- 출처 : 실전 아파치 카프카

분산 메시징 구조

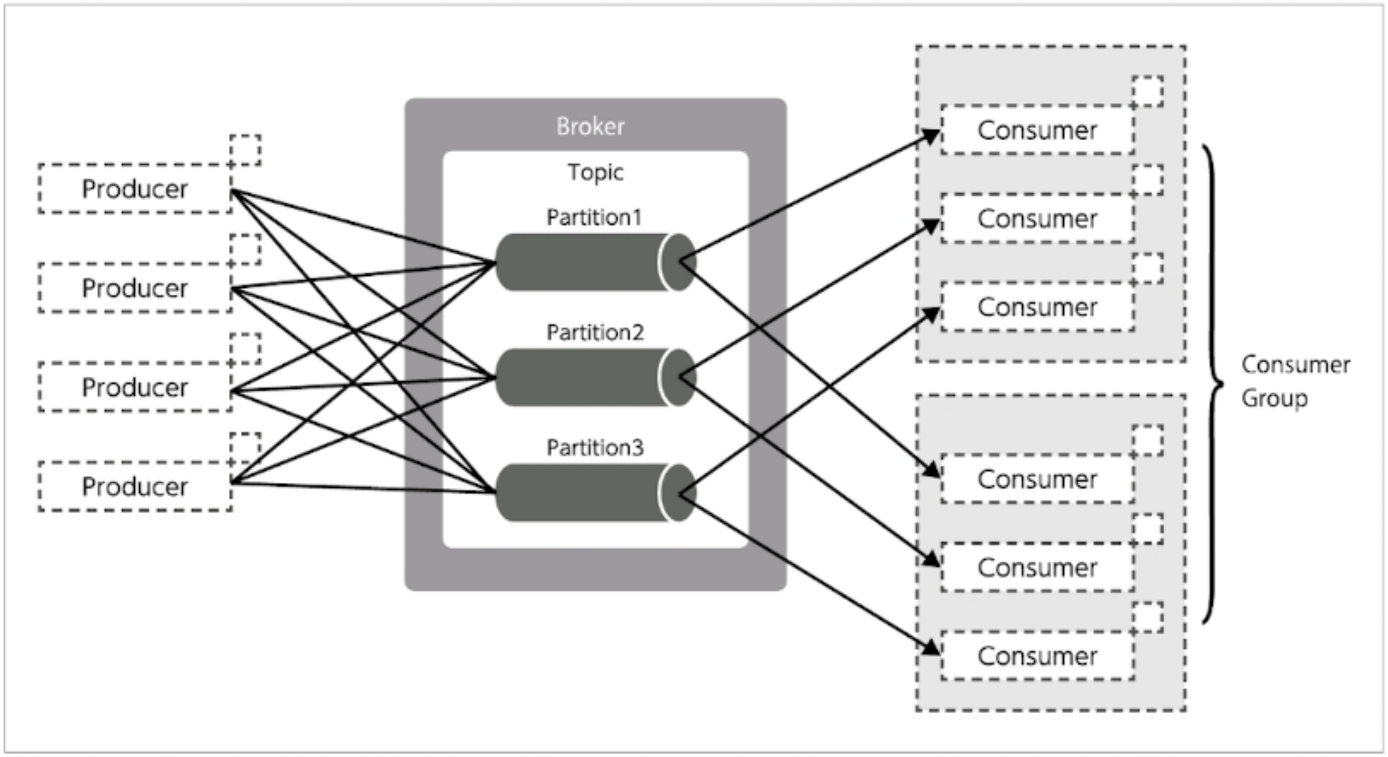


그림 2-3 분산 메시지 관리 이미지

파티션

- 토픽에 대한 대량의 메시지 입출력을 지원하기 위해, 브로커상의 데이터를 읽고 쓰는 것은 파티션(partition)이라는 단위로 분할되어 있음
- 토픽을 구성하는 파티션은 브로커 클러스터 안에 분산 배치되어 프로듀서에서의 메시지 수신, 컨슈머로의 배달을 분산해서 실시함으로써 하나의 토픽에 대한 대규모 데이터 수신과 전달을 지원
- 각 파티션을 브로커에 어떻게 배치되는지에 대한 정보는 브로커 측에 유지
- 프로듀서 API/컨슈머 API가 파티션들을 은폐해서 통신하기 때문에 프로듀서/컨슈머에서는 토픽만을 지정하고, 구현 시에 송신처 파티션을 의식할 필요가 없음

컨슈머 그룹

- 카프카는 분산 스트림 처리도 고려하여 설계되어 있음
- 단일 애플리케이션 안에서 여러 컨슈머가 단일 토픽이나 여러 파티션에서 메시지를 취득하는 방법으로 컨슈머 그룹(Consumer Group)이라는 개념이 존재
- 카프카 클러스터 전체에서 글로벌 ID를 컨슈머 그룹 전체에서 공유하고 여러 컨슈머는 자신이 소속한 컨슈머 그룹을 식별해, 읽어들일 파티션을 분류하고 재시도를 제어

오프셋

- 각 파티션에서 수신한 메시지에는 각각 일련번호가 부여되어 있어 파티션 단위로 메시지 위치를 나타내는 오프셋(offset)이라는 관리 정보를 이용해 컨슈머가 취득하는 메시지의 범위 및 재시도를 제어
- 제어에 사용되는 오프셋
  - Log-End-Offset(LEO) : 파티션 데이터의 끝을 나타냄
  - Current Offset : 컨슈머가 어디까지 메시지를 읽었는가를 나타냄
  - Commit Offset : 컨슈머가 어디까지 커밋했는지를 나타냄

