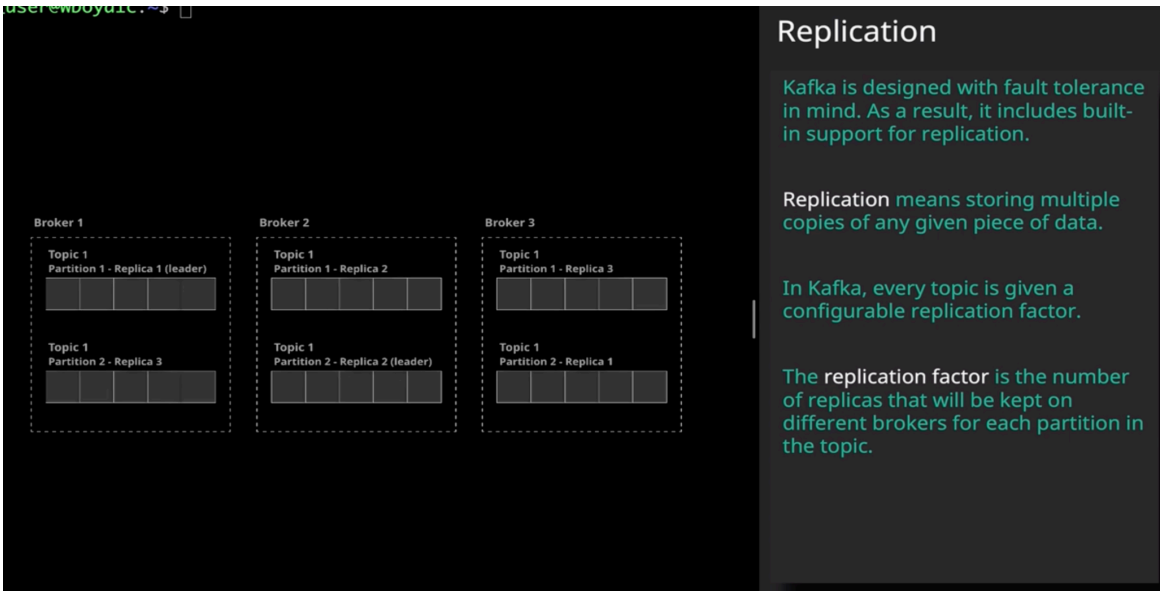
****
https://github.com/confluentinc/cp-all-in-one/blob/7.5.0-post/cp-all-in-one/docker-compose.ym
|****

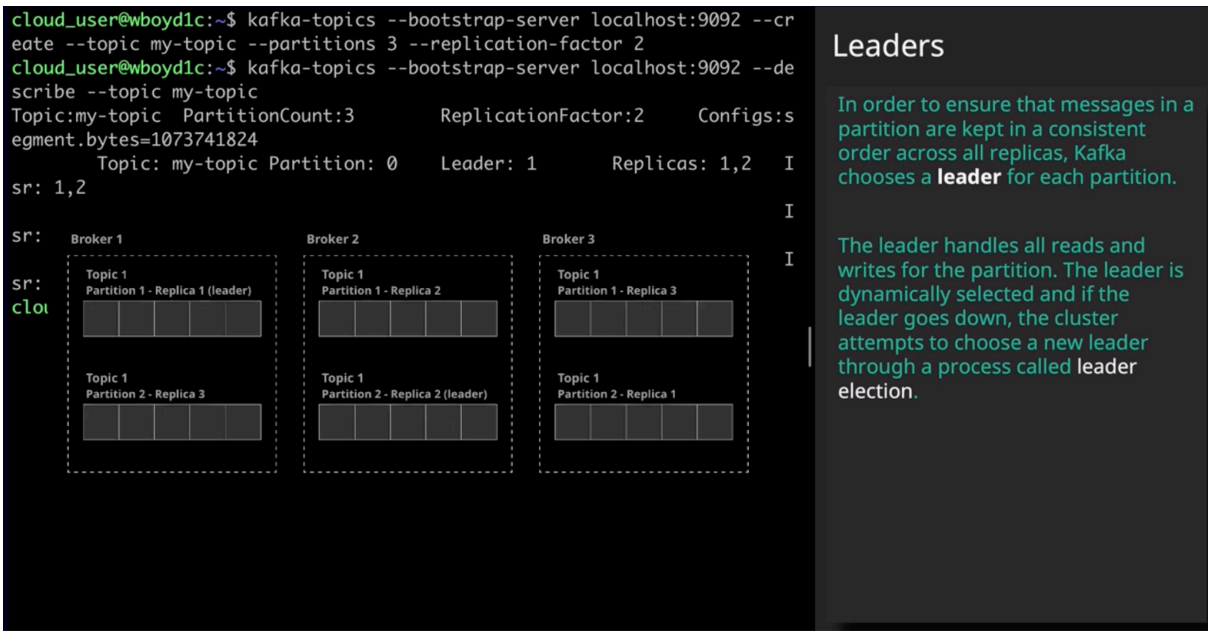
In Kafka, the **replication factor** determines how many copies of data are stored across several Kafka brokers. Here's what you need to know:

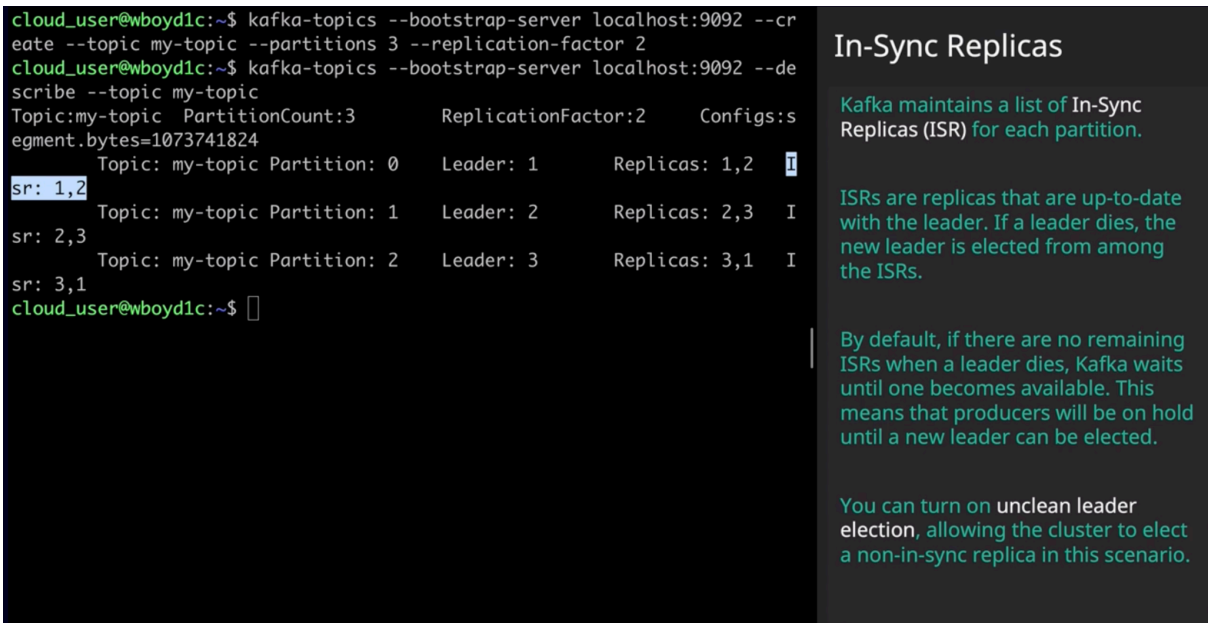
1. **Replication Factor Configuration:**
    - The replication factor is set at the **topic level** during topic creation.
    - It specifies the total number of replicas, including the **leader**.
    - Topics with a replication factor of **1** have no replication (not recommended for production).
    - In production, use a replication factor of **2 or 3** for high availability.
2. **How Replication Works:**
    - Kafka replicates data across brokers to ensure availability and prevent data loss.
    - Each partition has a **single leader** and zero or more **followers**.
    - Followers consume messages from the leader and apply them to their own log.
    - The leader keeps track of **in-sync replicas** (nodes that replicate writes from the leader).
    - If a follower fails or falls behind, the leader removes it from the in-sync list.
3. **Message Durability:**
    - Kafka guarantees that a **committed message** won't be lost.
    - Committed messages are applied to all in-sync replicas.
    - Consumers only receive committed messages, ensuring data consistency.

Remember to choose an appropriate replication factor based on your Kafka cluster's requirements!




```
cloud_user@wboyd1c:~$ kafka-topics --bootstrap-server localhost:9092 --cr
eate --topic my-topic --partitions 3 --replication-factor 2
cloud_user@wboyd1c:~$ kafka-topics --bootstrap-server localhost:9092 --de
scribe --topic my-topic
Topic:my-topic  PartitionCount:3        ReplicationFactor:2     Configs:s
egment.bytes=1073741824
        Topic: my-topic Partition: 0    Leader: 1       Replicas: 1,2   I
sr: 1,2
        Topic: my-topic Partition: 1    Leader: 2       Replicas: 2,3   I
sr: 2,3
        Topic: my-topic Partition: 2    Leader: 3       Replicas: 3,1   I
sr: 3,1
cloud_user@wboyd1c:~$
```

