분산 메시지 큐 (240131)

▼ 🤔 메시지 큐?

왜 쓰는가?

- **결합도 완화**
  - 컴포넌트 사이의 강한 결합이 사라짐
  - 각 컴포넌트를 독립적으로 갱신할 수 있음
- **규모 확장성 개선**
  - 생산자와 소비자 시스템 규모를 트래픽 부하에 맞게 독립적으로 늘릴 수 있음
  - 트래픽이 많이 몰리는 시간에는 더 많은 소비자를 추가하여 처리 용량을 늘림
- **가용성 개선**
  - 특정 컴포넌트에 장애가 발생해도 다른 컴포넌트는 큐와 상호작용하면 됨
- **성능 개선**
  - 비동기 통신이 쉬움
  - 생산자) 응답을 기다리지 않고 메시지를 보냄
  - 소비자) 읽을 메시지가 있을 때만 메시지를 소비

메시지 큐 vs 이벤트 스트리밍 플랫폼

지원하는 기능이 서로 수렴하면서 차이가 희미해지고 있다.

RabbitMQ

- 스트리밍 기능을 추가하는 옵션이 있음
  - 메시지 반복 소비 가능
  - 데이터 장기 보관 가능

Apache Pulsar

- 카프카의 경쟁자
- 분산 메시지 큐로도 사용 가능

⇒ `데이터 장기 보관`, `메시지 반복 소비` 등 부가 기능을 갖춘 메시지 큐를 설계해보자.

▼ 1️⃣ 문제 이해 및 설계 범위 확정

기본 기능

- 생산자는 메시지를 큐에 보낸다.
- 소비자는 큐에서 메시지를 꺼낼 수 있으면 꺼낸다.

질문 목록

메시지

- 형태
- 평균 크기
- 멀티미디어 지원 여부
- 반복 소비 여부
- 순서
- 전달 방식(최대/최소/정확히)

데이터

지속성 보장 여부

비기능

생산자, 소비자 지원 수

대역폭

지연 시간

요구사항 정리

기능 요구사항

- 생산자는 메시지 큐에 메시지를 보낸다.
- 소비자는 메시지 큐를 통해 메시지를 받는다.
- 메시지는 반복 수신 / 한 번만 수신이 설정 가능해야 한다.
- 오래된 이력 데이터는 삭제될 수 있다.
- 메시지 크기는 KB 수준이다.
- 메시지가 생산된 순서대로 소비자에게 전달되어야 한다.
- 메시지 전달 방식은 최소 한 번, 최대 한 번, 정확히 한 번 가운데 설정 가능해야 한다.

비기능 요구사항

- 높은 대역폭 / 낮은 전송 지연 가운데 하나를 설정으로 선택
- 규모 확장성
- 지속성 및 내구성

🤔 전통적 메시지 큐와 다른 점

전통적인 메시지 큐는 ...

- 메시지가 소비자에 전달되기 충분한 기간 동안만 메시지를 메모리에 보관한다.
- 처리 용량을 넘어선 메시지는 디스크에 보관하지만 이벤트 스트리밍 플랫폼이 감당하는 용량보다 아주 낮은 수준이다.
- 메시지 전달 순서도 보존하지 않는다.

▼ 2️⃣ 개략적 설계안 제시 및 동의 구하기

기본 기능

- 생산자는 메시지를 메시지 큐에 발행
- 소비자는 큐를 구독
- 소비자는 구독한 메시지를 소비
- 메시지 큐는 생산자와 소비자 사이 결합을 느슨하게 하는 서비스
- 메시지 큐는 생산자와 소비자의 독립적인 운영 및 규모 확장을 가능하게 하는 역할
- 생산자/소비자는 모두 클라이언트/서버 모델 관점에서 클라이언트, 메시지 큐는 서버 역할
    - 클라이언트와 서버는 네트워크를 통해 통신

메시지 모델

일대일 모델

전통적인 메시지 큐에서 흔히 발견되는 모델

- 큐에 전송된 메시지는 한 소비자만 가져갈 수 있다.
- 소비자가 메시지를 가져갔다는 사실을 큐에 알리면 해당 메시지는 큐에서 삭제된다.
- 데이터 보관을 지원하지 않는다.