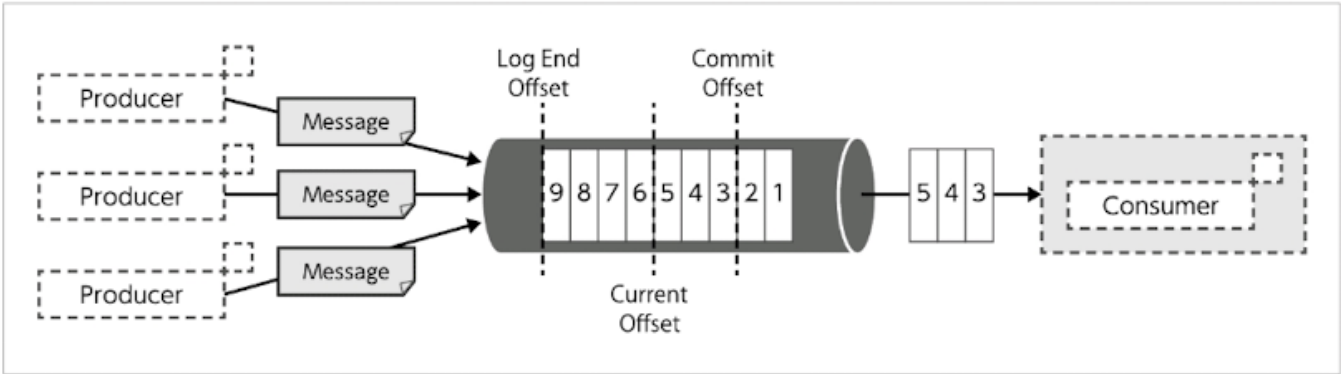


그림 2-4 오프셋 관리

- LEO는 브로커에 의해 파티션에 관한 정보로 관리 및 업데이트 됨
- Commit Offset은 컨슈머로부터 여기까지의 오프셋을 처리했다라는 것을 확인하는 오프셋 커밋 요청을 계기로 업데이트 됨

메시지 송신

- 송신은 하나 하나의 메시지 단위로 송수신하는 것은 아님
- 송수신 처리량을 높이기 위해 어느 정도 메시지를 축적하여 배치 처리로 송수신하는 기능 또한 제공

프로듀서의 메시지 송신

- 프로듀서가 토픽의 파티션에 메시지를 송신할 대 버퍼 기능처럼 프로듀서의 메모리를 이용하여 일정량을 축적 후 송신할 수 있음
- 데이터의 송신에 대해서는 지정한 크기까지 메시지가 축적되거나, 지정한 대기 시간에 도달하는 것 중 하나를 트리거로 전송

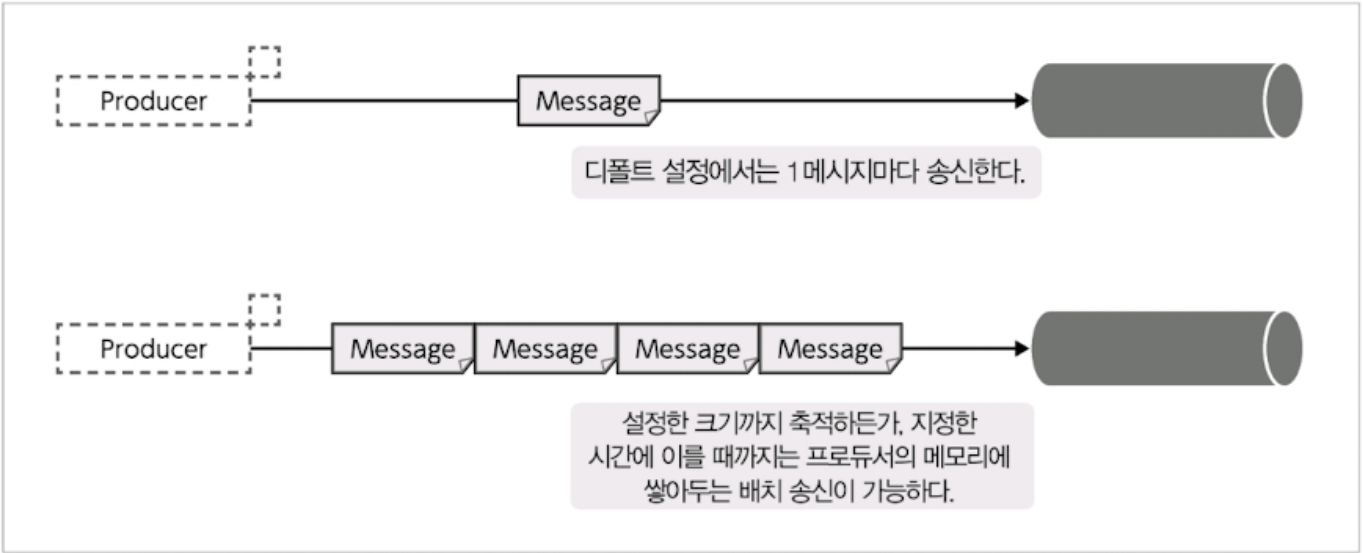


그림 2-5 프로듀서의 메시지 송신

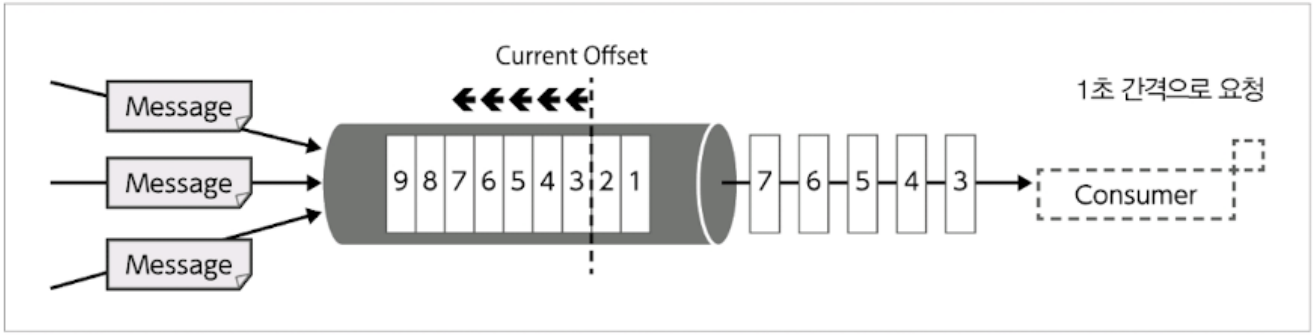
- 기본 설정으로 하나의 메시지는 1회 송신이지만, 수 바이트에서 수십 바이트의 작은 메시지를 대량으로 브로커에 송신하는 상황에서는 네트워크의 지연이 처리량에 영향을 주는 경우도 있어 메시지를 배치로 송신함으로 처리량을 향상시킨다
- 크키가 큰 텍스트 파일과 로그 파일에 포함된 각 레코드를 한 행 한행 브로커에 송신하는 경우에도 여러 행을 모아서 배치 송신함으로서 처리량을 향상시키는 효과를 기대할 수 있음

컨슈머의 메시지 취득

- 컨슈머는 취득 대상의 토픽과 파티션에 대해 Current Offset으로 나타나는 위치에 마지막으로 취득한 메시지부터 브로커에서 보관하는 최신 메시지까지 모아서 요청 및 취득을 실시하고, 그것을 반복함으로써 계속적으로 메시지를 취득
- 메시지의 유입 빈도가 동일한 경우 컨슈머의 브로커 요청 간격이 길수록 모인 메시지가 많아진다.

