# 신뢰성 있게 카프카를 사용하는 방법

## Goal

Apache Kafka 를 신뢰성 있게 사용하는 방법에 대해 알아보자.

## 카프카에 대한 기본적인 내용

앞으로 설명할 내용은 카프카에 대한 기본 지식을 가정하에 진행되므로, 카프카를 처음 접한 다면 아래의 코스를 통해 미리 학습할 것을 권장합니다.

### Short Course (Korean)

- Kafka 조금 아는 척하기 시리즈 1
- Kafka 조금 아는 척하기 시리즈 2 (Producer)
- Kafka 조금 아는 척하기 시리즈 3 (Consumer)

### Short Course (English)

- Confluent Kafka 101 Course

## 카프카가 기본적으로 제공해주는 신뢰성

데이터베이스 트랜잭션은 'ACID' 기능을 신뢰성 있게 제공한다. 마찬가지로 카프카 역시 기본적으로 신뢰성 있게 제공하는 기능들이 있다.

신뢰성은 **"시스템이 예상한대로 올바르게 작동하는 것"** 을 말한다. 카프카에게 원하는 신뢰성은 아마도 보낸 메시지가 **정확히 전달되고, 저장되며, 처리되는 것**을 말할 것이다. 즉, 메시지가 전달되지 않거나, 유실되거나, 갑자기 삭제되거나, 메시지 처리가 스킵되거나, 여러번 처리되지 않는 것들을 포함할 것이다.

카프카가 매력적인 이유는 기본적으로 제공하는 신뢰성이 있고, 개발자가 어떻게 설정하냐에 따라서 원하는 수준의 신뢰성을 달성할 수 있다.

다음은 카프카에서 기본적으로 제공해주는 신뢰성이다.

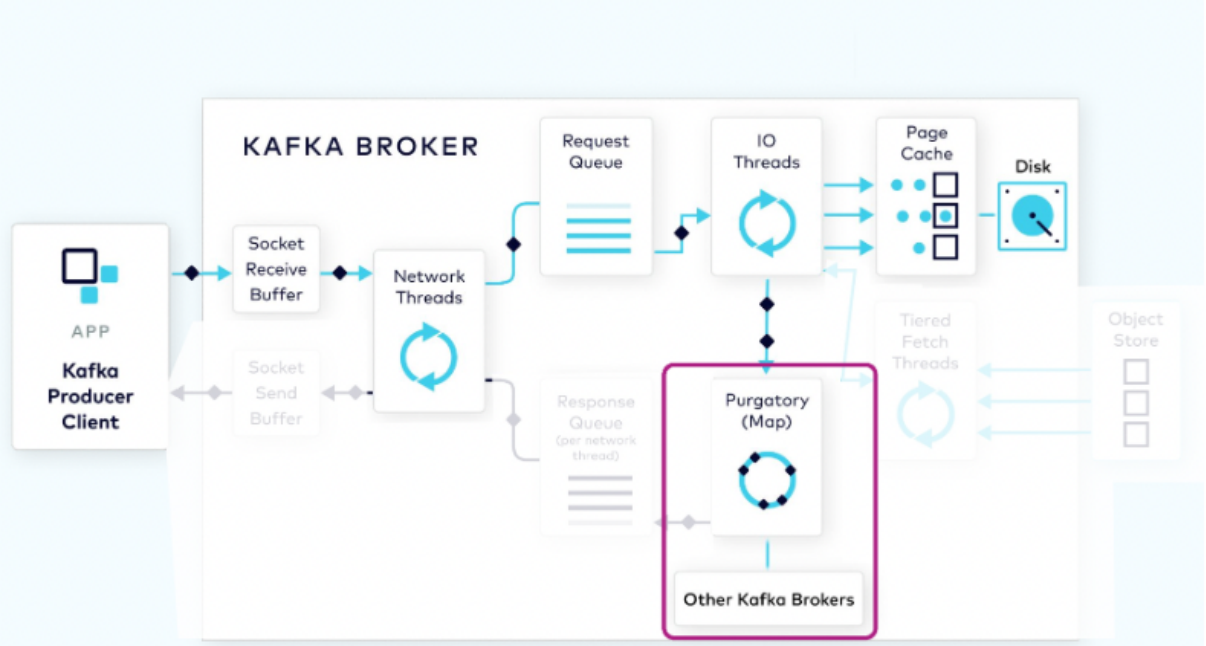
- 토픽 파티션내의 메시지들은 순서대로 저장되고, 컨슈머에 의해 순서대로 처리된다.
- 레플리카가 존재하는 한 메시지는 유실되지 않는다.

