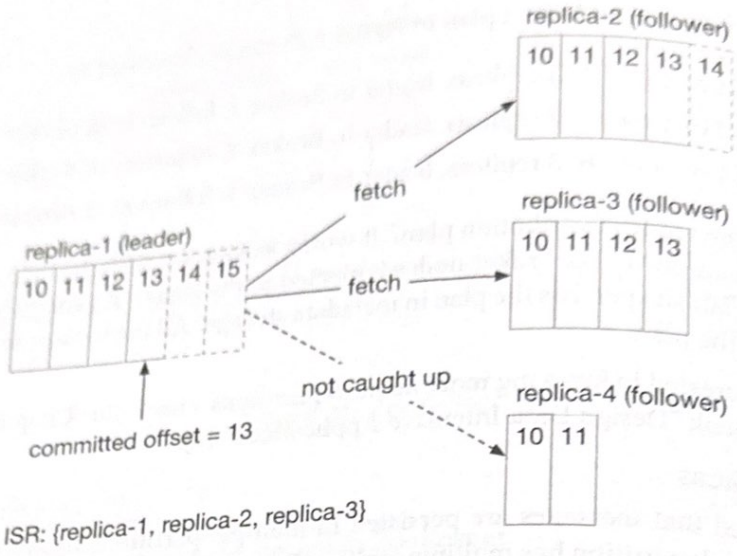


Figure 4.24: How ISR works

do we need ISR? The reason is that ISR reflects the trade-off between performance durability. If producers don't want to lose any messages, the safest way to do that is o ensure all replicas are already in sync before sending an acknowledgment. But a slow replica will cause the whole partition to become slow or unavailable.

Now that we've discussed ISR, let's take a look at acknowledgment settings. Producers can choose to receive acknowledgments until the k number of ISRs has received the message, where k is configurable.

ACK=all

Figure 4.25 illustrates the case with ACK=all. With ACK=all, the producer gets an ACK when all ISRs have received the message. This means it takes a longer time to send a message because we need to wait for the slowest ISR, but it gives the strongest message durability.

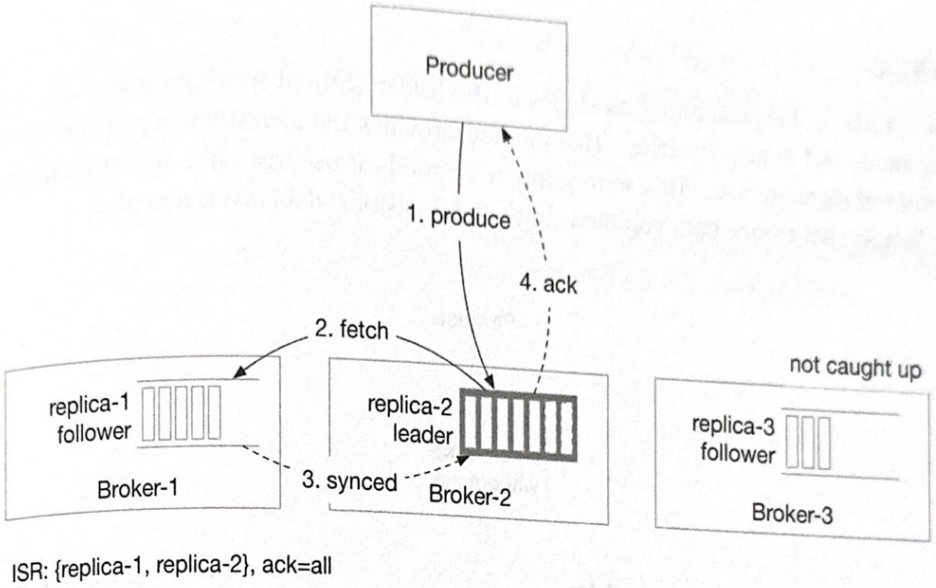


Figure 4.25: ACK=all

ACK=1

With ACK=1, the producer receives an ACK once the leader persists the message. The latency is improved by not waiting for data synchronization. If the leader fails immediately after a message is acknowledged but before it is replicated by follower nodes, then the message is lost. This setting is suitable for low latency systems where occasional data loss is acceptable.