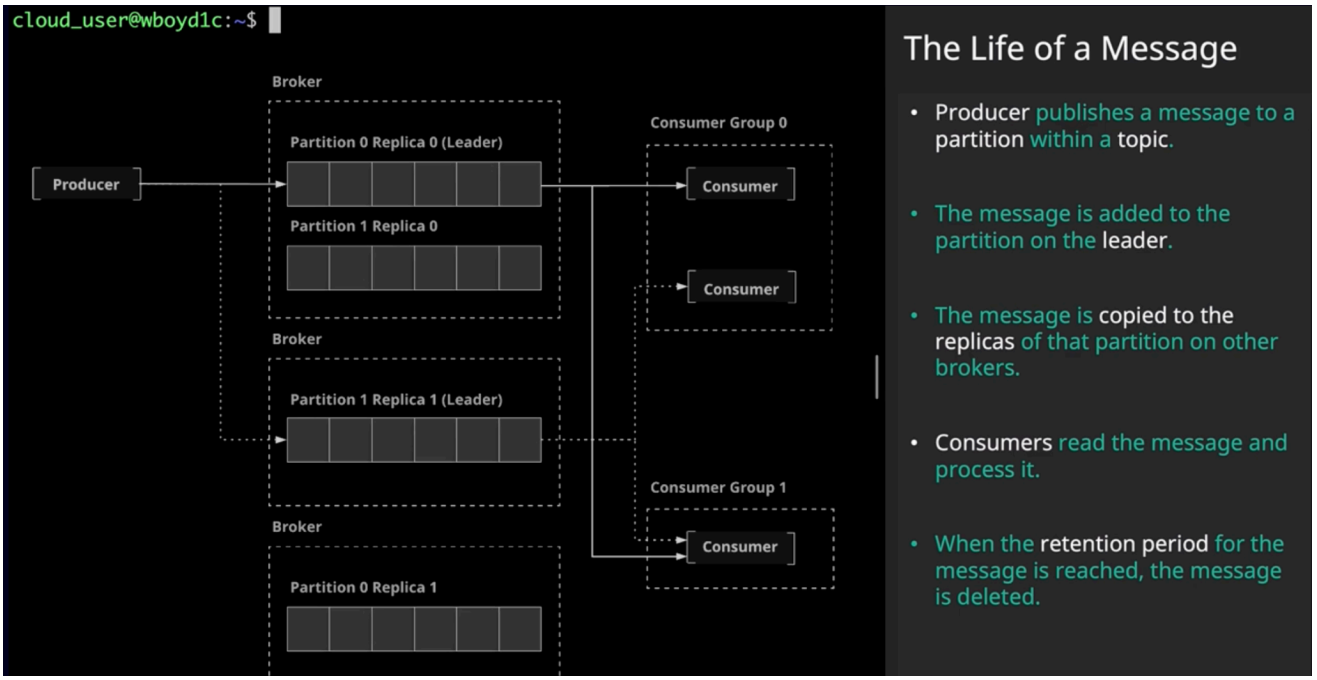




ISR (In-Sync Replicas) and **acks** in the context of Apache Kafka:

1. **In-Sync Replicas (ISR):**
    - ISR refers to the set of replicas that are **up-to-date** with the leader for a given partition.
    - When a producer sends a message, it is written to the leader's log first.
    - The leader then replicates the message to the ISR.
    - Only messages in the ISR are considered **committed**.
    - If a replica falls behind or becomes unavailable, it is removed from the ISR until it catches up.

2. **Acknowledgments (acks):**
    - Acks determine how many replicas must acknowledge a produce request before it's considered successful.
    - There are three ack levels:
        - `acks=0`: No acknowledgment. The producer assumes success after sending the message to the leader.
        - `acks=1`: The leader acknowledges the message. This is the default setting.
        - `acks=all` (or `-1`): All in-sync replicas must acknowledge the message for it to be considered committed.
3. **Choosing the Right Configuration:**
    - For **durability**, use `acks=all`. This ensures that data won't be lost even if a broker fails.
    - For **performance**, `acks=1` is faster since it requires only the leader's acknowledgment.
    - Consider your use case and trade-offs when configuring acks.



Monitoring:

https://docs.confluent.io/platform/current/kafka/monitoring.html#kraft-broker-metrics