## 복제의 중요성

카프카는 저장된 메시지들을 즉시 복제해서, 메시지 유실을 방지한다. 이는 MySQL 와 같은 데이터베이스와는 다른 방식이다. MySQL 같은 데이터베이스는 fsync() 같은 시스템 콜을 이용해서 데이터를 직접 디스크에 저장한다. 즉 데이터베이스는 이 방법으로 데이터 유실을 방지한다.

그러나 카프카는 대량의 데이터 처리를 위해서 바로 디스크에 저장하지 않는다. Page Cache 라는 커널 메모리 공간에서만 데이터를 쓰고, 이후에 시간이 지나면 데이터는 디스크로 반영된다. 그러므로 디스크에 반영되기 전에 카프카 브로커가 다운된다면 메시지는 유실될 수 있다.

카프카에서는 이 문제를 복제를 통해 해결한다. 데이터를 다른 브로커의 Page Cache 에 복제 함으로써 브로커가 다운되더라도 데이터가 유실되지 않도록 한다.




- 출처: Confluent Kafka Architecture

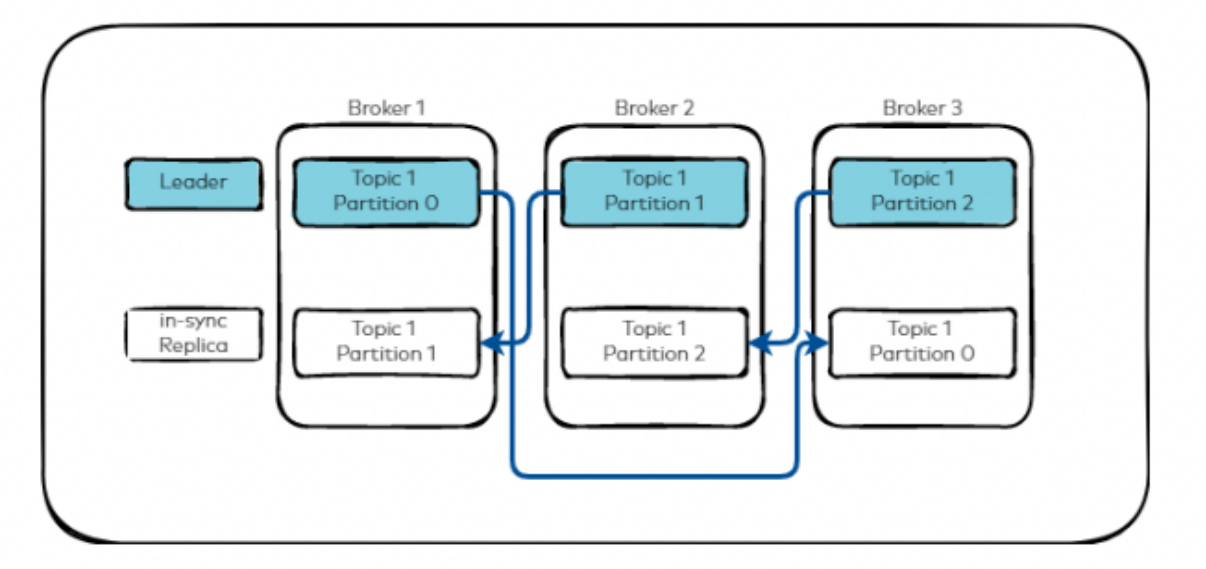
## 신뢰성 있게 브로커를 사용하는 방법

브로커의 주요 역할은 저장된 메시지들을 안전하게 보관해서 유실되지 않도록 하는 것이다. 이와 같은 신뢰성을 주기 위한 설정으로는 다음과 같다:

- 토픽 파티션의 복제 값을 정하는 것
- 최소 In-Sync 레플리카 수를 유지하는 것

### 토픽 파티션의 복제 수를 올바른 값으로 정하기

메시지들은 토픽 파티션 내에 저장된다. 만약에 토픽 파티션을 단일 복제로만 운영할 경우, 해당 브로커 장비가 손상되면 데이터는 사라질 위험이 있다. 그러므로 토픽 파티션의 복제 수를 늘려서 데이터의 가용성과 신뢰성을 높이는 것이 중요하다.




- 출처: Confluent Kafka Reference

`replication.factor` 설정 값을 통해서, 수동으로 토픽을 생성할 때 파티션 복제 수를 지정할 수 있고, `default.replication.factor` 설정을 통해 자동으로 토픽이 생성될 때 파티션 복제 수를 지정할 수 있다.

Note: 토픽 파티션의 복제 수가 너무 많은 것은 바람직하지 않다. 복제 수가 많다면 그만큼 데이터를 더 많이 저장하니까 디스크 사용량이 늘어나고, 복제를 위해 더 많이 다른 브로커들과 통신하니 네트워크 대역폭 사용량도 늘어난다. 이러한 이유로 이 값은 브로커의 노드 수에 맞춰서 값을 정하는 것이 좋다.

## 최소 In-Sync 레플리카 수를 유지하기

먼저, In-Sync 레플리카는 현재 복제가 가능한 토픽 파티션을 의미한다. 알 수 없는 이유로 현재 복제가 불가능한 토픽 파티션은 Out-Sync 레플리카라고 한다.

메시지는 리더 토픽 파티션에 기록된 후 In-Sync 레플리카에 복제되면 메시지 저장이 성공했다는 응답이 반환된다.

만약 최소 In-Sync 레플리카 값을 설정하지 않아서 복제 없이도 응답이 반환되도록 설정되어 있다면  데이터 유실 가능성이 있으므로, 이 값을 설정하는 것이 중요하다. 이는 `min.insync.replicas` 를 통해 설정할 수 있다.

Note: 이 값은 고가용성(High Availability) 과도 관련이 깊다. 만약 이 값을 `replication.factor` 와 동일하게 설정하면 가용성이 떨어지므로, 브로커 수를 고려해 적절한 값으로 설정하는 것이 중요하다.

## 브로커 Durability 와 관련된 설정

`log.retention.hours` :

- 메시지를 얼마나 오랫동안 보관할 지 결정하는 설정이다.
- 기본값은 168시간 (= 1주일) 이다.

`log.retention.bytes`

- `log.retention.hours` 와 유사한 의미를 가지며, 이 값의 크기에 도달할 때까지 메시지를 보관한다.
- 기본값으로 설정된 `-1` 은 로그 파일의 크기 제한을 없앤다는 의미다.
- `log.retention.hours` 와 함께 사용할 경우 혼란을 줄 수 있으므로 일반적으로 하나의 설정만 사용하는 것이 권장된다.

`log.segment.bytes`

- 기본적으로 토픽 파티션안의 메시지들은 하나의 거대한 로그 파일로 관리되지 않고 여러개의 세그먼트로 관리된다. 이 설정은 세그먼트의 최대 크기를 결정하며, 기본값으로는 1GB이다.
- 로그 세그먼트 파일은 최대 크기에 도달하지 않는다면 닫히지 않으며. 닫히지 않는 파일은 기본적으로 삭제되지 않는다는 사실을 알아야한다. 즉 하루에 100MB 씩 메시지가 쌓이고, `log.segment.bytes` 설정을 기본값으로 사용하고, `log.retention.hours` 값이 일주일 설정이라면 최대 17일동안 메시지는 보관될 수 있는 것이다.