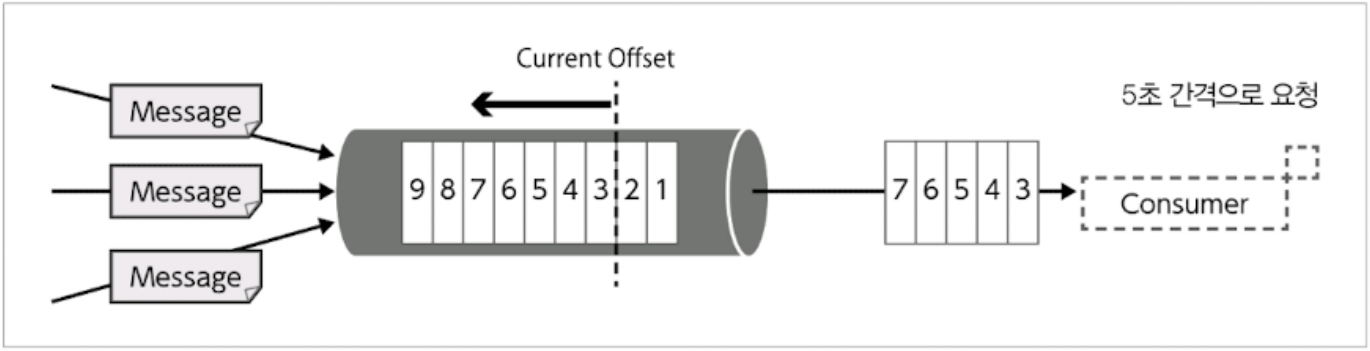
작은 범위로 요청하는 경우

- 요청으로 하나의 메시지를 취득하는 경우 하나의 메시지마다 Current Offset을 업데이트



일정 간격을 두고 요청하는 경우

- 하나의 요청으로 5개의 메시지를 취득할 경우 5개의 메시지만큼 Current Offset을 업데이트 함

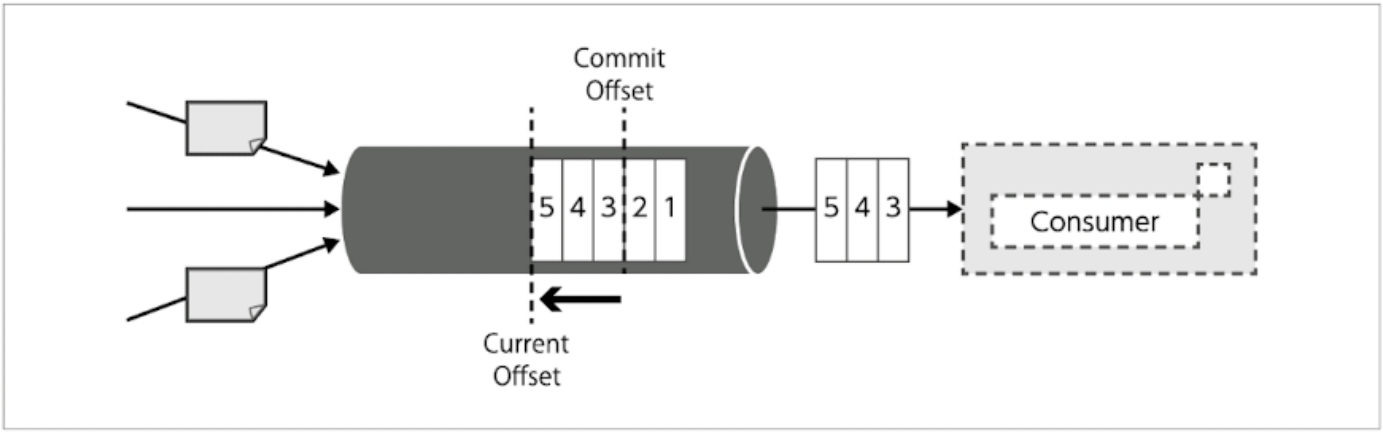


- 프로듀서, 컨슈머에서도 어느 정도 메시지를 모아 배치 처리함으로써 처리량을 향상시키는 효과는 기대할 수 있지만 프로듀서 송신과 컨슈머 수신 처리의 지연 시간은 증가함
- 취급하는 데이터와 시스템에 따라 밀리초에서 초 단위의 지연 시간도 바람직하지 않은 경우도 있음
- 그렇기 때문에 배치 처리의 간격에 대해서는 처리량과 대기 시간의 trade-off를 고려한 설계가 필요

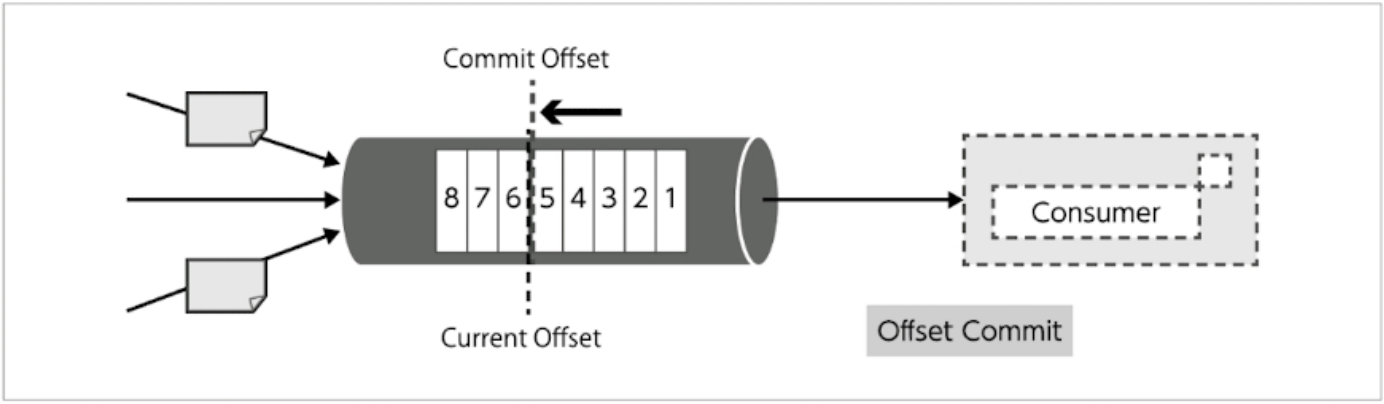
컨슈머의 롤백

- 컨슈머는 오프셋을 진행하면서 지속적으로 메시지를 취득하지만, Offset Commit의 구조를 이용해 컨슈머 처리 실패, 고장 시 롤백 메시지 재취득을 실현
- 메시지를 로드할 때 컨슈머 장애 시 재시도 흐름을 아래 그림과 같음
- 아래 예제는 계속적으로 프로듀서에서 메시지가 송신되는 상황에서 컨슈머에 의한 데이터 취득이 2회 발생하는 시나리오로 두 번째 취득에서 장애가 발생할 때의 동작을 설명