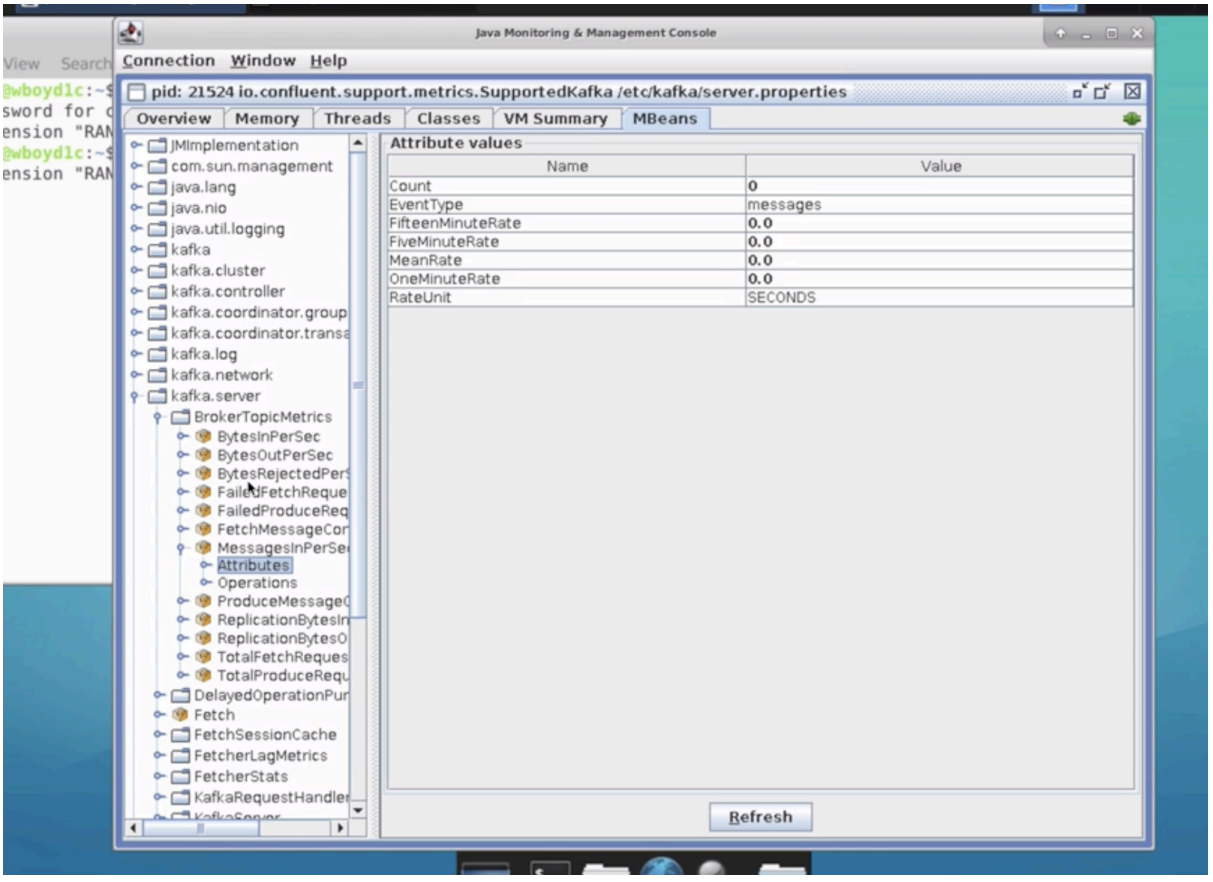
```
./bin/kafka-storage.sh random-uuid



QCFQxofwRDewkHgNEBBzaw



./bin/kafka-storage.sh format -t QCFQxofwRDewkHgNEBBzaw -c
./config/kraft/controller-1.properties

./bin/kafka-storage.sh format -t QCFQxofwRDewkHgNEBBzaw -c
./config/kraft/controller-2.properties

./bin/kafka-storage.sh format -t QCFQxofwRDewkHgNEBBzaw -c
./config/kraft/controller-3.properties

./bin/kafka-storage.sh format -t QCFQxofwRDewkHgNEBBzaw -c
./config/kraft/broker-1.properties

./bin/kafka-storage.sh format -t QCFQxofwRDewkHgNEBBzaw -c
./config/kraft/broker-2.properties

./bin/kafka-storage.sh format -t QCFQxofwRDewkHgNEBBzaw -c
./config/kraft/broker-3.properties



./bin/kafka-server-start.sh ./config/kraft/controller-1.properties

./bin/kafka-server-start.sh ./config/kraft/controller-2.properties

./bin/kafka-server-start.sh ./config/kraft/controller-3.properties




./bin/kafka-metadata-shell.sh --snapshot
/tmp/kraft-controller-1-logs/__cluster_metadata-0/00000000000000000000.log



./bin/kafka-server-start.sh ./config/kraft/broker-1.properties

./bin/kafka-server-start.sh ./config/kraft/broker-2.properties

./bin/kafka-server-start.sh ./config/kraft/broker-3.properties
```

Metrics and Monitoring

Both Kafka and ZooKeeper offer metrics through JMX, a built-in Java technology for providing and accessing performance metric data.

You can access metrics from Kafka brokers as well as your Kafka client applications.

You can find a full list of available metrics in the Kafka documentation.



Structured Streaming Metrics (SMM) is a powerful tool for monitoring and analyzing Kafka clusters. It provides detailed insights into the performance, resource utilization, and overall health of your Kafka brokers. Here are some key points about using SMM:

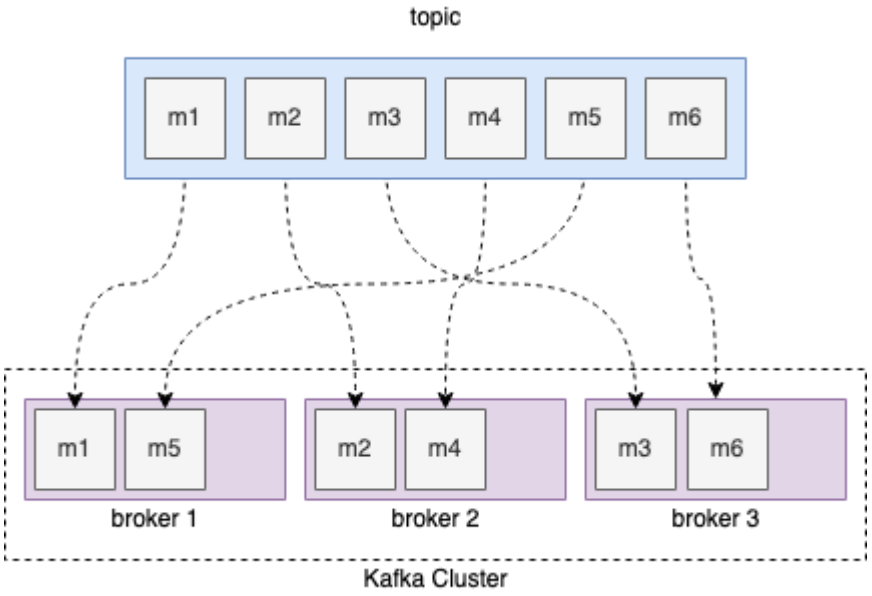
1. **Installation and Configuration**:
    - Install SMM on your Kafka brokers by following the official documentation.
    - Configure SMM properties in the `server.properties` file to specify the metrics you want to collect and the reporting intervals.
2. **Metrics Collection**:
    - SMM collects various metrics related to Kafka brokers, topics, partitions, and consumers.
    - Examples of collected metrics include:
        - **Broker Metrics**: CPU usage, memory, disk I/O, network traffic, etc.
        - **Topic Metrics**: Message rate, byte rate, partition count, etc.
        - **Partition Metrics**: Under-replicated partitions, leader election rate, etc.
        - **Consumer Metrics**: Lag, offset commit rate, etc.
3. **Visualization and Alerts**:
    - Use the SMM dashboard to visualize metrics over time.
    - Set up alerts based on thresholds to receive notifications when specific metrics deviate from expected values.
4. **Troubleshooting and Optimization**:
    - Analyze metrics to identify bottlenecks, anomalies, or performance issues.
    - Optimize Kafka configuration based on SMM insights.