지속성 계층을 포함하여 메시지를 보관하고 반복 소비될 수 있게 할 수 있지만… 이 모델은 부합하지 않는 듯

발행-구독 모델

토픽

메시지를 주제별로 정리하는 데 사용되는 개념

- 메시지 큐 서비스 전반에 고유한 이름을 가짐
- 토픽에 전달된 메시지는 토픽을 구독하는 모든 소비자에게 전달

파티션

토픽을 여러 조각으로 분할(샤딩)한 메시지의 작은 부분집합

- 메시지 큐 클러스터 내의 서버에 고르게 분산 배치
- 같은 파티션 안에서 메시지 순서가 유지됨
- 파티션 내 메시지 위치는 **오프셋**이라고 한다.
- 생산자가 보낸 메시지는 해당 토픽의 파티션 중 하나로 보내진다.
- 메시지에는 키를 붙일 수 있는데, 같은 키를 가진 모든 메시지는 같은 파티션으로 보내진다.
    - 키가 없는 메시지는 무작위로 선택된 파티션으로 전송된다.

브로커

파티션을 유지하는 서버

- 파티션을 브로커에 분산하는 것이 높은 규모 확장성을 달성하는 비결

소비자 그룹

토픽을 구독하는 소비자는 하나 이상의 파티션에서 데이터를 가져온다.

토픽을 구독하는 소비자가 여럿이면 각 구독자는 해당 토픽을 구성하는 파티션의 일부를 담당한다.

이 때 소비자들을 해당 토픽의 소비자 그룹이라고 부른다.

⚠️ 문제

데이터를 병렬로 읽으면 대역폭 측면에서는 좋지만 같은 파티션 내에 메시지를 순서대로 소비할 수 없다.

→ 어떤 파티션의 메시지는 한 그룹 안에서 한 소비자만 읽을 수 있도록 제약사항을 추가하자.

→ `그룹 내 소비자 수 > 구독하는 토픽의 파티션 수`면 어떤 소비자는 해당 토픽에서 데이터를 못 읽는 문제 발생!

→ 결국 일대일 모델로 수렴하는 걸 방지하기 위해 미리 충분한 파티션을 할당하고 **처리 용량을 늘리려면 소비자를 더 추가하자!**

개략적 설계안

클라이언트

- 생산자: 메시지를 특정 토픽으로 보낸다.
- 소비자 그룹: 토픽을 구독하고 메시지를 소비한다.

핵심 서비스

- 브로커
  - 파티션들을 유지한다.
  - 하나의 파티션은 특정 토픽에 대한 메시지의 부분 집합을 유지한다.
- 조정 서비스
  - 서비스 탐색: 어떤 브로커가 살아있는지 알려준다.
  - 리더 선출
    - 브로커 가운데 하나는 컨트롤러 역할을 담당해야 한다.
    - 한 클러스터에는 반드시 활성 상태 컨트롤러가 하나 있어야 한다.
    - 이 컨트롤러가 파티션 배치를 책임진다.
    - **아파치 주키퍼**, **etcd**가 보통 이용된다.

저장소

- 데이터 저장소: 메시지는 파티션 내 데이터 저장소에 보관된다.
- 상태 저장소: 소비자 상태는 이 저장소에 유지된다.
- 메타데이터 저장소: 토픽 설정, 토픽 속성 등은 이 저장소에 유지된다.

▼ 3️⃣ 상세 설계

▼ 데이터 저장소

메시지를 어떻게 지속적으로 저장할 것인가

🗂️ 고려사항

- 읽기와 쓰기가 빈번
- 갱신/삭제 연산은 없음
- 순차적인 읽기/쓰기 위주

선택지 1) 데이터베이스

- **관계형 데이터베이스**
  - 토픽별로 테이블 만들기
  - 토픽에 보내는 메시지는 해당 테이블에 새로운 레코드로 추가
- **NoSQL 데이터베이스**
  - 토픽별로 컬렉션 만들기
  - 토픽에 보내는 메시지는 하나의 문서가 됨

→ 동시에 대규모로 읽기/쓰기 연산을 처리하는 데이터베이스는 설계하기 어렵다.
→ 오히려
시스템 병목이 될 수도…

선택지 2) 쓰기 우선 로그(WAL)

새로운 항목이 추가되기만 하는 일반 파일 (ex: MySQL 복구 로그, 아파치 주키퍼, …)

