Distributed Message Queue

Modern architecture -> system broken into small component with interface between them
Message queue provides communication between them.

Advantages of message queue
Decoupling
Improve scalability - eg increase consumer instance during peak hour
improve availability
better performance - producer don't have to wait for consumer response

Popular message queue
event streaming platform - apache kafkaesque, apache pulsar
message queue - apache rocketMQ, rabbitMQ, activeMQ, ZeroMQ

NOTE - RabbitMQ allow repeated message consumption and long message retention, implementation uses append-only log.

Design distributed message queue with additional features eg long retention, repeated message consumption.

Requirements (functional and non functional)

1. Producer send message and consumer consume message
2. Other consideration - performance, message delivery semantics, data detention etc
3. Questions:
    1. Data format and average message? Text only or include multimedia - and: text only, eg range KBs
    2. Message be repeated consume - yes, by different consumer. NOTE: traditional distributed message queue not retain message once.
    3. Message consumed in same order - yes
    4. Data retention - 2 weeks (added feature)
    5. How many producer and consumer - the more the better
    6. Data delivery semantic (at most once, at least once, exactly once) - at-least once
    7. Target throughput and end-to-end latency - support high throughput and low latency
4. **Functional requirements:**
    1. Producer send message to message queue
    2. Consumer consume message from message queue
    3. Message can be consumed repeatedly or only once
    4. Historical data can be trusted
    5. Message size = kb range
    6. Message order same as delivery

    7. Data delivery semantics (at least once, almost once, exactly once)
5. **Nonfunctional requirements**
    1. High throughput or low latency
    2. Scalable - support sudden surge
    3. Persisted and durable - persisted on disk and replicated across multiple nodes.
6. NOTE:
    1. Traditional message store msg in in memory till the time they consumed
    2. Traditional message not maintain message order

STEP 2: HLD and get buy in

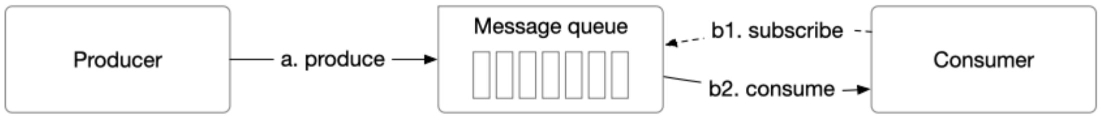

Figure 2 Key components in a message queue

Here, both producer and consumer are client in client/server model, message = queue in the server. Client and server communicate over the network.

Message models

Point-to-point
message is sent to queue and consumed by one and only one consumer, though there can be multiple consumer waiting to consume message.
NOTE: no data retention here, but we need to include persistence layer to keep message for 2 layer.

publish-subscribe
message sent to a topic and received by consumer subscribe to that topic.

Topic, partition and brokers
Problem - when data volume is too large for a single server
Ans: Divide topic into partitions. Server that hold partition = brokers. Can scale topic capacity by expanding partition number.
Topic partition operate in FIFO form, position of message in partition called an offset.

producer send message with optional message key (eg userid). All message with same message key will be send to same partition. If key absent - message send to random partition.

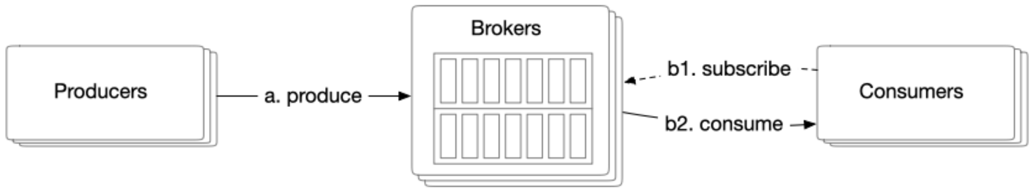

Figure 6 Message queue cluster

Consumer group
support both point-to-point and subscribe-publish model.
consumer can be subscribed to multiple topics and maintain its own offset.