  1. Offset 2까지 취득해 Offset Commit이 끝난 단계에서 Offset 3, 4, 5의 메시지를 취득

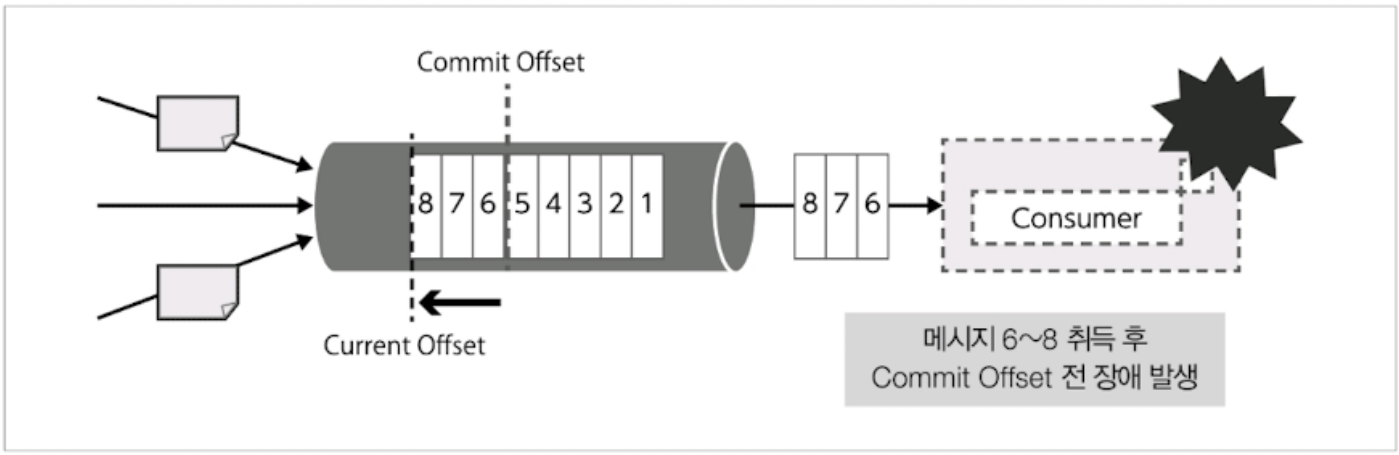


  2. 컨슈머 쪽 처리가 끝나 Offset Commit를 실행하고, Commit Offset을 5까지 진행

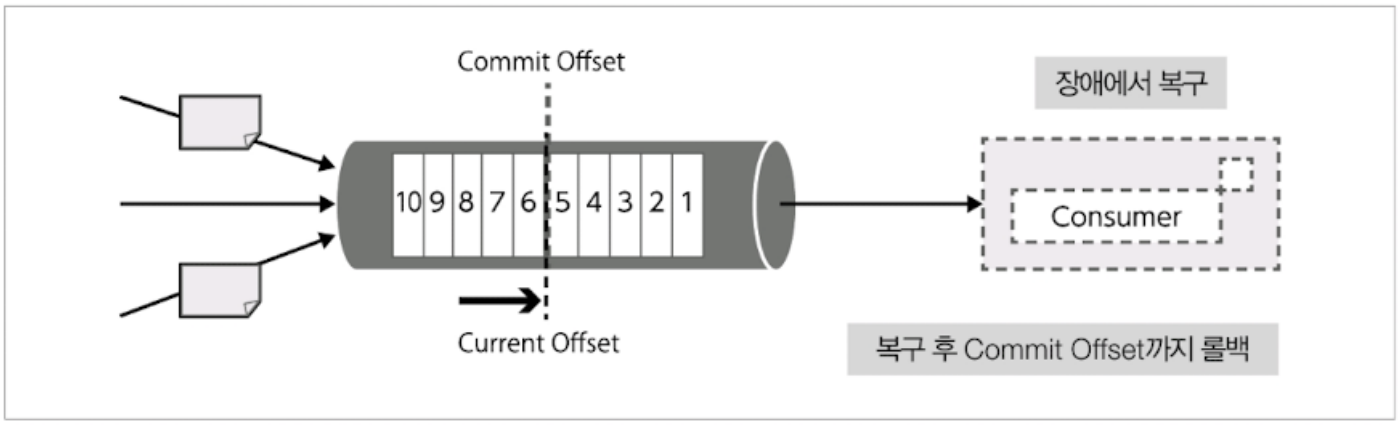


- 여기까지는 정상적인 동작의 예로 메시지를 계속 취득한다

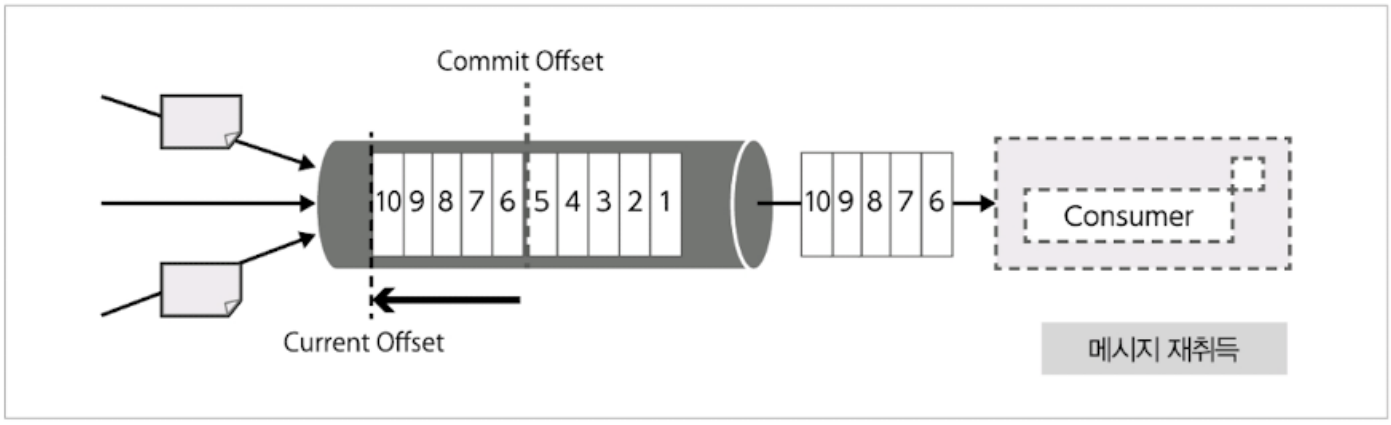
3. 컨슈머 쪽에서 처리 중 Offset Commit을 실행하기 전에 컨슈머에서 장애가 발생했음