`message.max.bytes`

- 카프카 브로커에 기록할 수 있는 최대 메시지 크기를 지정하는 설정이다. 즉 이 값보다 큰 메시지는 저장되지 않고 거절당한다.
- 기본값은 1MB 이다.

- 이 설정은 컨슈머 설정인 `fetch.message.max.bytes` 와 일치시키는 것이 좋다. `fetch.message.max.bytes` 값은 컨슈머에서 가지고 올 수 있는 최대 메시지 크기를 의미하는데, 만약 `message.max.bytes` 가 `fetch.message.max.bytes` 보다 더 크다면 컨슈머는 메시지를 읽지 못하는 상황이 발생할 수 있다.

## 신뢰성 있게 프로듀서를 사용하는 방법

프로듀서의 주요 임무는 브로커에 메시지를 잘 전달하는 것이다. 이와 관련된 설정으로는 다음과 같다.

- 메시지가 브로커에 복제될 때까지 대기하는 것
- 메시지 전송이 실패했다면 재시도 하는 것.
- (Optional) 메시지 순서를 보장하는 것.
- (Optional) 메시지 중복을 없애는 것.

### 메시지가 브로커에 복제될 때까지 기다리기

메시지가 브로커에 복제될 때까지 기다리는 옵션은 프로듀서의 `acks` 설정과 관련이 있다.

- `acks=0` 으로 설정하면, 프로듀서는 메시지 전송 후 응답을 기다리지 않는다.
- `acks=1` 으로 설정하면, 프로듀서는 리더 토픽 파티션에서만 메시지가 기록된 후 응답을 받는다.
- `acks=all` 로 설정하면, 프로듀서는 `min.insync.replicas` 값 만큼의 복제 파티션에 메시지가 기록될 때까지 기다린 후 응답을 받는다. 이는 메시지 유실을 최소하하는 가장 신뢰성 높은 옵션이다.

그러므로 메시지 유실을 방지하기 위해 `acks=all` 로 설정하는 것이 중요하다.

Note: 프로듀서는 메시지를 전달한 후 응답을 기다리는 동기식 처리 방법과, 응답을 기다리지 않고 응답이 왔을 때 콜백을 실행하도록 하는 비동기석 처리 방법이 있다. 동기식 처리 방법은 직관적이고 실패에 대한 핸들링이 쉽긴 하나 처리량이 정말 낮다는 단점이 있기 때문에 가능하다면 비동기식 처리 방법이 좋다.

## 메시지 전송이 실패했다면 재시도 하기

카프카에서는 메시지 전송 실패 시 내부적으로 재시도를 자동으로 수행한다. 재시도 횟수는 기본적으로 `Int.MAX` 값 만큼 수행하며, `deliver.timeout.ms` 설정에 따라 정해진 시간 동안만 재시도가 이뤄진다.

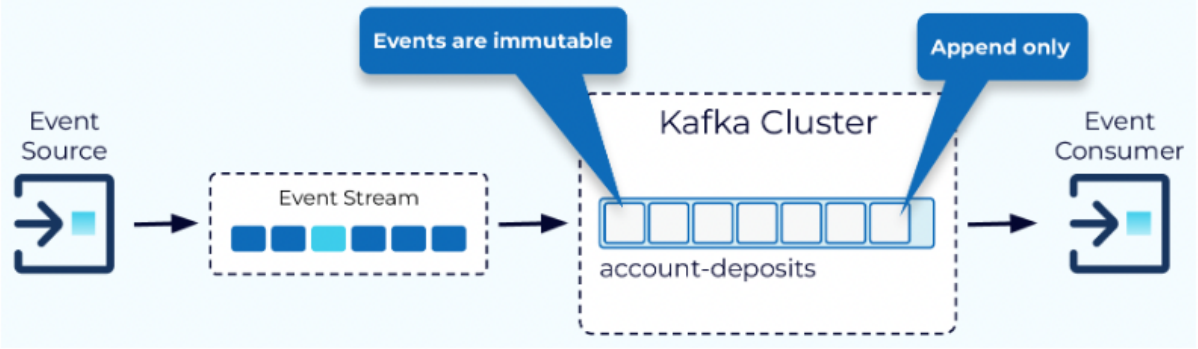
- `deliver.timeout.ms` 값은 기본적으로 2분이다.