NOTE: parallel data reading improve throughput, but message order consumption can't be guaranteed.
Solution: constraint - 1 partition consumed by 1 consumer in same group.. So, now if we place consumer in same group - message consumed by only one consumer = point to point model.

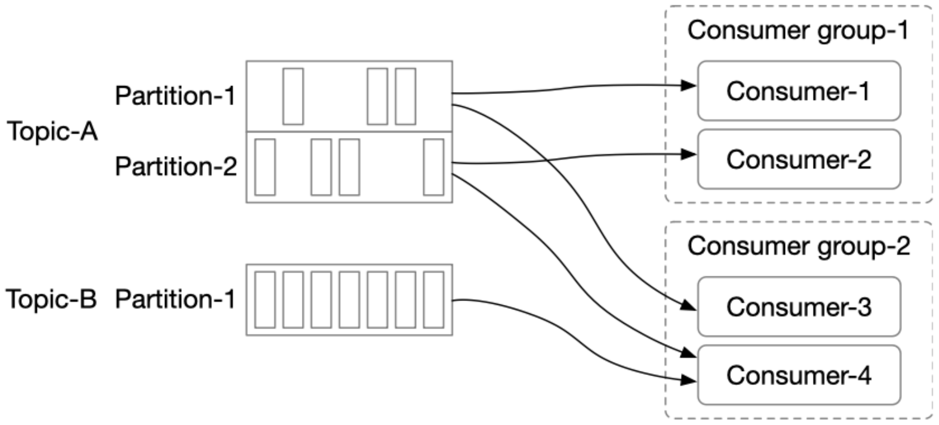

Figure 7 Consumer groups

High level architecture

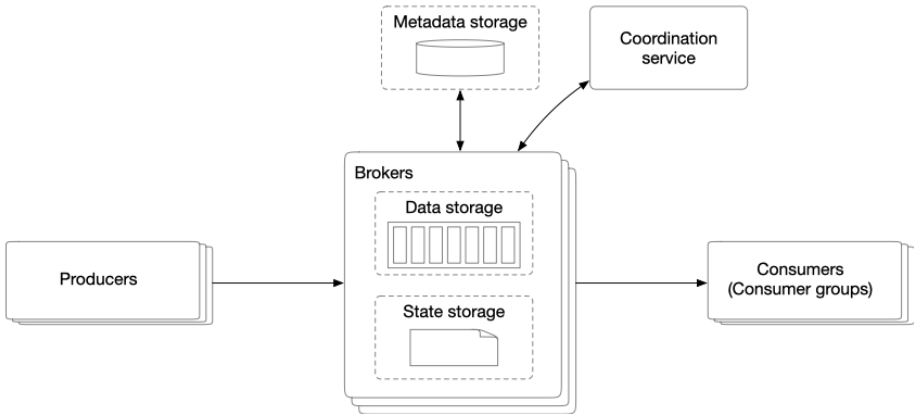

Figure 8 High-level design

broker - hold multiple partition (topic subset for message)
data storage = message persist in partition
state storage = store consumer state
metadata storage = topic configuration and properties
service discovery = which brokers are alive
leader election elect one active controller in cluster, to assign partitions
Apache zookeeper or etc are commonly used to elect controller.

Step 3 - design deep dive

3 important design choice
1. On-disk structure - great sequential access performance of rotational disks and aggressive disk caching strategy of modern operating system.
2. Message data structure - message pass from producer - queue - consumer, with no modification (minimize copy, and save cost)
3. Favor batching - small IO enemy of high throughput - so send message in batch whenever possible

Data storage
1. Database -
relational db - create topic table and write message
nosql db - create collection as topic and write message as document
NOT ideal - not support write heavy and read heavy in large scale

2. Write ahead log (WAL) - plain file where new entries are appended to append-only log. Eg use case = redo log in MySQL and WAL in Zookeeper
recommend to use because advantage - pure sequential read/write access pattern. Disk performance of sequential access is very good. Also, rotational logs have large capacity.