Remember that SMM complements other monitoring tools like Prometheus and Grafana.

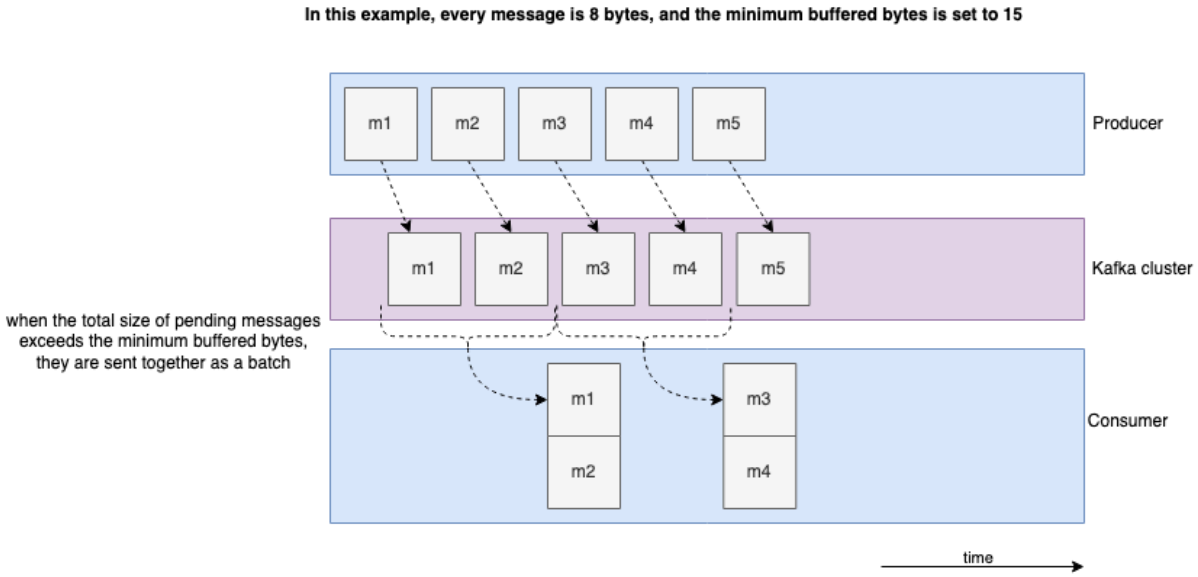
To set up and analyze Kafka clusters using **Streams Messaging Manager (SMM)**, follow these steps:

1. **Install and Configure SMM**:
    - Install SMM on your Kafka brokers by following the official documentation.
    - Configure SMM properties in the `server.properties` file to specify the metrics you want to collect and the reporting intervals[1].
2. **Enable SRM Integration**:
    - Use **Streams Replication Manager (SRM)** to implement cross-cluster Kafka topic replication.
    - Configure SRM properties in SMM to enable integration between SRM and SMM.
    - After configuring SRM in SMM, you can monitor all Kafka cluster replications targeting the configured cluster[2].
3. **Monitor Replications**:
    - In the SMM UI, navigate to the **Cluster Replications** section.
    - Monitor the status of Kafka cluster replications, the number of topics associated with replication, throughput, replication latency, and checkpoint latency.
    - Utilize SMM's alerting features to receive alerts related to your replications[1].

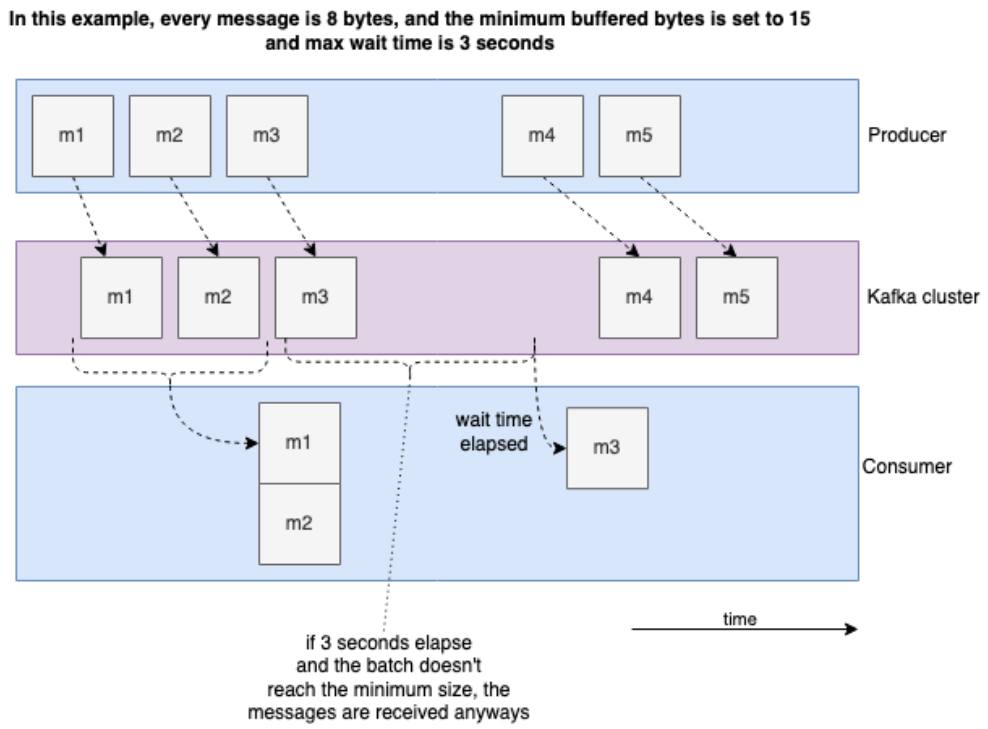
Remember that SMM provides valuable insights into Kafka cluster performance, making it easier to manage and optimize your Kafka environment.