지속성을 보장해야 하는 메시지는 디스크에 WAL로 보관
읽기/쓰기 연산 모두 순차적
회전식 디스크 기반 저장장치는 큰 용량을 저렴한 가격에 제공

- 세그먼트 단위로 나눔
- 새 메시지를 활성 상태의 세그먼트 파일에만 추가
- 세그먼트 크기가 한계에 도달하면 새 활성 세그먼트 파일이 새 메시지 수용, 기존 파일은 비활성 상태로 변경
- 비활성 세그먼트는 읽기 요청만 처리

▼ 메시지 자료 구조

생산자, 메시지 큐, 소비자 사이의 계약. 불필요한 복사를 막고 높은 대역폭 달성!

메시지 키

- **파티션 결정**

- 메시지 큐 내부적 개념으로 클라이언트에 노출되면 안됨
- 키 - 파티션 대응 알고리즘을 잘 정의하면 파티션 수가 달라져도 모든 파티션에 메시지가 계속 균등 분산 가능

메시지 값

- 메시지의 내용(페이로드)

메시지 기타필드

- **토픽**: 메시지가 속한 토픽의 이름
- **파티션**: 메시지가 속한 파티션 ID
- **오프셋**: 파티션 내 메시지의 위치
- **타임스탬프**: 메시지 저장 시각
- **크기**: 메시지의 크기
- **CRC**: 순환 중복 검사, 주어진 데이터의 무결성 보장에 이용

▼ 일괄 처리

생산자, 소비자, 메시지 큐는 메시지를 가급적 일괄 처리한다.
메시지 큐 안에서 메시지 일괄 처리를 위해 무슨 일을 할까?

- 여러 메시지를 한 번의 네트워크 요청으로 전송 → 네트워크 왕복 비용 감소
- 브로커가 메시지를 한 번에 로그에 기록하면 더 큰 규모의 순차 쓰기 연산 발생 → 운영체제가 관리하는 디스크 캐시에서 더 큰 공간을 점유 → 더 높은 디스크 접근 대역폭 달성

→ 높은 대역폭과 낮은 응답 지연은 동시 달성이 어렵다.

▼ 생산자 측 작업 흐름

라우팅 계층을 도입해서 적절한 브로커에 메시지를 보내자.

1. 생산자는 메시지를 라우팅 계층으로 보낸다.
2. 라우팅 계층은 메타데이터 저장소에서 사본 분산 계획을 읽어 자기 캐시에 보관한다.
3. 메시지가 도착하면 라우팅 계층은 리더 사본에 보낸다.
4. 리더 사본이 메시지를 받고 리더를 따르는 다른 사본은 해당 리더로부터 데이터를 받는다.
5. 충분한 수의 사본이 동기화되면 리더는 데이터를 디스크에 기록한다.

리더와 사본이 필요한 이유는?

장애 감내가 가능한 시스템을 만들기 위해

단점

- 라우팅 계층을 도입하면 네트워크 노드가 하나 더 늘어 오버헤드가 발생. 네트워크 전송 지연이 늘어남
- 일괄 처리를 고려하지 않은 설계

→ 라우팅 계층을 생산자 내부로 편입시키고 버퍼를 도입

- 전송 지연 감소
- 생산자가 독립적인 로직을 가짐
- 전송할 메시지를 버퍼 메모리에 보관했다가 일괄 전송하여 대역폭을 높임

얼마나 많은 메시지를 일괄처리할까? 대역폭과 응답 지연 사이에서 타협점을 찾는다.

▼ 소비자 측 작업 흐름

브로커가 소비자에게 보낼 것이냐. 소비자가 브로커에서 가져갈 것이냐.

푸시 모델

👍 장점

- 낮은 지연: 메시지를 받는 즉시 소비자에게 보낼 수 있다.

👎 단점

- 소비자가 메시지를 처리하는 속도 < 생산자가 메시지를 만드는 속도: 소비자에게 부하 발생
- 생산자가 데이터 전송 속도를 좌우하므로 소비자는 그에 대응해야 한다.