to manage capacity - divide log file into segments. New message append only to active segments, when it reaches certain size, new active segment create to receive new message, and current segment becomes inactive. Non active segment only serve read request, old non-active segment files can be truncated if they exceed retention or capacity limit.
folder name “Partition-{:partition_id}”

NOTE on disk performance
NOTE: rotational disk are slow only for random access. For sequential, modern disk drive in RAID configuration can achieve easily several 100 mb/sec of read and write speed.
NOTE: WAL also adv for heavy OD disk caching

Message data structure:
sample schema = field name (key, value, topic and partition, offset, timestamp, size, crc[5])
message key - can be string or number, need not be unique. If key not defined, partition randomly chosen, else partition = hash(key)%numPartitions.
message value - message payload, can be plain text or compressed binary
other fields - topic, partition, offset, timestamp, size, CRC (cyclic redundancy check)

batching - producer send data in batches
1. Allow OS to group and send message in single network.
2. Broker write message to append log in large chunk - seq write large block, and contiguous block of disk cache
NOTE: there will be tradeoff between throughput and latency

Producer flow
producer send message to route layer - route to correct broker. If broker are replicated, “correct broker” = leader replica. Below is the flow
1. Producer send message to routing layer
2. Routing layer read replica distribution plan from metadata storage and cache it locally. When message arrive, route message to leader replica of partition 1, stored in broker 1.
3. Leader replica receive message and follower replica pull data from leader (**fault tolerance)**
4. When enough replica have synchronized message, leader commit data (disk persisted) - data ready to consume

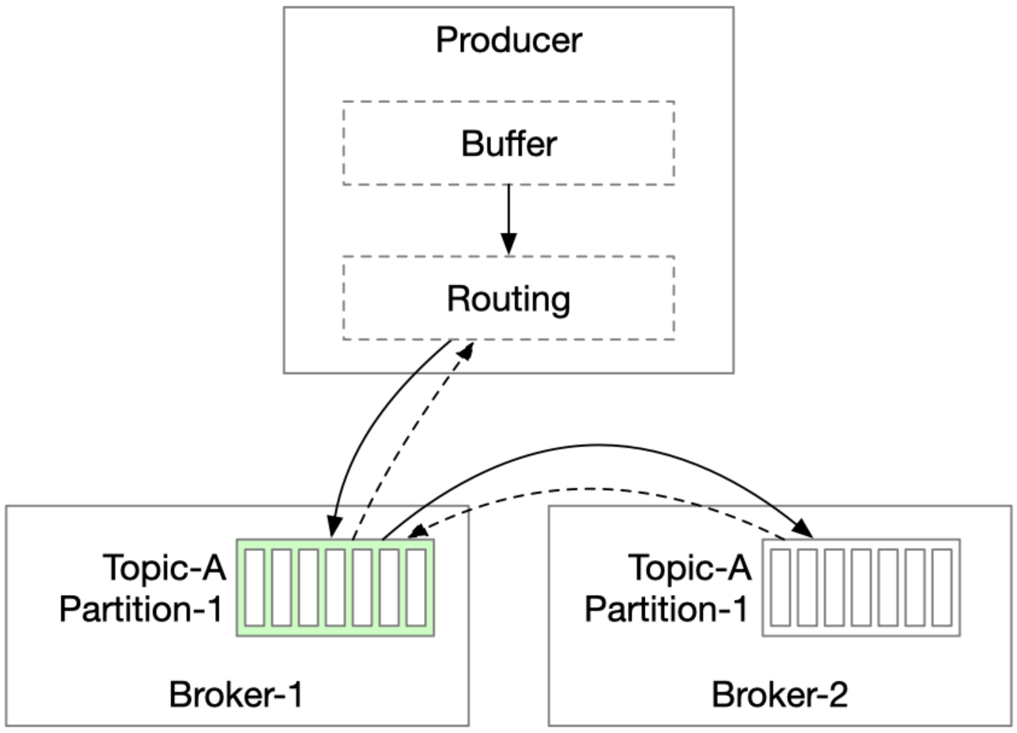

Figure 12 Producer with buffer and routing

routing layer wrapped into producer and buffer component added to producer -> few network hops, producer have own logic to determine partition, batching increase throughput.