Minimum Buffered Bytes



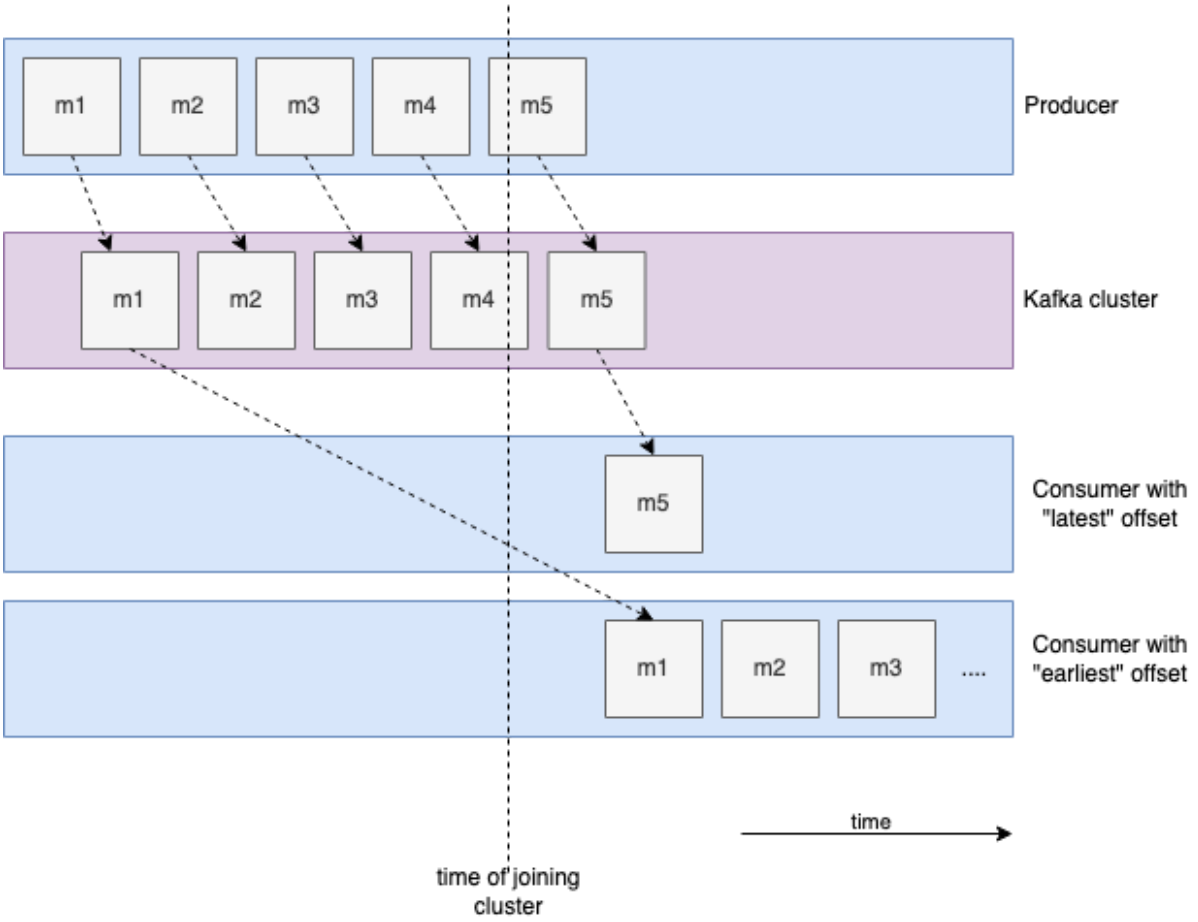
Max Wait Time



Start Offset

When a new consumer is added to a topic, it has two options for where it wants to start consuming data from:

1. **Earliest** - The consumer will start consuming data for a topic starting from the earliest message that is available.
2. **Latest** - Only consume new messages that appear after the consumer has joined the cluster. This is the default setting.