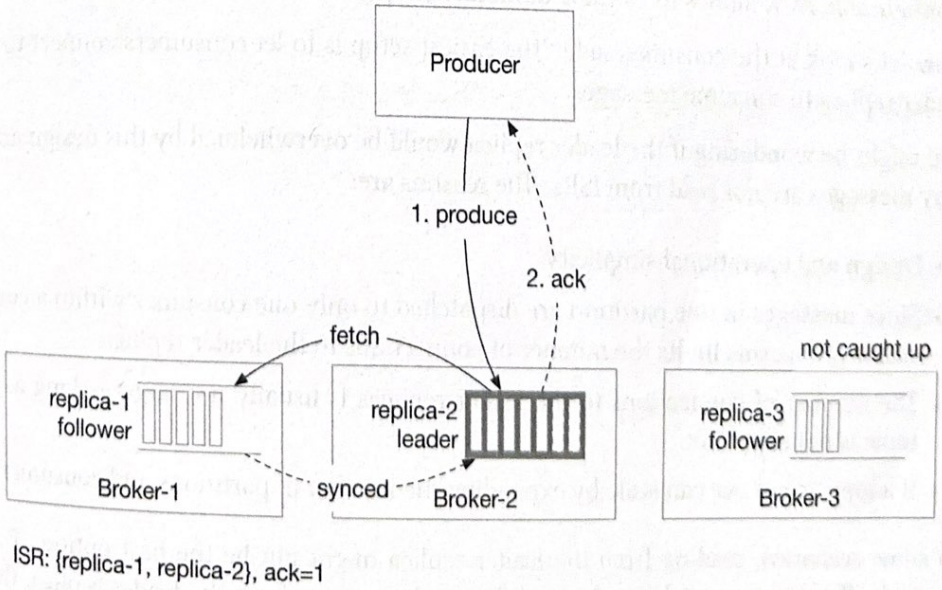


Figure 4.26: ACK=1

ACK=0

The producer keeps sending messages to the leader without waiting for any acknowledgment, and it never retries. This method provides the lowest latency at the cost of potential message loss. This setting might be good for use cases like collecting metrics or logging data since data volume is high and occasional data loss is acceptable.

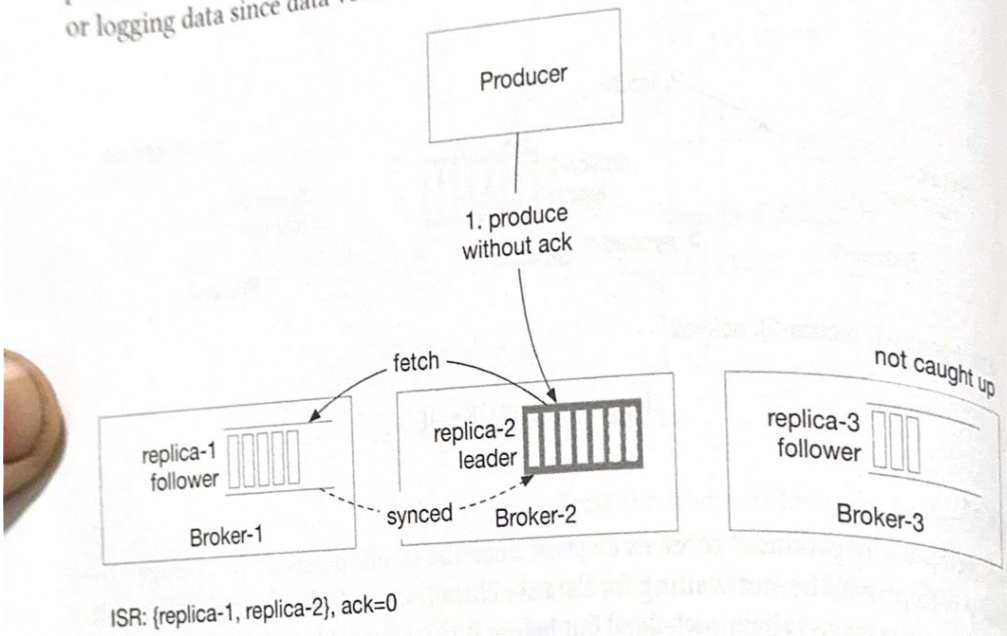


Figure 4.27: ACK=0

Configurable ACK allows us to trade durability for performance.

Now let's look at the consumer side. The easiest setup is to let consumers connect to a leader replica to consume messages.

You might be wondering if the leader replica would be overwhelmed by this design and why messages are not read from ISRs. The reasons are:

- Design and operational simplicity.
- Since messages in one partition are dispatched to only one consumer within a consumer group, this limits the number of connections to the leader replica.
- The number of connections to the leader replicas is usually not large as long as a topic is not super hot.
- If a topic is hot, we can scale by expanding the number of partitions and consumers.

In some scenarios, reading from the leader replica might not be the best option. For example, if a consumer is located in a different data center from the leader replica, the read performance suffers. In this case, it is worthwhile to enable consumers to read from the closest ISRs. Interested readers can check out the reference material about this [11].

ISR is very important. How does it determine if a replica is ISR or not? Usually, the leader

for every partition tracks the ISR list by computing the lag of every replica from itself. If you are interested in detailed algorithms, you can find the implementations in reference materials [12] [13].

