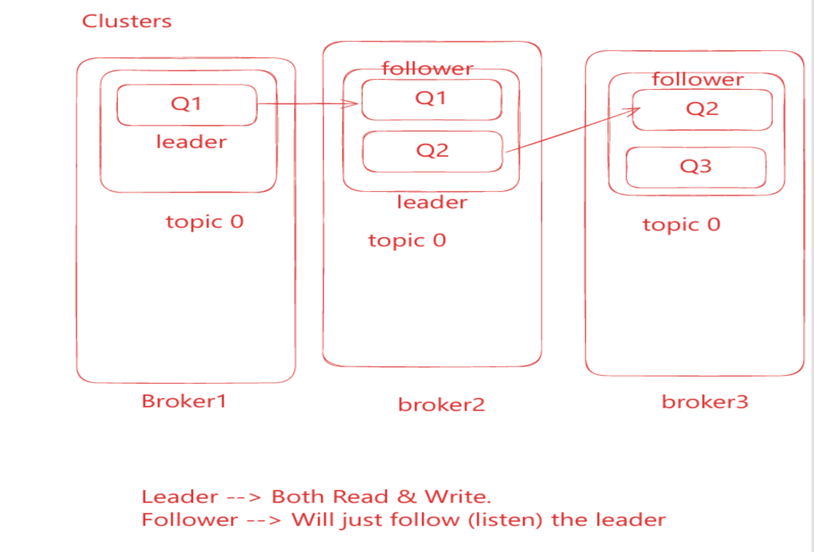
- Zookeeper Kafka ne nahi banaya, ye **independently partition ka kaam** karta hai
- Kafka Zookeeper se group info leta hai
- **Partition read ke baad offset update hota hai**

🔹 Zookeeper ka Role

- Ye saari info **store karta hai**
- Agar **consumer1 down** ho gaya:

- ○ Broker bolega → consumer1 ab partition listen nahi kar sakta
- ○ Partition ka kaam **consumer2** karega
- ○ Zookeeper offset track karega → **offset 2 se read start**

🏢 KAFKA CLUSTER ARCHITECTURE



- **Leader** → Both Read & Write ✅
- **Follower** → Follow/Listen leader, Read only ⚡

🔹 High Availability & Fault Tolerance

- Kafka ki service **multiple brokers pe deploy** hoti hai
- **Ek broker crash** → Data lost nahi hota
- **Data replication** → Q1 info broker2 me, Q2/Q3 alag broker me
- Agar **leader down** → koi **follower leader ban jaata hai**

💡 Key Points

- Consumer group ka **1 partition** → **1 consumer** rule
- Multiple consumers **auto-balance** across partitions
- Zookeeper offset track karta hai → consumer down/up → reading continue
- Kafka ensures **high availability & fault tolerance**
- Leader-follower architecture ensures **load balancing & replication**

📚 COMPLETE SUMMARY 🌟

🎯 Key Messaging Patterns:

- **Queue (FIFO):** 1 producer → 1 consumer
- **Pub/Sub:** 1 producer → multiple consumers
- **Kafka:** High-throughput distributed messaging

🚀 Kafka Advantages:

- High throughput (100k+ msg/sec) ⚡
- Pace matching for fast producers & slow consumers
- Async communication
- Fault tolerance via replication

🏗️ Kafka Architecture Components:

- **Broker, Topic, Partition, Consumer Group, Offset**

🔧 Important Rules:

- 1 partition = 1 consumer
- Multiple partitions can be consumed by 1 consumer
- Auto-balancing
- Leader-follower for high availability

💡 Real-World Applications:

- **Ola/Uber:** GPS tracking
- **LinkedIn:** High-volume feeds
- **Notification systems**
- **Data pipelines**

📊 Performance Numbers:

- **Throughput:** 100,000+ messages/sec
- **Scalability:** Horizontal with multiple brokers
- **Latency:** Low for real-time