We can set the `auto.offset.reset` property to `earliest` or `latest` (the default) depending on our requirements

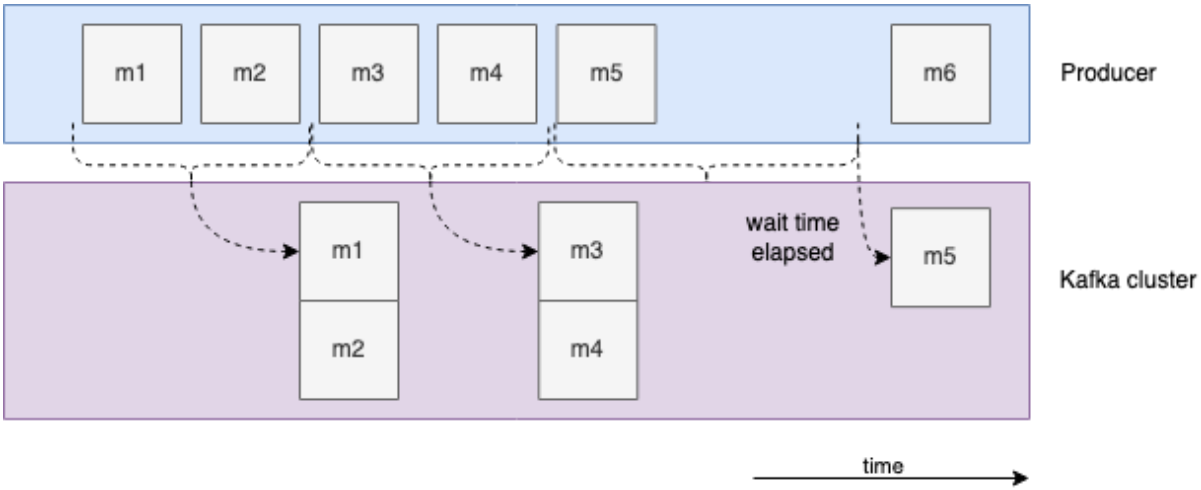
Producer Side Configuration:

Message Batching

So far we've looked at configuration on the consumer side. Let's take a look at some producer/writer side configuration options.
Similar to the consumer, the producer also tries to send messages in batches. This is to reduce the total number of network round trips and improve efficiency in writing messages, but comes at the cost of increased overall latency.

When batching messages, we can set:

1. Batch Size - The total number of messages that should be buffered before writing to the Kafka brokers.
2. Batch Timeout (Linger) - The maximum time before which messages are written to the brokers. That means that even if the message batch is not full, they will still be written onto the Kafka cluster once this time period has elapsed.



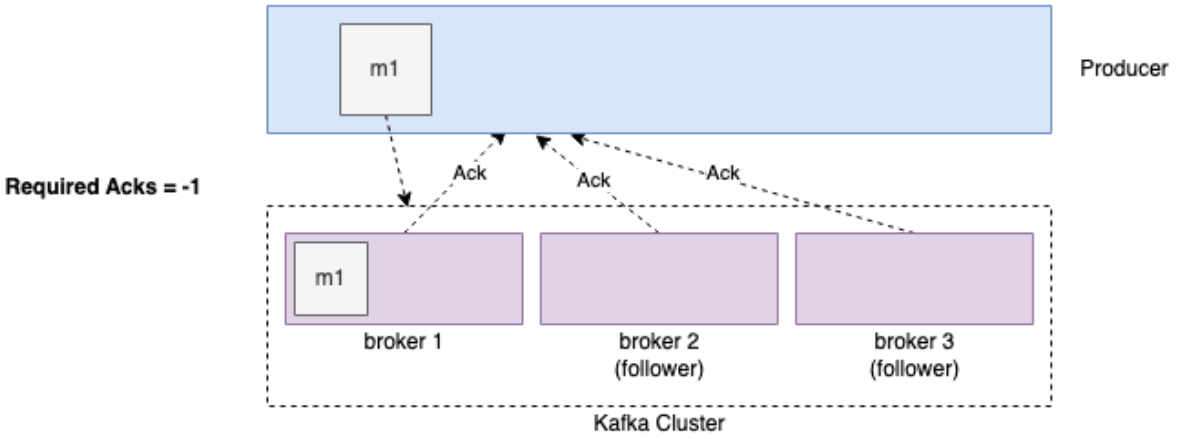
Required Acknowledgements

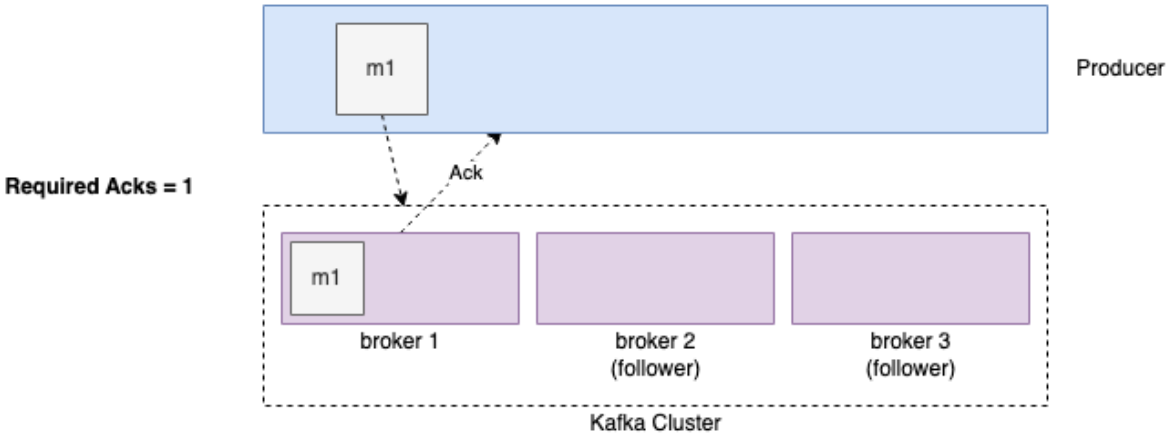
When we call the `producer.send` method in our example, it returns a future that resolves once the message is confirmed to be written. However, the definition of what "confirmed" means can be different based on your settings.
Remember, the Kafka cluster (and your topic partitions) is distributed between multiple brokers. Of these, one of the brokers is the designated leader and the rest are followers.
Keeping this in mind, there are three modes of acknowledgement (represented by integers) when writing messages to the cluster:

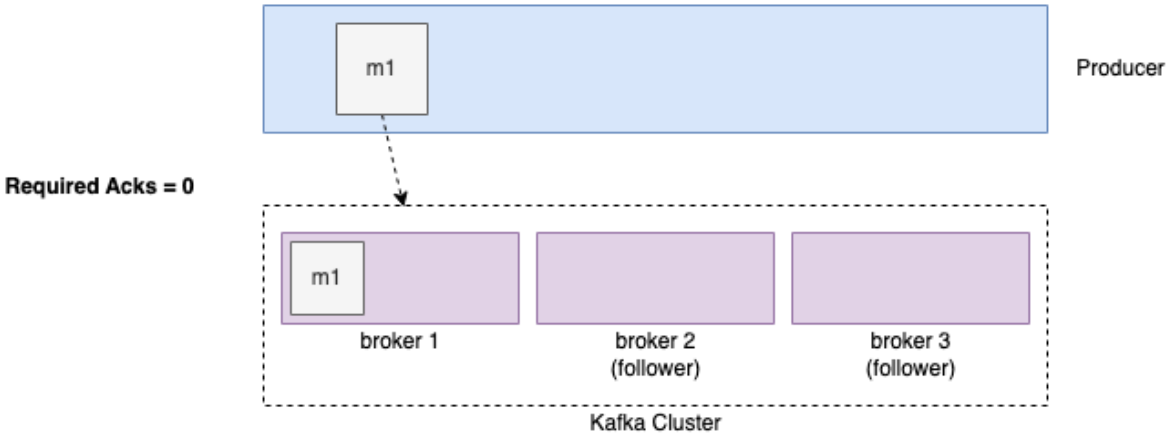
1. **All brokers acknowledge** that they have received the message (represented as `-1`)

2. Only the **leading broker acknowledges** that it has received the messages (represented as `1`). The remaining brokers can still eventually receive the message, but we won't wait for them to do so.
3. **No one acknowledges** receiving the message (represented as `0`). This is basically a fire-and-forget mode, where we don't care if our message is received or not. This should only be used for data that you are ok with losing a bit of, but require high throughput for.

In this example, broker 1 is the leader for message "m1"
while broker 2 and 3 are followers (or replicas)







Implementing Idempotent Producers in Kafka

In the context of distributed systems, idempotence refers to the ability to apply an operation multiple times without changing the result beyond the initial application. In Apache Kafka, implementing idempotent producers is critical for ensuring that messages are delivered exactly once, avoiding duplicates that can lead to data inconsistency and other issues.

Why Idempotence Matters

Without idempotence, network errors or service interruptions could cause producers to resend messages, leading to duplicate records in Kafka topics. This is problematic for consumers that need accurate data for processing. Idempotent producers eliminate this concern by ensuring that retries do not result in duplicate data.

Implementing Idempotent Producers in Kafka

To implement an idempotent producer in Kafka, you must configure your producer client with specific settings. The key configurations are:

- `enable.idempotence`: Set to `true` to enable idempotent delivery.
- `acks`: Set to `all` to ensure full acknowledgment from all in-sync replicas.
- `retries`: Configure the number of retries for transient errors.
- `max.in.flight.requests.per.connection`: Limit concurrent sends to maintain order.

The following code snippet demonstrates how to configure an idempotent Kafka producer using Java:

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

props.put("enable.idempotence", true);

props.put("acks", "all");

props.put("retries", Integer.MAX_VALUE);

props.put("max.in.flight.requests.per.connection", 5);


KafkaProducer<String, String> producer = new
KafkaProducer<>(props);
```

Ensuring Unique Identifiers

Besides configuring the producer client, it's also important to ensure that each message has a unique identifier. This often involves using a combination of a primary key and a sequence number. The Kafka producer uses this information to de-duplicate messages on retries.

Handling Errors and Retries

An idempotent producer will handle errors and retries internally. However, developers should still be aware of potential exceptions thrown by the producer and handle them appropriately in their application logic.

Kafka Producer Transactional Support

In addition to idempotence, Kafka also supports transactions. By setting `transactional.id`, you can ensure exactly-once semantics across multiple partitions and topics within a single transaction.

```
props.put("transactional.id", "my-transactional-id");

producer.initTransactions();


try {

producer.beginTransaction();

sendRecords(producer);

producer.commitTransaction();

} catch (ProducerFencedException | OutOfOrderSequenceException
| AuthorizationException e) {

// Fatal exceptions, should close the producer

producer.close();

} catch (KafkaException e) {

// For other exceptions

producer.abortTransaction();

}
```

This ensures that either all messages are written to the log, or none of them are if something goes wrong.