⭐ 풀 모델

👍 장점

- 메시지 소비 속도는 소비자가 결정한다. (실시간 or 일괄 or ?)
- 메시지 소비 속도 < 메시지 생산 속도 : 소비자를 늘리거나, 대기 가능
- 일괄 처리에 적합

👎 단점

- 브로커에 메시지가 없어도 소비를 시도하므로 컴퓨팅 자원이 낭비된다.

▼ 소비자 재조정

어떤 소비자가 어떤 파티션을 책임지는지 다시 정하는 프로세스

새로운 소비자 합류/이탈/장애 시 or 파티션 조정 시

코디네이터

- 소비자 재조정을 위한 통신 브로커 노드
- 소비자로부터 오는 박동 메시지를 살피고 소비자의 파티션 내 오프셋 정보 관리

동작 방식

- 소비자는 특정 그룹에 속하며 전담 코디네이터가 있음
- 코디네이터는 자신에게 연결된 소비자 목록을 유지
- 목록에 변화가 생기면 코디네이터는 새 리더를 선출
- 새 리더는 새 파티션 배치 계획을 만들고 코디네이터에게 전달
- 코디네이터는 해당 계획을 그룹 내 다른 모든 소비자에게 알림

▼ 상태 저장소

무엇이 저장되는가?

- 소비자에 대한 파티션의 배치 관계
- 각 소비자 그룹이 각 파티션에서 마지막으로 가져간 메시지의 오프셋

언제 사용되는가?

- 읽기/쓰기는 빈번하지만 양은 많지 않음
- 데이터 갱신은 빈번하지만 삭제는 거의 안됨
- 읽기 쓰기 연산은 무작위적 패턴
- 데이터의 일관성 중요

→ 주키퍼 같은 키-값 저장소를 사용하는 것이 좋겠다!

▼ 메타데이터 저장소

토픽 설정, 속성 정보(파티션 수, 메시지 보관 기간, 사본 배치 정보) 보관
상태 저장소와 비슷해서 주키퍼가 역시 적절

▼ 복제

높은 가용성을 보장하기 위해 …

파티션마다 3개의 사본을 갖고 모두 다른 브로커 노드에 분산한다.
생산자는 파티션에 메시지를 보낼 때 리더에게만 보내고,
다른 사본들은 리더에서 메시지를 가져와서 동기화한다.

메시지 동기화가 완료된 사본 개수가 N개를 넘으면 리더는 생산자에게 잘 받았다고 인사한다.

사본 분산 계획

어떻게 분산할 것인가?

조정 서비스(카프카)의 도움으로 브로커 노드 가운데 하나가 리더로 선출되면 해당 리더 브로커 노드가 사본 분산 계획을 만들고 메타데이터 저장소에 보관한다.

▼ 사본 동기화

동기화된 사본 `ISR`에서 동기화됐다는 기준이 뭐냐 → 토픽의 설정에 따라 달라진다.

ISR은 왜 필요한가?

- `성능`과 `영속성` 사이의 타협을 위해 !
- 모든 사본을 동기화해야 한다면, 하나라도 동기화를 빨리 못하면 전체가 느려진다.

메시지 수신 응답 설정

생산자는 k개의 ISR이 메시지를 받았을 때 리더가 대답하도록 설정할 수 있다.

- **ACK=all:** 가장 느린 응답, 영속성 최고
- **ACK=1**: 리더 저장 후 바로 응답, 직후 리더 장애 시 메시지 소실
- **ACK=0**: 수신 확인 대기 x, 재시도 x, 지표 수집/데이터 로깅용

▼ 규모 확장성

생산자

그룹 단위가 아니므로 매우 간단! 새로 추가/삭제가 쉬움

소비자

새 소비자 그룹은 추가/삭제 쉬움
같은 소비자 그룹 내 소비자가 추가/삭제 시
재조정 메커니즘이 처리

브로커

브로커 노드가 추가/삭제될 때 사본을 재배치

한시적으로 시스템에 설정된 사본 수보다 많은 사본을 허용하도록 변경하여 데이터 손실 방지

결함 내성을 위해 추가로 고려할 것

- 메시지가 성공적으로 합의되었다고 하려면 얼마나 많은 사본에 메시지가 반영되야 하나?
- 파티션의 모든 사본이 같은 브로커 노드에 있으면 노드 장애 발생 시 파티션이 소실된다.
- 파티션의 모든 사본에 문제가 생기면 파티션의 데이터는 영원히 사라진다.