Scalability

By now we have made great progress designing the distributed message queue system. In the next step, let's evaluate the scalability of different system components:

- Producers
- Consumers
- Brokers
- Partitions

Producer

The producer is conceptually much simpler than the consumer because it doesn't need group coordination. The scalability of producers can easily be achieved by adding or removing producer instances.

Consumer

Consumer groups are isolated from each other, so it is easy to add or remove a consumer group. Inside a consumer group, the rebalancing mechanism helps to handle the cases where a consumer gets added or removed, or when it crashes. With consumer groups and the rebalance mechanism, the scalability and fault tolerance of consumers can be achieved.

Broker

Before discussing scalability on the broker side, let's first consider the failure recovery of brokers.

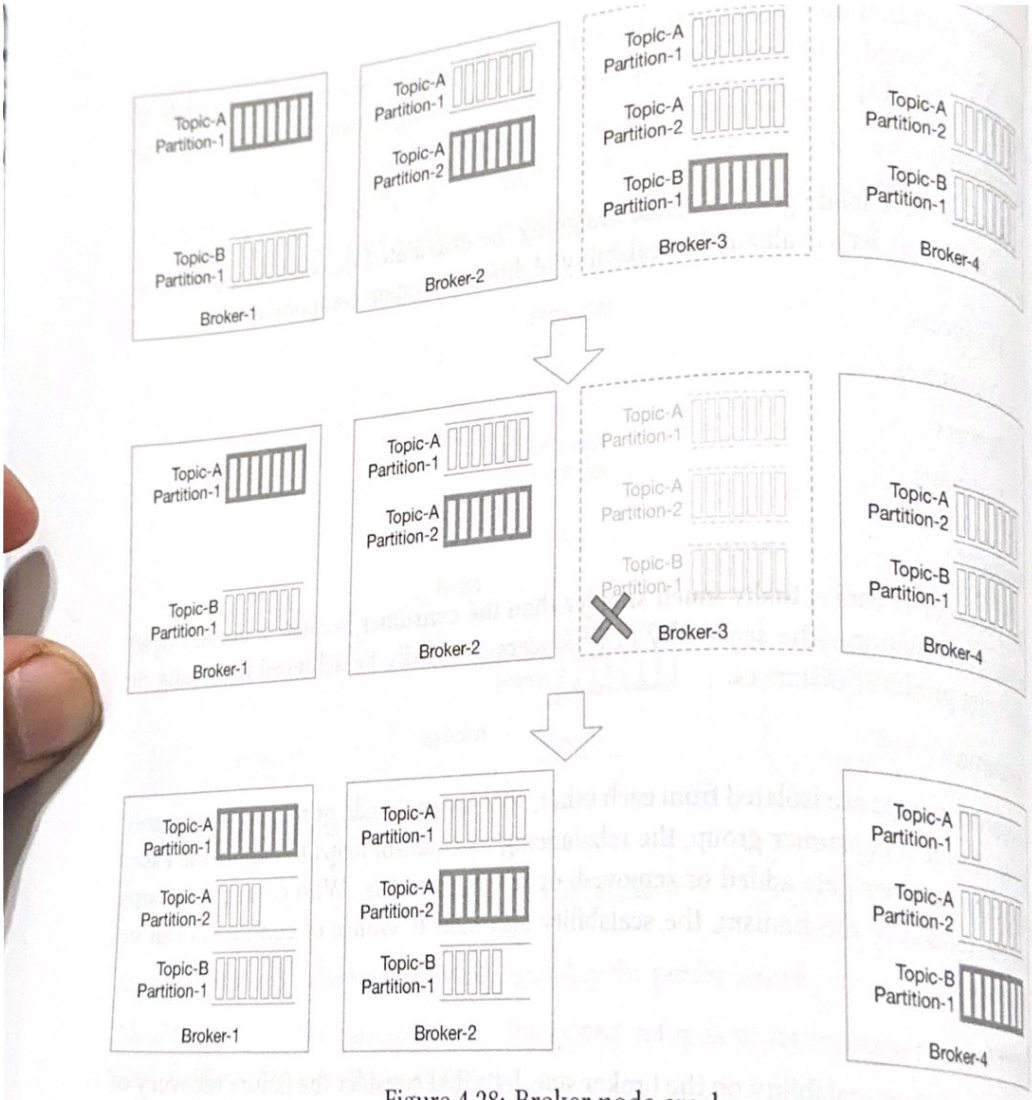


Figure 4.28: Broker node crashes

Let's use an example in Figure 4.28 to explain how failure recovery works.

1. Assume there are 4 brokers and the partition (replica) distribution plan is shown below:
   - Partition-1 of topic A: replicas in Broker-1 (leader), 2, and 3.
   - Partition-2 of topic A: replicas in Broker-2 (leader), 3, and 4.
   - Partition-1 of topic B: replicas in Broker-3 (leader), 4, and 1.
2. Broker-3 crashes, which means all the partitions on the node are lost. The partition distribution plan is changed to:
   - Partition-1 of topic A: replicas in Broker-1 (leader) and 2.
   - Partition-2 of topic A: replicas in Broker-2 (leader) and 4.
   - Partition-1 of topic B: replicas in Broker-4 and 1.