Note: 프로듀서 내부에 재시도 매커니즘이 내장되어 있기 때문에 어플리케이션 레벨에서 별도로 재시도 로직을 구현하지 않는 것이 중요하다. 어플리케이션에서 추가적인 재시도를 할 경우에는 메시지 중복이 발생할 수 있다.

## (Optional) 메시지 순서를 보장하고 싶다면?

카프카의 토픽 파티션은 로그 파일에 append only 로 데이터를 저장하기 때문에 파티션에 순서대로 메시지를 저장하면 메시지 순서는 보장된다. 즉 토픽 파티션마다 처리 순서가 보장된다.

- 메시지를 원하는 파티션으로 보내고 싶다면 Key 값을 정하면 된다.




출처: Confluent Kafka Architecture

그러나 프로듀서의 재시도 매커니즘으로 인해 메시지 순서가 바뀔 수 있다. 프로듀서에서는 원하는 파티션으로 메시지를 순차적으로 보내지만, 이후에 메시지 전달에 실패했다고 판단하면 해당 메시지는 재시도를 통해서 다시 전송되므로, 이 과정에서 순서가 뒤바뀔 수 있다.

- 프로듀서는 `max.in.flight.requests.per.connection` 설정 값만큼 한 번에 메시지를 보내니까 메시지 전송 실패 시, 재시도 과정에서 메시지 순서는 변경될 수 있는 것이다.

이러한 문제를 방지하고자 한다면 `enable.idempotency=true` 로 설정해서 멱등성 프로듀서를 사용하면 된다. 멱등성 프로듀서는 메시지에 프로듀서를 식별하는 `Producer Id` 와 메시지 순서를 식별하는 `Sequence Number` 를 부여한다. 이 정보를 통해 브로커 단에서 메시지 순서를 보장해주고, 이전에 받은 메시지라고 판단되면 이를 버림으로써 중복을 방지한다.

### 다중 프로듀서를 쓴다면 메시지 순서 보장이 어렵다.

다중 프로듀서를 사용할 경우, 각각의 프로듀서는 멱등성을 통해 자신이 전송하는 메시지 순서를 보장할 수 있지만, 여러 프로듀서가 동일 파티션에 메시지를 경쟁적으로 전송하는 상황에서 전체 메시지 순서를 보장하기는 어렵다. 이런 경우, 메시지의 전역 순서를 보장하기 위해 글로벌 메시지 ID 를 사용하거나, 각 프로듀서 어플리케이션마다 하나의 토픽 파티션을 담당하도록 파티셔닝 해야한다.

### 멱등성 프로듀서는 중복 메시지가 발생할 수 있다.

멱등성 프로듀서를 사용하여 메시지를 전송하는 도중 프로듀서 어플리케이션이 크래쉬 나서 브로커로부터 메시지 전달 성공 응답을 받지 못했지만 실제로 메시지는 기록된 상황에서 재부팅 후 메시지를 다시 전송하게 되면 메시지 중복이 발생할 수 있다.

이는 멱등성 프로듀서의 Producer Id 특성으로 인해 메시지 중복을 방지하는 매커니즘과 관련있는데, 프로듀서 어플리케이션이 재부팅 된다면 동일한 Producer Id 값을 유지하지 못할 수도 있기 때문이다.

그러므로 이러한 문제를 완전히 방지하고자 한다면, 멱등성 프로듀서가 아닌 트랜잭셔널 프로듀서로 업그레이드 해서 사용하는 것이 좋다. 트랜잭셔널 프로듀서는 항상 동일한 Producer Id 를 유지하며, Epoch Number 를 통해 죽은 줄 알았던 이전 프로듀서 어플리케이션을 무효화 함으로써 좀비 어플리케이션 문제까지 해결해서 메시지 중복을 방지한다.