파티션

파티션 수의 조정은 생산자-소비자 간 안전성에 영향을 주지 않는다.
파티션을 늘리면 간단히 토픽의 규모를 늘릴 수 있다.
파티션을 줄이면 일정 시간 유지 후 제거하기 때문에 저장 용량이 바로 늘어나지 않는다.
실제 파티션 제거 시점에 생산자 그룹은 재조정을 해야 한다.

▼ 메시지 전달 방식

분산 메시지 큐가 어떻게 메시지를 전달할 것인가.

최대 한 번

메시지가 소실되더라도 다시 전달되는 일은 없다.

- 생산자 → 토픽에 비동기적 메시지 송신 후 수신 응답을 기다리지 않음(ACK=0)
- 메시지 전달 실패 시 재시도하지 않음
- 소비자는 메시지 처리 전 오프셋 갱신. 오프셋 갱신 직후 장애 발생 시 재소비 불가

지표 모니터링, 소량의 데이터 손실 감수가 가능한 애플리케이션에 사용

최소 한 번

메시지가 소실되지 않는다.

- 생산자 → 토픽에 동기/비동기적 메시지 송신 후 ACK=1 or ACK=all 구성 이용
- 메시지 전달 실패 또는 타임아웃 시 계속 재시도

- 소비자는 데이터 처리 성공시에만 오프셋 갱신. 오프셋 갱신 전 장애 발생 시 중복 처리됨

중복 메시지 전송 가능성이 있으므로 메시지마다 고유한 키가 있는 경우 필터링해서 처리하자

정확히 한 번

- 구현 까다로움, 사용자 입장에서 편리
- 중복 허용하지 않음
- 지불, 매매, 회계 등의 시스템에 적합한 전송 방식

▼ 고급 기능

메시지 필터링

토픽에서 특정한 유형의 메시지만 관심있는 소비자 그룹이 있다면?

- **토픽을 시스템 별로 분리하자**
  → 다른 시스템에도 필요하면?
  → 같은 메시지를 여러 토픽에 저장하는 것은 자원 낭비다.
- **일단 받은 다음 버리자**
  → 유연성은 높지만 불필요한 트래픽이 발생
  → 시스템 성능이 저하된다.
- **브로커에서 메시지를 필터링해서 소비자가 받게 하자**

  → 복호화 / 역직렬화가 필요하다면 브로커 성능이 저하될 것이다.
  → 민감 데이터가 포함돼있다면 읽어서는 안된다.
  💡
  메시지마다 태그를 두자

메시지 지연 / 예약 전송

토픽에 바로 저장하지 않고 브로커 내부의 임시 저장소에 넣어두었다가 시간이 지나면 토픽으로 옮긴다.

`임시 저장소`와 `타이밍 기능`이 핵심

- 하나 이상의 특별 메시지 토픽을 임시 저장소로 활용 가능
- 메시지 지연 전송 전용 메시지 큐를 사용하여 특정 시간동안 메시지 전송을 지연하는 기능(타이밍)
- 계층적 타이밍 휠 사용

▼ 4️⃣ 마무리

더 이야기하려면...

- 프로토콜
  - 메시지 생산과 소비, 박동 메시지 교환 등의 모든 활동 설명
  - 대용량 데이터를 효과적으로 전송할 방법 설명
  - 데이터의 무결성을 검증할 방법 설명

→ AMQP, 카프카 프로토콜 등

- 메시지 소비 재시도
  - 새로 몰려드는 메시지들이 제대로 처리되지 못하는 일을 막으려면 어떻게 재시도 할까?
    - 실패한 메시지는 재시도 전용 토픽에 보내서 나중에 다시 처리하자
- 이력 데이터 아카이브
  - 시간 기반 / 용량 기반 로그 보관 메커니즘이 있다고 가정할 때 이미 삭제된 메시지를 다시 처리하고 싶다면?
    - HDFS 같은 대용량 저장소 시스템 or 객체 저장소에 보관해둔다.

토론 주제

메시지 지연/예약 전송을 소비자 기준으로 할 수는 없을까?

책에서 `주문 후 30분 이내 미결제시 주문 취소` 를 예시로 들었는데 이건 소비자 기준으로 확인해야 하는 문제지 않나? 더 깊게 이해하고 싶다.