4. 컨슈머가 장애에서 복구되면 Commit Offset부터 재개한다



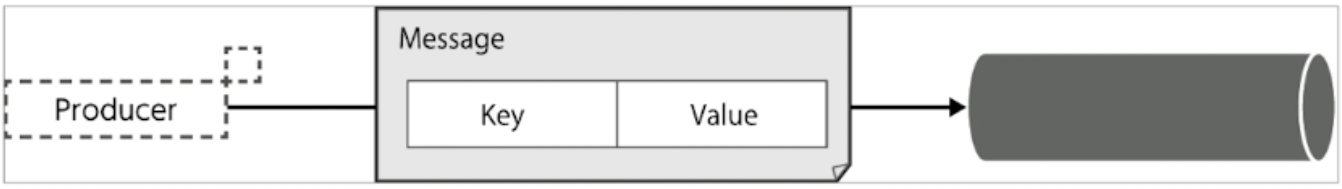
5. 메시지를 재취득한다



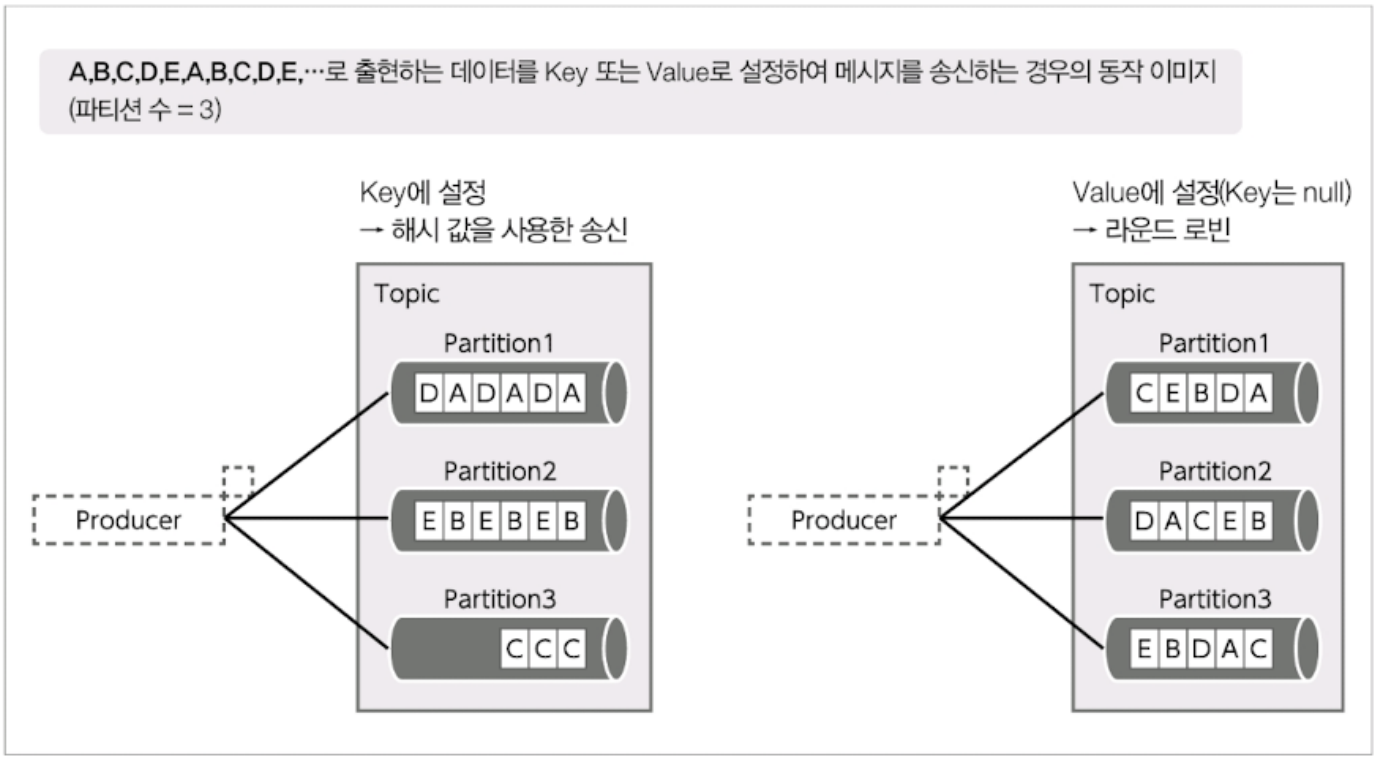
- 위의 예에서 주의해야 할 점은 Commit Offset까지 되돌아 온 오프셋 간 메시지에 대한 처리는 후속 애플리케이션에 맡긴다는 점
- 메시지를 처리 완료 상태에서 Commit Offset 업데이트 직전의 고장의 경우는 동일한 메시지가 재전송되고, 메시지 중복 처리(또는 중복 허용)가 필요
- 이 재시도는 Exactly Once(빠짐없이 중복이 없는 송신)가 아니라 At Least Once(최소 1번 이상)로 송신하는 구조
- 또한 고장의 감지, 복구에 대해서도 카프카에서 제공되는 것은 아니기 때문에 Consumer API를 이용한 애플리케이션 쪽에서의 대처가 필요함