## 신뢰성 있게 컨슈머를 사용하는 방법

컨슈머의 주요 임무는 어느 메시지까지 읽었는지, 어디서부터 메시지 처리를 시작해야 하는지를 정확히 추적하는 것이다. 이를 통해 메시지 처리 과정에서 발생할 수 있는 누락을 방지할 수 있다.

컨슈머를 신뢰성 있게 사용하는 설정은 다음과 같다.

- `group.id` 를 통한 컨슈머 그룹 설정
- `auto.offset.reset` 설정
- 명시적으로 오프셋 커밋하기
- Dead Letter Queue 설정

## group.id 를 통한 컨슈머 그룹 설정

컨슈머는 컨슈머 그룹 별로 독립적으로 메시지를 처리하기 때문에 메시지 처리 목적이 서로 다르다면 각 목적에 맞는 별도의 컨슈머 그룹을 설정해야한다. 예를 들어, 다양한 처리 목적을 가진 여러 컨슈머가 동일한 컨슈머 그룹으로 속해서 토픽을 구독한다면, 각 컨슈머들은 토픽 내 전체 메시지를 처리하는 것이 아니라 각 파티션에서 일부 메시지만 처리하게 된다.

이는 카프카는 토픽 파티션을 컨슈머 그룹 내의 컨슈머들 사이에서 분배하는 과정 때문이다. 이로 인해, 하나의 컨슈머 그룹 내에 각 컨슈머는 토픽의 파티션들 중 일부를 할당받아 그 파티션의 메시지만을 처리하게 된다. 따라서, 만약 서로 다른 목적으로 메시지를 전체 처리해야 한다면, 각기 다른 컨슈머 그룹을 설정하여 메시지를 각각의 목적에 맞게 처리할 수 있도록 해야 한다.

## auto.offset.reset 설정

컨슈머가 브로커에 없는 오프셋을 요청하거나, 오프셋 데이터가 만료되어 사라진 경우 컨슈머 오프셋을 어떻게 리셋할지 결정하는 설정이다. 이 설정은 특히 컨슈머 그룹이 처음 토픽을 구독할 때, 또는 지정된 오프셋이 더 이상 존재하지 않을 때 사용된다.

- 오프셋은 consumer_offsets 이라는 토픽에 저장되며, 컨슈머 그룹에 활성화된 컨슈머가 하나도 없는 경우에 `offset.retention.minutes` 설정에 따라 이 기간이 지나면 해당 오프셋 데이터가  삭제된다.

`auto.offset.reset` 으로 설정할 수 있는 값은 다음과 같다:

- **“earliest”**: 기본값으로 파티션에 저장된 메시지 중 첫번째 메시지부터 읽도록 한다.
- **“latest”**: 파티션에 저장된 메시지 중 제일 마지막 메시지부터 읽도록 한다.
- **“none”**: 오프셋이 없다면 에러를 발생시키고 시작을 거부한다.

메시지 처리 과정에 누락이 없도록 하려면 `auto.offset.reset=earliest` 값으로 설정하는 것이 좋다. 물론 이 경우에는 메시지 처리 중복을 고려해야한다.

반면 최신 메시지 처리에만 관심이 있다면, `auto.offset.reset=latest` 설정을 사용할 수 있다. 설정을 선택할 때는 애플리케이션의 요구 사항과 메시지 처리 방식을 고려해야 한다.