3. The broker controller detects Broker-3 is down and generates a new partition distribution plan for the remaining broker nodes:
    - Partition-1 of topic A: replicas in Broker-1 (leader), 2, and 4 (new).
    - Partition-2 of topic A: replicas in Broker-2 (leader), 4, and 1 (new).
    - Partition-1 of topic B: replicas in Broker-4 (leader), 1, and 2 (new).
4. The new replicas work as followers and catch up with the leader.

To make the broker fault-tolerant, here are additional considerations:

- The minimum number of ISRs specifies how many replicas the producer must receive before a message is considered to be successfully committed. The higher the number, the safer. But on the other hand, we need to balance latency and safety.
- If all replicas of a partition are in the same broker node, then we cannot tolerate the failure of this node. It is also a waste of resources to replicate data in the same node. Therefore, replicas should not be in the same node.
- If all the replicas of a partition crash, the data for that partition is lost forever. When choosing the number of replicas and replica locations, there's a trade-off between data safety, resource cost, and latency. It is safer to distribute replicas across data centers, but this will incur much more latency and cost, to synchronize data between replicas. As a workaround, data mirroring can help to copy data across data centers, but this is out of scope. The reference material [14] covers this topic.

Now let's get back to discussing the scalability of brokers. The simplest solution would be to redistribute the replicas when broker nodes are added or removed.

However, there is a better approach. The broker controller can temporarily allow more replicas in the system than the number of replicas in the config file. When the newly added broker catches up, we can remove the ones that are no longer needed. Let's use an example as shown in Figure 4.29 to understand the approach.

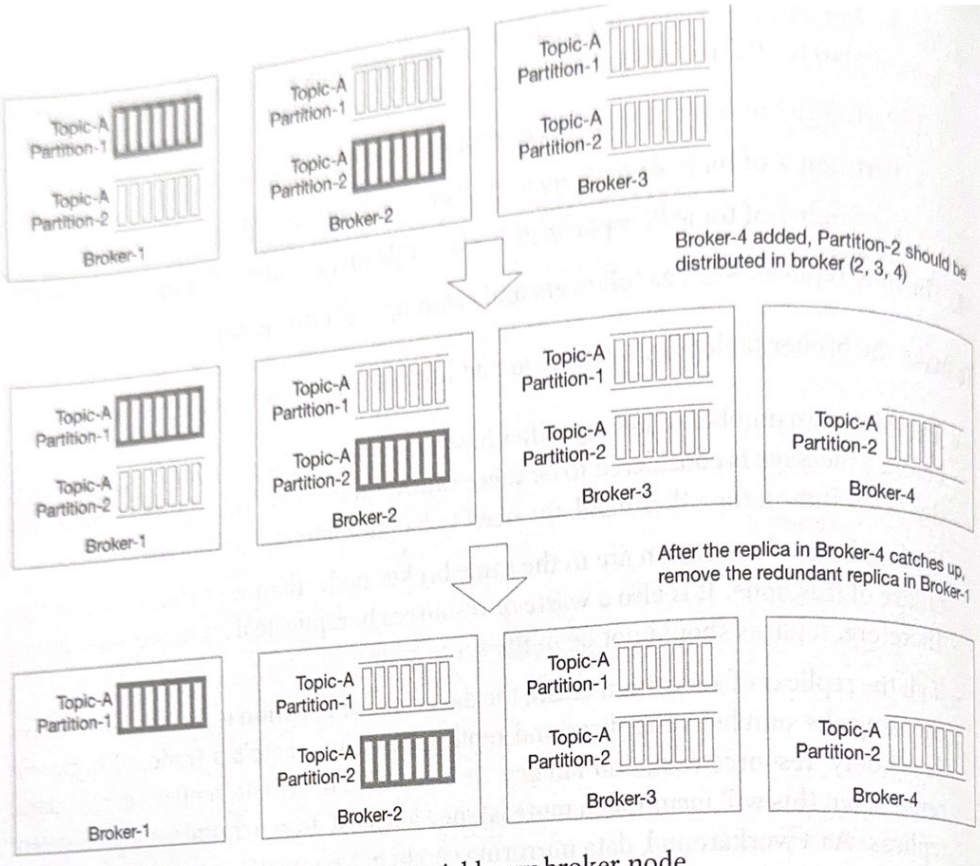


Figure 4.29: Add new broker node

1. The initial setup: 3 brokers, 2 partitions, and 3 replicas for each partition.
2. New Broker-4 is added. Assume the broker controller changes the replica distribution of Partition-2 to the broker (2, 3, 4). The new replica in Broker-4 starts to copy data from leader Broker-2. Now the number of replicas for Partition-2 is temporarily more than 3.
3. After the replica in Broker-4 catches up, the redundant partition in Broker-1 is gracefully removed.