NOTE: tradeoff of batch size -> large = increase throughput but high latency, else throughput suffer.

Consumer flow

Push model	**Pull model (popular choice)**
Pros: low latency - broker can push message immediately.	Consumer control consumption rate. If consumption rate < production rate -> scale out.
-	More suitable for batch processing
Cons: consumer overwhelmed when consumption rate < production rate.	Cons: when no message in broker, consumer might still keep pulling data, wasting resource
cons: difficult to deal with consumers with diverse processing power.	

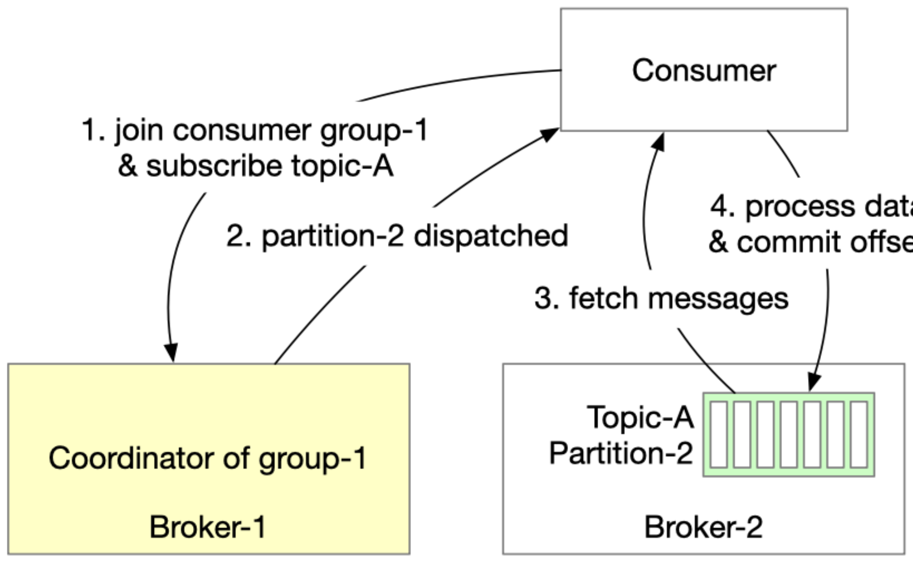

Figure 15 Pull model

1. New consumer want to join new group and subscribe to new topic. Determine broker node by hashing group name.
2. Coordinator confirm that consumer has joined group and assign partition 2 to consumer using different strategy (eg round robin)
3. Consumer fetch message from last consumed offset (managed by state storage)
4. Consumer process message and commit offset to broker

Consumer rebalancing

- decide which consumer is responsible for which partition.
- happen when consumer join, leave or partition adjusted.
- coordinator receive heartbeat of consumer and manage their offset on partition. It maintain joined consumer list, when list change - coordinator elect new leader of group - generate new partition dispatch plan - broadcast plan to consumer