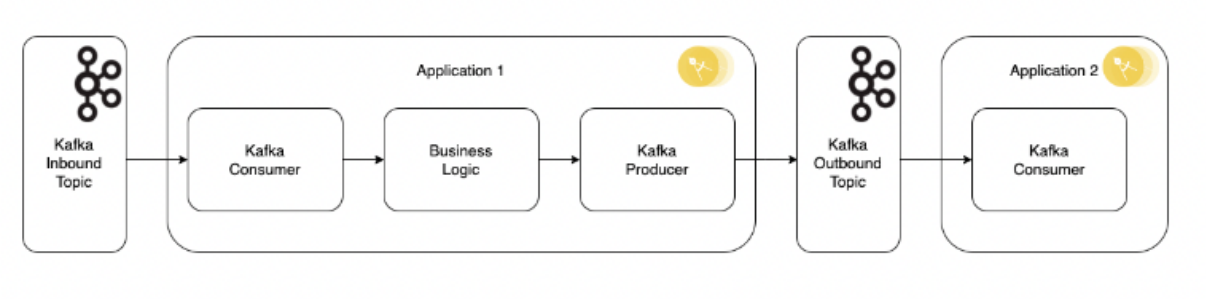
## 명시적으로 오프셋 커밋하기

오프셋 커밋은 컨슈머가 메시지를 어디까지 처리했는지 카프카에 알려주는 역할을 한다. 이는 처리된 메시지의 위치를 기록함으로써 컨슈머가 중단된 후 재시작 할 때 이미 처리한 메시지를 건너뛰고, 처리하지 않은 메시지부터 처리를 시작할 수 있게 한다. 따라서 **메시지를 처리한 후에 오프셋을 커밋하는 매커니즘**이 중요하다. 만약 메시지 처리 전에 먼저 커밋을 한다면, 메시지 처리 중에 문제가 발생해서 중단되었을 때 메시지 처리에 누락이 발생할 위험이 있다.

오프셋 커밋 방법은 자동 커밋과 수동 커밋이 있다. `enable.auto.commit` 설정에 따라 이를 결정할 수 있다:

- **자동 커밋**: `enable.auto.commit=true`로 설정할 경우, 카프카는 주기적으로 오프셋을 자동으로 커밋한다. 이 방법은 구현이 간단하며, 개발자가 오프셋 커밋 시점을 직접 관리할 필요가 없다. 또 자동 커밋은 메시지 처리에 누락이 생길 수 있다고 편견을 가질 수 있는데, 그렇지 않다. 자동 커밋하는 시점은 메시지를 처리하고 다음 메시지를 가지고 올 때 수행하므로 안전한 설정이다. 하지만, 자동 커밋은 처리 중인 메시지에 대한 커밋 타이밍을 세밀하게 제어하기 어렵기 때문에, 메시지 중복 처리가 발생할 수 있다.
- **수동 커밋**: `enable.auto.commit=false`로 설정하고, 애플리케이션이 메시지 처리 후 명시적으로 오프셋을 커밋하는 방식이다. 이 방식은 개발자가 오프셋 커밋의 정확한 시점을 제어할 수 있으며, 메시지 처리를 보다 정확히 관리할 수 있어서 중복 처리를 예방할 수 있다.

추가로, 컨슈머가 Read-Process-Produce 패턴으로 처리하는 경우에도 커밋 타이밍이 중요하다. 올바른 커밋 시점은 처리가 완료되고, 메시지가 새로운 토픽에 성공적으로 전달된 후에 오프셋을 커밋하는 것이다. 또 다른 방법으로는, 트랜잭셔널 프로듀서를 사용하여 커밋과 메시지 전달을 원자적(atomic)으로 관리할 수 있다.




출처: Kafka Exactly-Once Delivery

## Dead Letter Queue 설정

컨슈머가 메시지를 가져와서 처리하다가 실패하는 경우엔 어떻게 대응해야할까? 일시적인 문제일 수 있으니 재시도를 해보는 것이 좋을 것 같다. 그러나 모든 문제가 재시도로 해결되지는 않는다. 재시도 후에도 문제가 지속된다면 계속해서 재시도를 하는 것은 시스템에 부담을 주고 전체 메시지 처리 성능에 영향을 줄 수 있기 때문에 실패한 메시지는 DLQ 로 분리하는 것이 좋다. 이를 통해 메시지 처리를 원활하게 진행할 수 있고, 실패한 메시지를 유실되지 않도록 하면서, 필요하다면 실패한 메시지에 대한 처리를 재개 할 수 있다.