By following this process, data loss while adding brokers can be avoided. A similar process can be applied to remove brokers safely.

Partition

For various operational reasons, such as scaling the topic, throughput tuning, balancing availability/ throughput, etc., we may change the number of partitions. When the number of partitions changes, the producer will be notified after it communicates with any broker, and the consumer will trigger consumer rebalancing. Therefore, it is safe for both the producer and consumer.

Now let's consider the data storage layer when the number of partitions changes. As in Figure 4.30, we have added a partition to the topic.

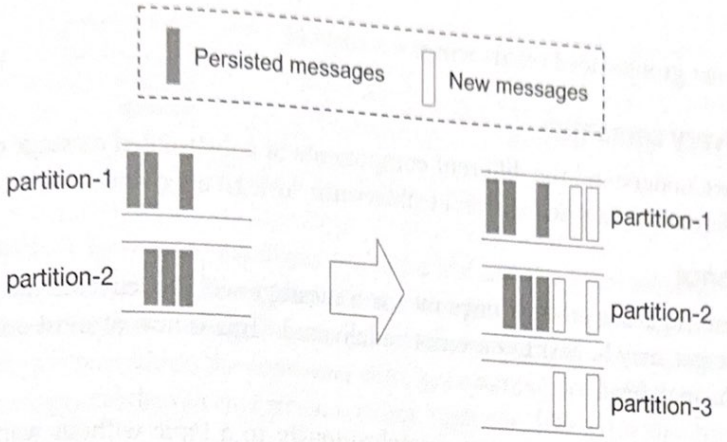


Figure 4.30: Partition increase

- Persisted messages are still in the old partitions, so there's no data migration.
- After the new partition (partition-3) is added, new messages will be persisted in all 3 partitions.

So it is straightforward to scale the topic by increasing partitions.

Decrease the number of partitions

Decreasing partitions is more complicated, as illustrated in Figure 4.31.

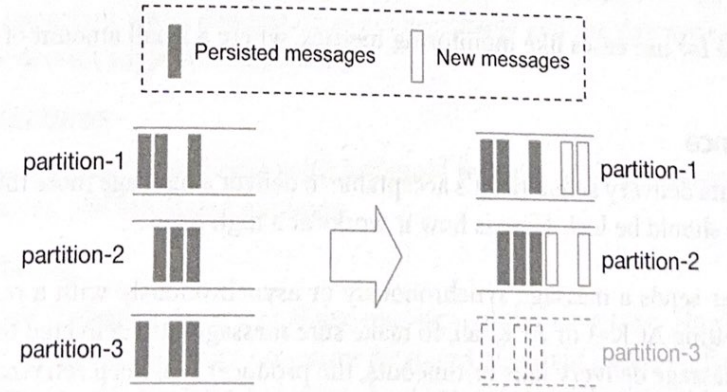


Figure 4.31: Partition decrease

- Partition-3 is decommissioned so new messages are only received by the remaining partitions (partition-1 and partition-2).
- The decommissioned partition (partition-3) cannot be removed immediately because data might be currently consumed by consumers for a certain amount of time. Only after the configured retention period passes, data can be truncated and storage space is freed up. Reducing partitions is not a shortcut to reclaiming data space.
- During this transitional period (while partition-3 is decommissioned), producers only send messages to the remaining 2 partitions, but consumers can still consume from all 3 partitions. After the retention period of the decommissioned partition expires,

consumer groups need rebalancing.

Data delivery semantics

Now that we understand the different components of a distributed message queue, let's discuss different delivery semantics: at-most once, at-least once, and exactly once.

At-most once

As the name suggests, at-most once means a message will be delivered not more than once. Messages may be lost but are not redelivered. This is how at-most once delivery works at the high level.

- The producer sends a message asynchronously to a topic without waiting for an acknowledgment (ACK=0). If message delivery fails, there is no retry.
- Consumer fetches the message and commits the offset before the data is processed. If the consumer crashes just after offset commit, the message will not be re-consumed.

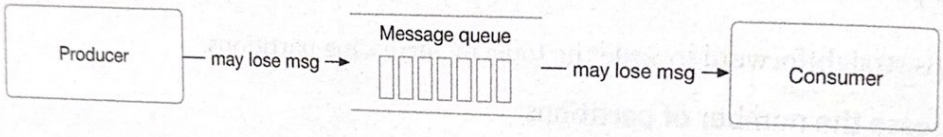


Figure 4.32: At-most once

It is suitable for use cases like monitoring metrics, where a small amount of data loss is acceptable.