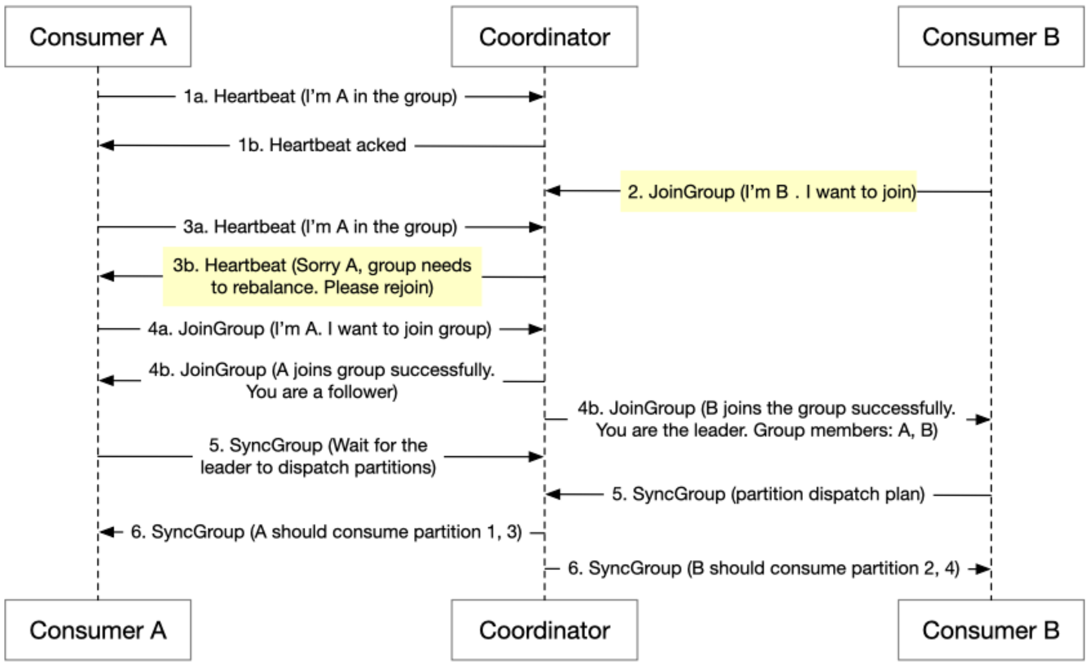
Consumer A
Coordinator
Consumer B
1a. Heartbeat (I'm A in the group)
1b. Heartbeat acked
2. JoinGroup (I'm B . I want to join)
3a. Heartbeat (I'm A in the group)
3b. Heartbeat (Sorry A, group needs to rebalance. Please rejoin)
4a. JoinGroup (I'm A. I want to join group)
4b. JoinGroup (A joins group successfully. You are a follower)
4b. JoinGroup (B joins the group successfully. You are the leader. Group members: A, B)
5. SyncGroup (Wait for the leader to dispatch partitions)
5. SyncGroup (partition dispatch plan)
6. SyncGroup (A should consume partition 1, 3)
6. SyncGroup (B should consume partition 2, 4)
Consumer A
Coordinator
Consumer B