메시지 전송 시 파티셔닝

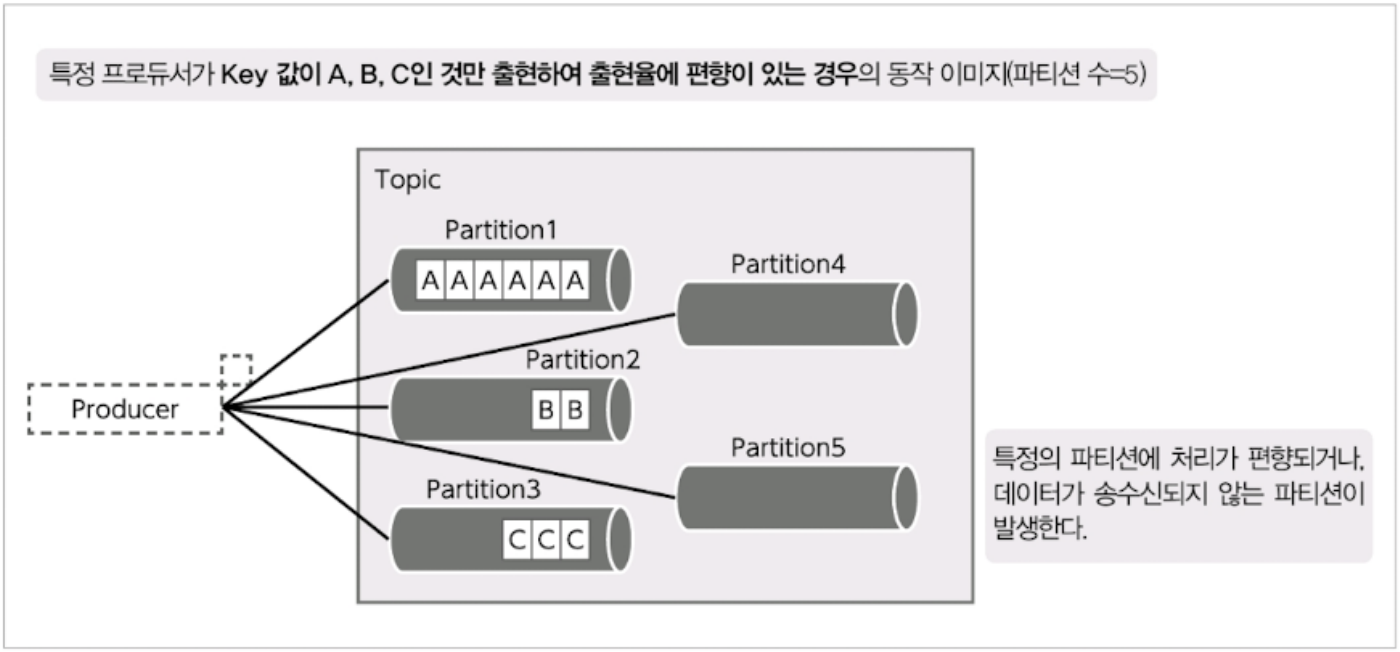
- 프로듀서에서 송신하는 메시지를 어떻게 파티션으로 보낼지 결정하는 파티셔닝(분할) 기능이 제공되고 있음
- 보내는 메시지에 포함된 Key와 Value 중 Key의 명시적인 지정 여부에 따라 다음의 두 가지 패턴 로직으로 송신됨
  1 Key의 해시 값을 사용한 송신
  - 메시지의 Key를 명시적으로 지정함으로써 Key에 따라 송신처 파티션을 결정하는 로직이 됨
  - 동일한 Key를 가진 메시지를 동일한 ID를 가진 파티션에 송신



2. 라운드 로빈에 의한 송신
- 메시지 Key를 지정하지 않고 Null로 한 경우 여러 파티션으로의 메시지 송신을 라운드 로빈 방식으로 실행
- 아래 그림은 라운드 로빈과 해시에 의한 파티셔닝 그림예시



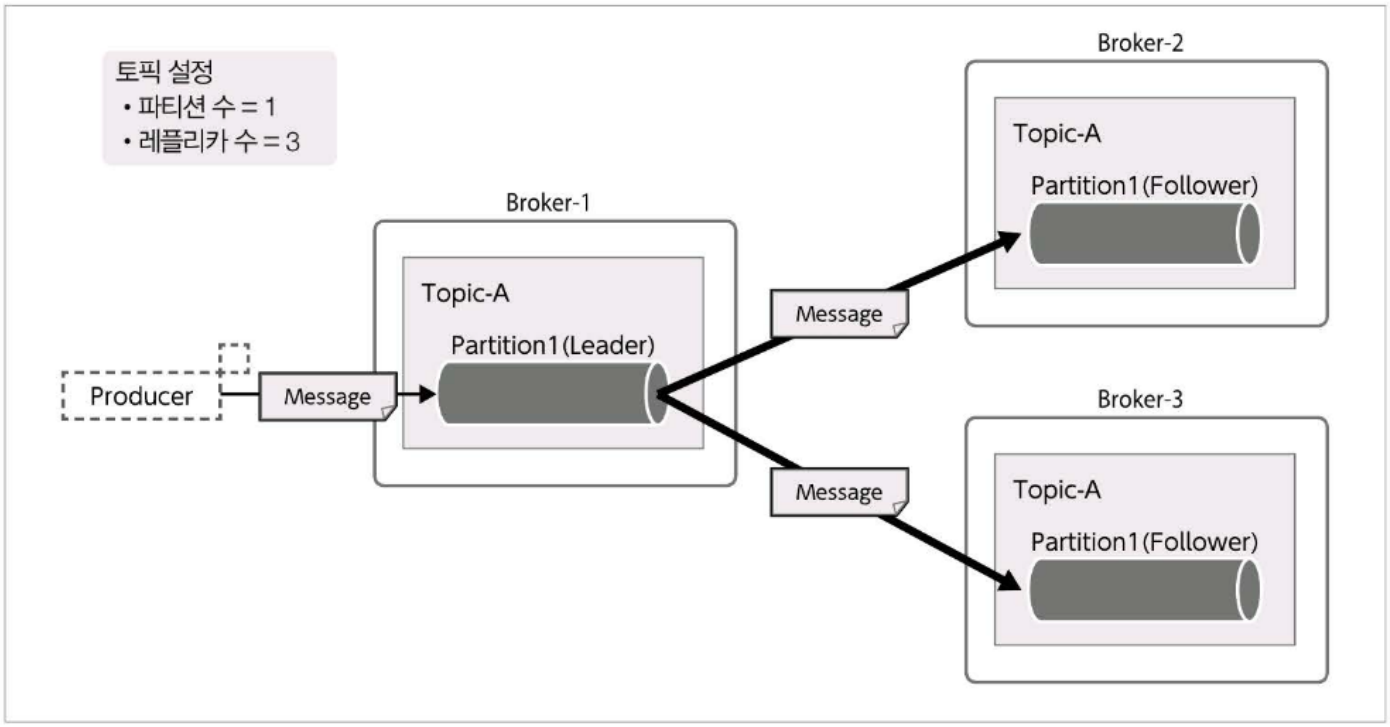
- 예를 들어 웹의 액세스 로그를 송신할 때 발신지 IP 주소에 따라 파티션별로 나누어 던지는 경우는 로그 안에서 '발신지 IP 주소'를 Key로 설정하여 전송함으로써 가능
- 해시에 의한 파티셔닝을 이용함으로써 동일한 Key를 가진 메시지는 동일한 컨슈머에서 취득하여 처리하는 식으로 제어
- 그러나 파티셔닝을 이용하는 경우는 데이터 편차에 따른 파티션의 편향에도 주의를 기울여야 함
- 극단적인 경우 준비한 파티션 수에 대해 출현하는 Key의 종류가 충분하지 않은 때는 파티션에 편향이 발생하여 리소스를 부분적으로 사용할 수 없는 상태가 됨