At-least once

With this data delivery semantic, it's acceptable to deliver a message more than once, but no message should be lost. Here is how it works at a high level.

- Producer sends a message synchronously or asynchronously with a response callback, setting ACK=1 or ACK=all, to make sure messages are delivered to the broker. If the message delivery fails or timeouts, the producer will keep retrying.
- Consumer fetches the message and commits the offset only after the data is successfully processed. If the consumer fails to process the message, it will re-consume the message so there won't be data loss. On the other hand, if a consumer processes the message but fails to commit the offset to the broker, the message will be re-consumed when the consumer restarts, resulting in duplicates.
- A message might be delivered more than once to the brokers and consumers.

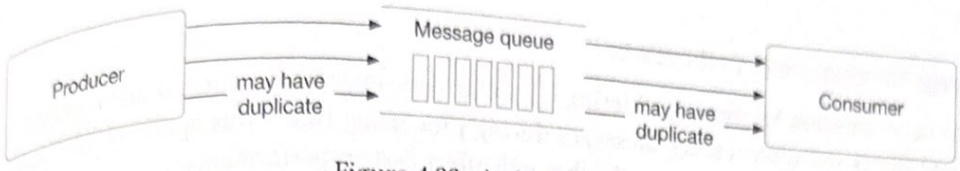


Figure 4.33: At-least once

Use cases: With at-least once, messages won't be lost but the same message might be delivered multiple times. While not ideal from a user perspective, at-least once delivery semantics are usually good enough for use cases where data duplication is not a big issue or deduplication is possible on the consumer side. For example, with a unique key in each message, a message can be rejected when writing duplicate data to the database.

Exactly once

Exactly once is the most difficult delivery semantic to implement. It is friendly to users, but it has a high cost for the system's performance and complexity.

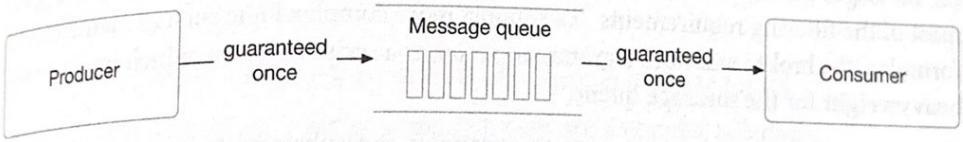


Figure 4.34: Exactly once

Use cases: Financial-related use cases (payment, trading, accounting, etc.). Exactly once is especially important when duplication is not acceptable and the downstream service or third party doesn't support idempotency.

Advanced features

In this section, we talk briefly about some advanced features, such as message filtering, delayed messages, and scheduled messages.

Message filtering

A topic is a logical abstraction that contains messages of the same type. However, some consumer groups may only want to consume messages of certain subtypes. For example, the ordering system sends all the activities about the order to a topic, but the payment system only cares about messages related to checkout and refund.

One option is to build a dedicated topic for the payment system and another topic for the ordering system. This method is simple, but it might raise some concerns.

- What if other systems ask for different subtypes of messages? Do we need to build dedicated topics for every single consumer request?
- It is a waste of resources to save the same messages on different topics.
- The producer needs to change every time a new consumer requirement comes, as the producer and consumer are now tightly coupled.

Therefore, we need to resolve this requirement using a different approach. Luckily, mes-

sage filtering comes to the rescue.

A naive solution for message filtering is that the consumer fetches the full set of messages and filters out unnecessary messages during processing time. This approach is flexible but introduces unnecessary traffic that will affect system performance.

A better solution is to filter messages on the broker side so that consumers will only get messages they care about. Implementing this requires some careful consideration. If data filtering requires data decryption or deserialization, it will degrade the performance of the brokers. Additionally, if messages contain sensitive data, they should not be readable in the message queue.

Therefore, the filtering logic in the broker should not extract the message payload. It is better to put data used for filtering into the metadata of a message, which can be efficiently read by the broker. For example, we can attach a tag to each message. With a message tag, a broker can filter messages in that dimension. If more tags are attached, the messages can be filtered in multiple dimensions. Therefore, a list of tags can support most of the filtering requirements. To support more complex logic such as mathematical formulae, the broker will need a grammar parser or a script executor, which might be too heavyweight for the message queue.

With tags attached to each message, a consumer can subscribe to messages based on the specified tag, as shown in Figure 4.35. Interested readers can refer to the reference material [15].