Figure 18 New consumer joins

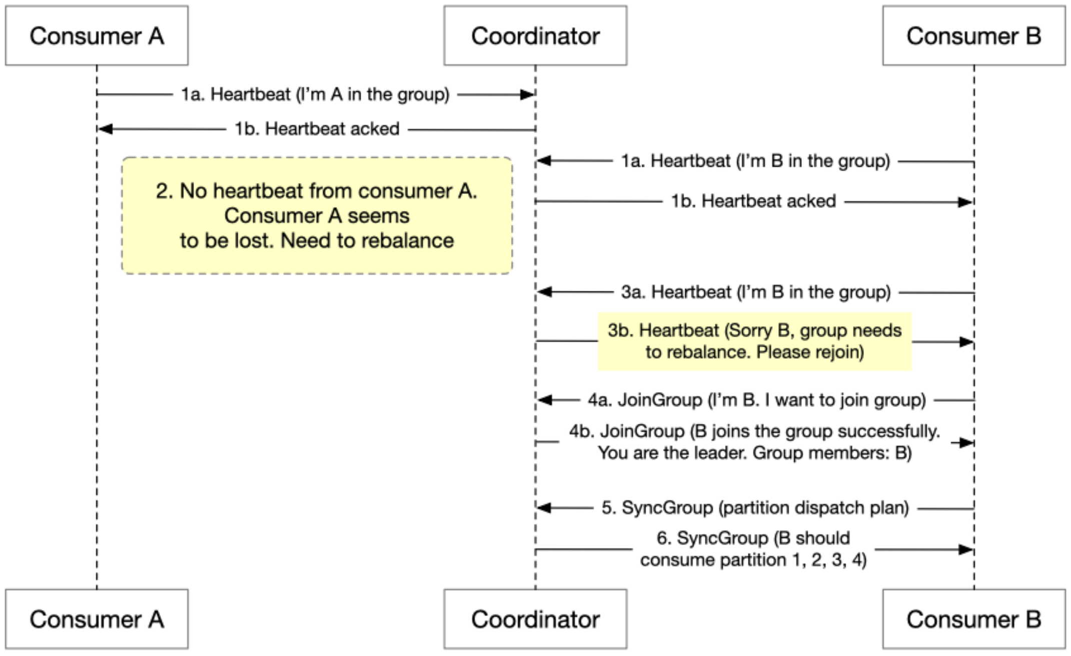
Consumer A
Coordinator
Consumer B
1a. Heartbeat (I'm A in the group)
1b. Heartbeat acked
1a. Heartbeat (I'm B in the group)
2. No heartbeat from consumer A. Consumer A seems to be lost. Need to rebalance
1b. Heartbeat acked
3a. Heartbeat (I'm B in the group)
3b. Heartbeat (Sorry B, group needs to rebalance. Please rejoin)
4a. JoinGroup (I'm B. I want to join group)
4b. JoinGroup (B joins the group successfully. You are the leader. Group members: B)
5. SyncGroup (partition dispatch plan)
6. SyncGroup (B should consume partition 1, 2, 3, 4)
Consumer A
Coordinator
Consumer B

Figure 20 Existing consumer crashes

State storage

stores mapping between partition and consumer eg If consumer in group 1 consume message from partition in sequence and commit the consumed offset 6 - message before and at offset 6 are already consumed - in case of crash other consumer in same group will resume consumption by reading last consumed offset from state storage
requirement for data access pattern - frequent read/write operation, data update frequently but delete rarely, random read/write, consistency is important
solution - apache KV store like zookeeper, note kafkaesque has moved offset storage from zookeeper to kafkaesque brokers.

metadata storage
stores topic config and properties, including partition number, retention period, and replica distribution.

zookeeper (solution)
- offer hierarchical key-value store
- provide distributed config service, synchronization service and naming registry
- metadata and state storage move to zookeeper
- broker only need to maintain data storage for messages
- help with leader election of broker cluster

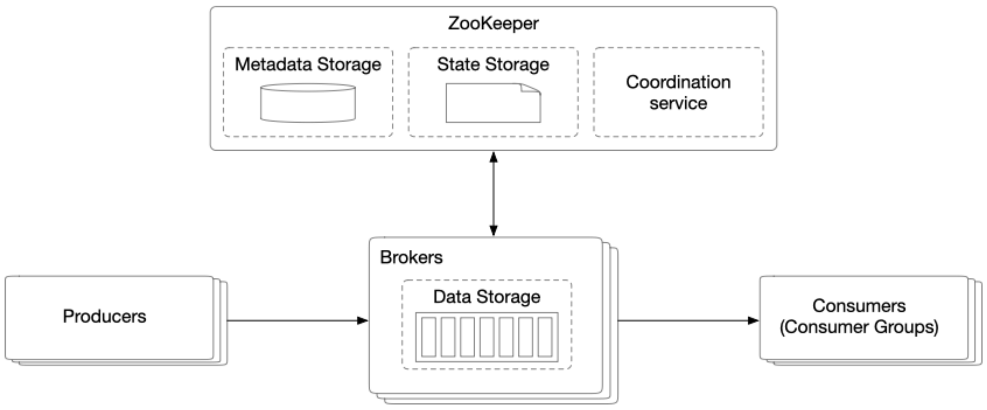

Figure 22 Zookeeper

Replication - to overcome hardware issues (fault tolerance)
producer send message to leader replica - following replica pull new message - once enough replicas are synchronized - acknowledge producer.