- 위의 파티셔닝 로직은 카프카에서 DefaultPartitioner 클래스를 이용하여 구현
- Producer API에서 제공하는 Partitioner 인터페이스를 구현함으로써 Key와 Value 값에 따라 송신 로직을 커스텀으로 구현할 수 있음

레플리케이션

- 카프카는 메시지를 중계함과 동시에 서버가 고장 났을 때에 수신한 메시지를 잃지 않기 위해 복제(Replication) 구조를 갖고 있음



- 위의 그림은 토픽 수 = 1, 파티션 수 = 1로 되어 있음
- 파티션은 단일 또는 여러 개의 레플리카로 구성되어 토픽 단위로 레플리카 수를 지정할 수 있음
- 레플리카 중 하나는 Leader이며, 나머지는 Follwer라고 불림
- Follower는 그 이름대로 Leader로부터 메시지를 계속적으로 취득하여 복제를 유지하도록 동작

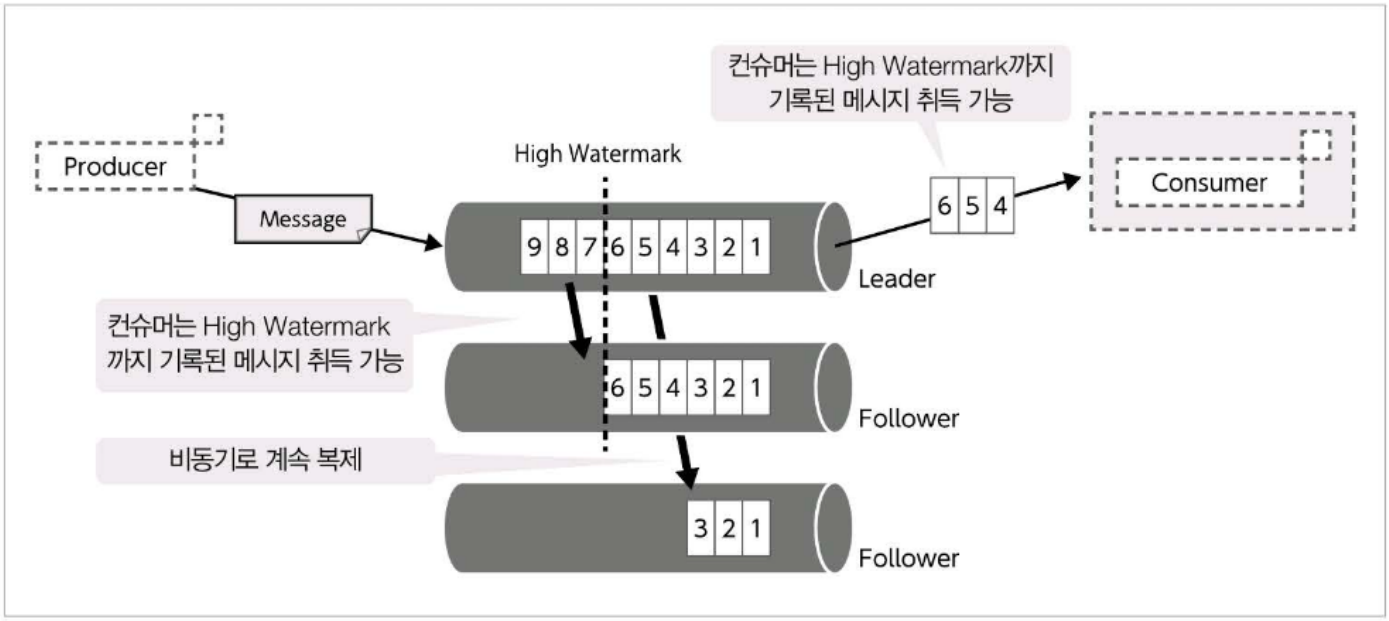
- 다만 프로듀서/컨슈머와 데이터 교환은 Leader가 담당한다

레플리카의 동기 상태

- Leader Replica의 복제 상태를 유지하고 있는 레플리카는 In-Sync Replica로 분류
- 콘솔 출력에서는 ISR로 표기
- In-Sync Replica로 되어 있지 않은 파티션을 Under Replicated Partitions이라고 부름
- 복제 수와는 독립적으로 최소 ISR 수(min.insync.replica) 설정이 가능하며, 고장 등으로 인한 일시적인 동기 지연을 허용하여 전체 읽고 쓰기를 계속하는 것이 가능

복제 완료 최신 오프셋

- 복제 사용 시 오프셋 관리에는 LEO(Log End Offset)이 사용
- 복제 완료 최신 오프셋(High Watermark)라는 개념이 존재
  - 복제가 완료된 오프셋
  - 컨슈머는 High Watermark까지 기록된 메시지를 취득할 수 있음