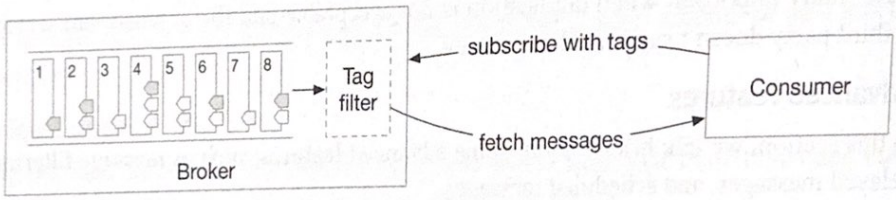


Figure 4.35: Message filtering by tags

Delayed messages & scheduled messages

Sometimes you want to delay the delivery of messages to a consumer for a specified period of time. For example, an order should be closed if not paid within 30 minutes after the order is created. A delayed verification message (check if the payment is completed) is sent immediately but is delivered to the consumer 30 minutes later. When the consumer receives the message, it checks the payment status. If the payment is not completed, the order will be closed. Otherwise, the message will be ignored.

Different from sending instant messages, we can send delayed messages to temporary storage on the broker side instead of to the topics immediately, and then deliver them to the topics when time's up. The high-level design for this is shown in Figure 4.36.

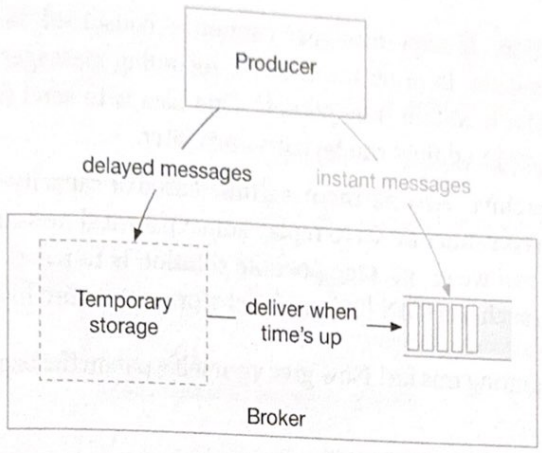


Figure 4.36: Delayed messages

Core components of the system include the temporary storage and the timing function.

- The temporary storage can be one or more special message topics.
- The timing function is out of scope, but here are 2 popular solutions:
  - Dedicated delay queues with predefined delay levels [16]. For example, RocketMQ doesn't support delayed messages with arbitrary time precision, but delayed messages with specific levels are supported. Message delay levels are 1s, 5s, 10s, 30s, 1m, 2m, 3m, 4m, 6m, 8m, 9m, 10m, 20m, 30m, 1h, and 2h.
  - Hierarchical time wheel [17].

A scheduled message means a message should be delivered to the consumer at the scheduled time. The overall design is very similar to delayed messages.

Step 4 - Wrap Up

In this chapter, we have presented the design of a distributed message queue with some advanced features commonly found in data streaming platforms. If there is extra time at the end of the interview, here are some additional talking points:

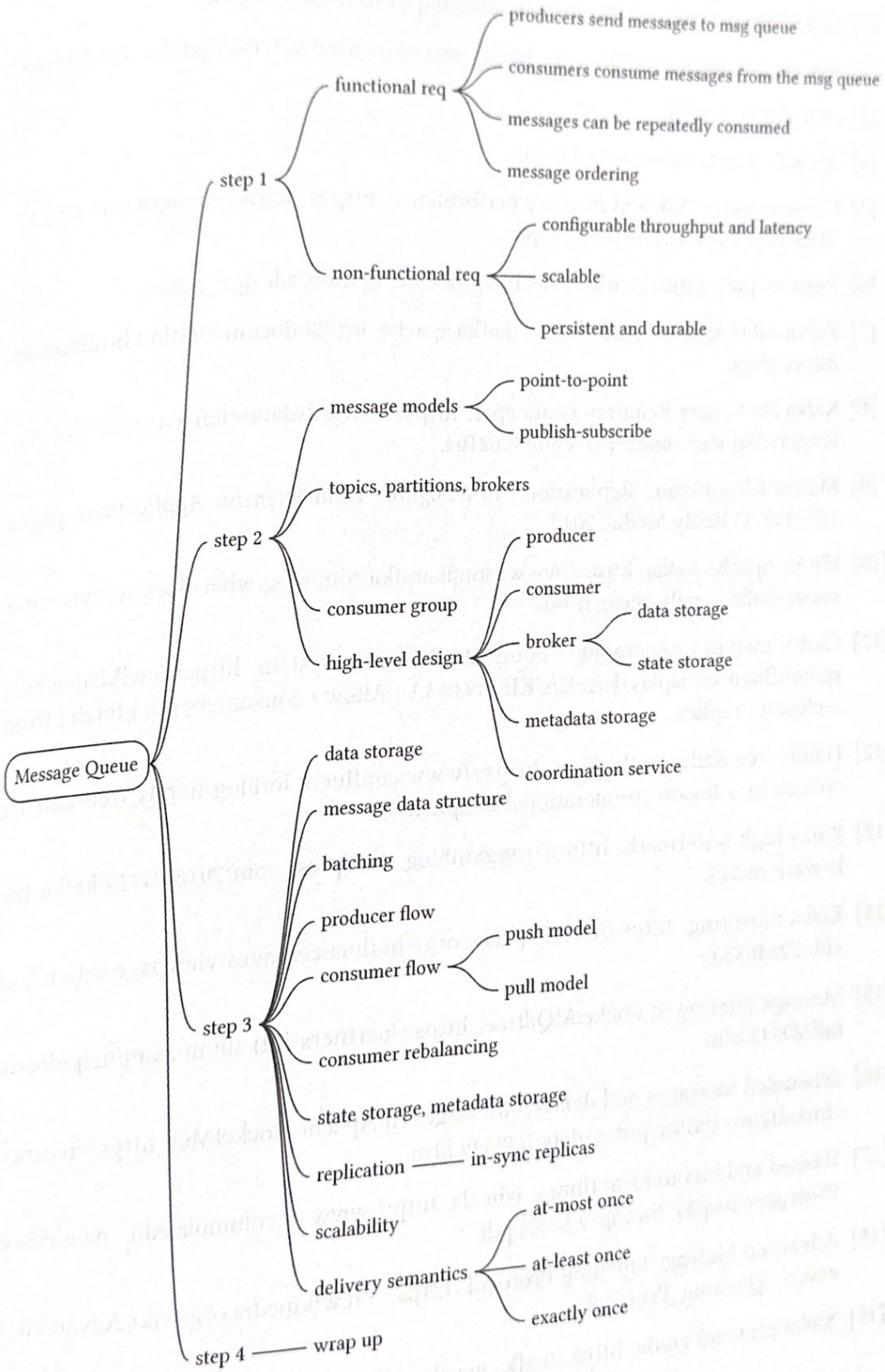
- Protocol: it defines rules, syntax, and APIs on how to exchange information and transfer data between different nodes. In a distributed message queue, the protocol should be able to:
  - Cover all the activities such as production, consumption, heartbeat, etc.
  - Effectively transport data with large volumes.
  - Verify the integrity and correctness of the data.