with the help of coordination service - broker nodes make replica distribution plan and persist it in metadata storage
keep in-sync replicas with leader

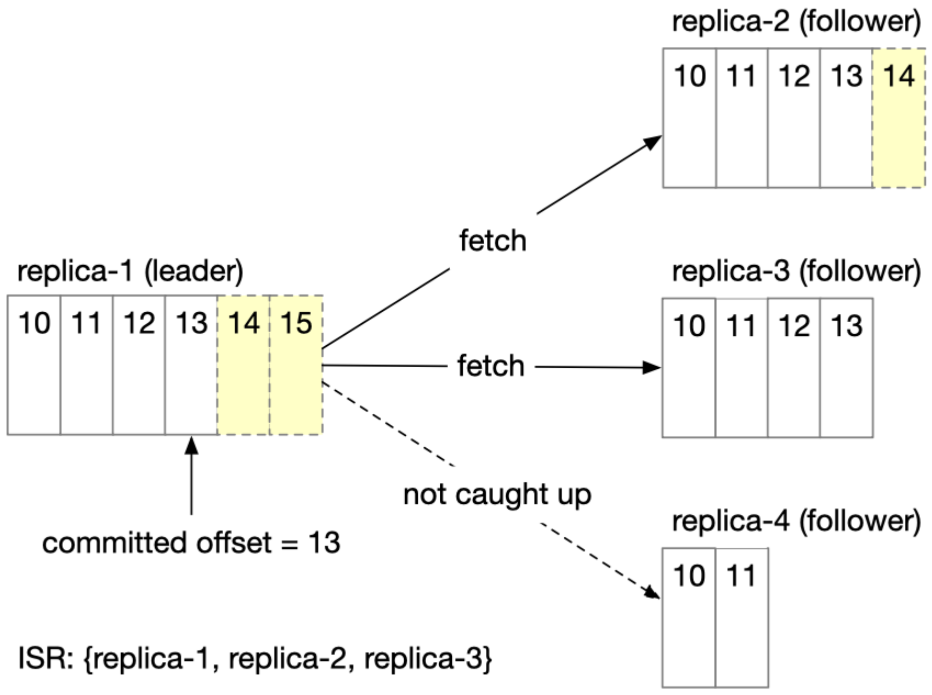

Figure 24 How ISR works

acknowledgement setting
ACK = all -> the producer get CAK when all ISR have receive message (high latency)
ACK=1 -> producer receives ACK once leader persist message (low latency but might have occasional data loss)
ACK=0 -> producer keep sending message to leader without waiting for any acknowledgement

consumer side - easiest way is to read from leader replica, however not always the best option since sometimes, the consumer is located in different data center from leader replica, read performance suffer (so worthwhile to enable read from closest ISR)

Scalability
1. Producer: much simpler than consumer. Not need group. Scaled by adding/removing producer instances.
2. Consumer: Consumer groups are isolated from each other, so easy to add/remove consumer group. Inside group : rebalance to add/remove consumer to support scalability and fault tolerance
3. Broker: distribute replicas when broker nodes are added/removed.
4. Partition: we change partition for various reasons like topic scaling, throughput tuning, availability/throughput balance -> trigger consumer rebalancing

add new partition -> persisted message are still in old partition, new message start persist in new partition
remove partition -> can't remove partition immediately, first migrate data to remaining partition during its retention period, producer start sending new message to remaining partition - after retention period expire, consumer group need rebalancing.

Message filtering
use tag to send message to subscribed topic to filter the message
- complex logic need broker grammar parser or script executor - too heavyweight for messagee queue

Delayed or schedules message
- schedule message - to delay message delivery (eg order closed if not paid within 30 mins after creation) : send delayed verification message after scheduled time - consumer receive message, check status, if payment not complete - message will be ignored

- delay message - to temporary storage on broker side instead to topics immediately (can be dedicated delay queue with predefined delay level, hierarchial time wheel)