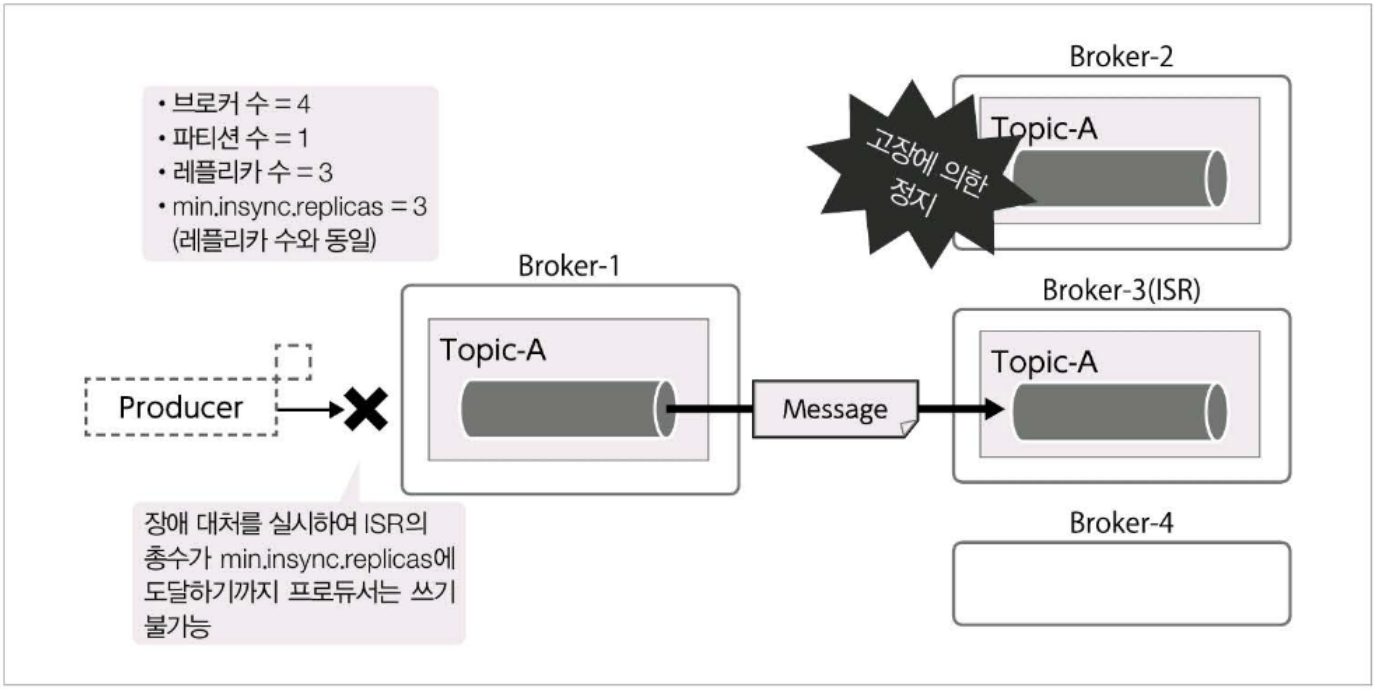


In-Sync Replica

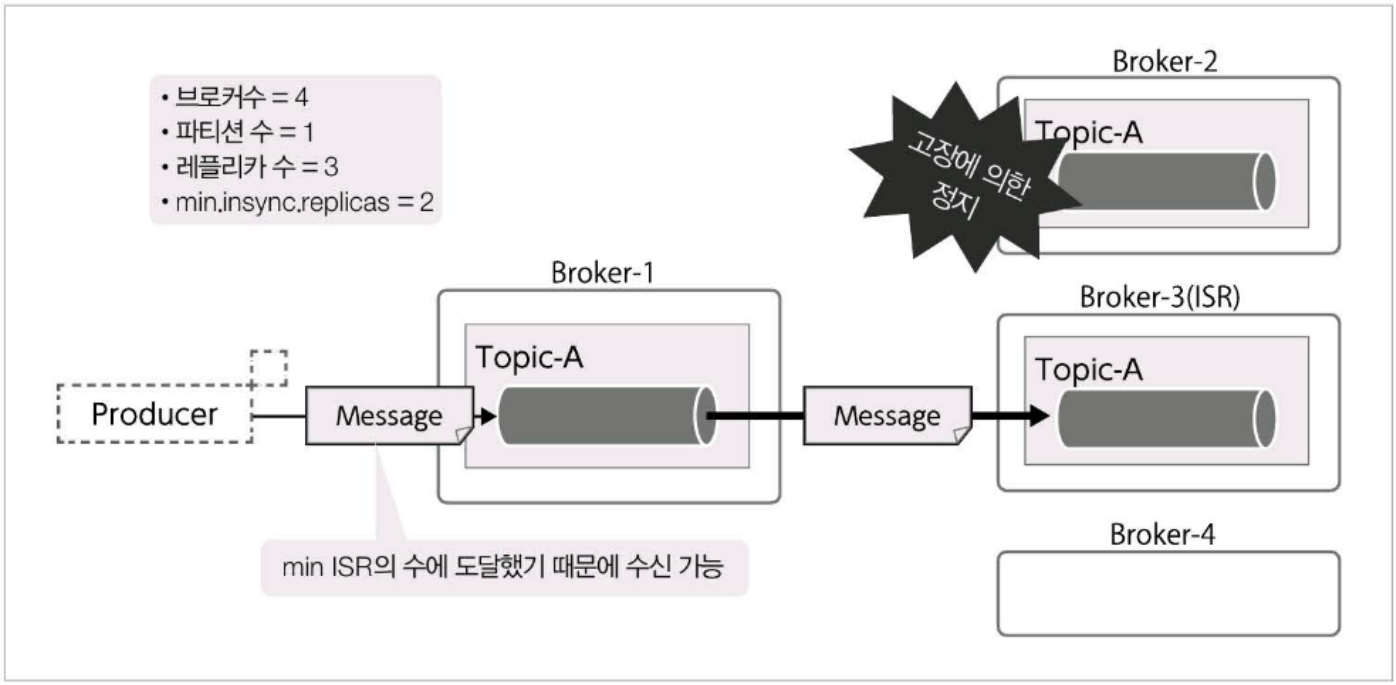
- 브로커 4대, 레플리카 수 3으로 브로커 1대가 고장나 레플리카를 하나 잃어버린 경우의 예시

1. min.insync.replicas = 3(레플리카 수와 동일), Ack=all인 경우
   - 브로커 서버가 1대 고장난 경우 프로듀서는 비정상 상태로 간주하여 잃어버린 레플리카가 ISR로 복귀할 때까지 데

이터를 쓸 수 없음



2. min.insync.replicas=2, Ack=all인 경우
   - 브로커 서버가 1대 고장난 경우에도 Ack를 반환하고 처리를 계속
   - 처리를 계속하는 점에 있어서는 1의 경우보다 나은 반면, 나중에 추가된 파티션이 복제를 완료해 ISR로 승격될 때까지 복제 수가 2가 됨
   - 복구 전에 2대가 고장난 경우는 처리 중인 메시지를 손실할 위험이 높아짐



- min.insync.replicas 설정은 서버 고장 시 데이터를 잃지 않은 것과 메시징 시스템을 포함한 전체 시스템의 처리를 계속하는 것 사이의 균형을 조정하는 항목이라고 할 수 있음

- 두 가지 경우는 어느 쪽이 뛰어나다는 것이 아니라 시스템 요구 사항과 제약 조건에 의해 결정돼야 한다는 점에 주의해야 함