  Some popular protocols include Advanced Message Queuing Protocol (AMQP) [18] and Kafka protocol [19].

- Retry consumption. If some messages cannot be consumed successfully, we need to retry the operation. In order not to block incoming messages, how can we retry the operation after a certain time period? One idea is to send failed messages to a dedicated retry topic, so they can be consumed later.
- Historical data archive. Assume there is a time-based or capacity-based log retention mechanism. If a consumer needs to replay some historical messages that are already truncated, how can we do it? One possible solution is to use storage systems with large capacities, such as HDFS [20] or object storage, to store historical data.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Queue Length Limit. https://www.rabbitmq.com/maxlength.html.

[2] Apache ZooKeeper - Wikipedia. https://en.wikipedia.org/wiki/Apache_ZooKeepe

[3] etcd. https://etcd.io/.

[4] MySQL. https://www.mysql.com/.

[5] Comparison of disk and memory performance. https://deliveryimages.acm.org/1(1145/1570000/1563874/jacobs3.jpg.

[6] Push vs. pull. https://kafka.apache.org/documentation/#design_pull.

[7] Kafka 2.0 Documentation. https://kafka.apache.org/20/documentation.html#consu merconfigs.

[8] Kafka No Longer Requires ZooKeeper. https://towardsdatascience.com/kafka-no- longer-requires-zookeeper-ebfbf3862104.

[9] Martin Kleppmann. Replication. In *Designing Data-Intensive Applications*, pages 151–197. O'Reilly Media, 2017.

ISR in Apache Kafka. https://www.cloudkarafka.com/blog/what-does-in-sync-in-a pache-kafka-really-mean.html.

[11] Global map in a geographic Coordinate Reference System. https://cwiki.apache.o rg/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from +closest+replica.

[12] Hands-free Kafka Replication. https://www.confluent.io/blog/hands-free-kafka-r eplication-a-lesson-in-operational-simplicity/.

[13] Kafka high watermark. https://rongxinblog.wordpress.com/2016/07/29/kafka-hig h-watermark/.

[14] Kafka mirroring. https://cwiki.apache.org/confluence/pages/viewpage.action?pag eId=27846330.

[15] Message filtering in RocketMQdtree. https://partners-intl.aliyun.com/help/doc-de tail/29543.htm.

[16] Scheduled messages and delayed messages in Apache RocketMQ. https://partners -intl.aliyun.com/help/doc-detail/43349.htm.

[17] Hashed and hierarchical timing wheels. http://www.cs.columbia.edu/~nahum/w69 98/papers/sosp87-timing-wheels.pdf.

[18] Advanced Message Queuing Protocol. https://en.wikipedia.org/wiki/Advanced_M essage_Queuing_Protocol.

[19] Kafka protocol guide. https://kafka.apache.org/protocol.