



Apache Kafka - 고영일 (2025년 8월 14일)

요약본

발표 자료

포인트크러쉬 크롤링 인프라 아키텍처 리뷰

1. 서론: 왜 이 글을 작성하는가?
2. 카프카가 없었다면?: 동기식 처리의 악몽
 - 2.1. 시나리오 1: 병목 현상 (Bottleneck)
 - 2.2. 시나리오 2: 단일 실패 지점 (SPOF)
3. 해결책: 디커플링과 메시지 큐
 - 3.1. 왜 Kafka를 선택했는가?
4. Kafka 아키텍처 해부
 - 4.1. 핵심 구성 요소
 - 4.2. 포인트크러쉬 인프라 현황
 - 4.3. 워크플로우 분석
5. 데이터의 흐름과 신뢰성 보장
 - 5.1. 파티셔닝 전략: 어떻게 작업을 분배하는가?
 - 5.2. 전송 보장 수준: 어디까지 믿을 수 있는가?
 - 5.3. 수동 커밋: 신뢰성의 마지막 방어선
6. 현황 리뷰: 장점과 기술 부채
 - 6.1. 무릎을 탁 친 순간 (Pros)
 - 6.2. 개선이 필요한 지점 (Cons / 기술 부채)
7. 우리의 과제: TODO List

요약본

[kafka-summary.html](#)

발표 자료

[kafka-ppt.html](#)

포인트크러쉬 크롤링 인프라 아키텍처 리뷰

작성자	고영일
작성일	2025년 8월 14일

1. 서론: 왜 이 글을 작성하는가?

안녕하세요. 오늘 이 글의 목표는 제가 현재 담당하고 있는 포인트크러쉬 크롤링 인프라의 구조와 흐름을 공유하고, 이 시스템을 지탱하는 핵심 기술인 **아파치 카프카(Apache Kafka)**에 대해 알아보는 시간을 갖는 것입니다.

이 시스템은 제가 처음부터 만든 것은 아니지만, 유지보수하고 발전시켜나가며 그 설계의 우수성과 몇 가지 기술적 과제들을 발견할 수 있었습니다. 이 글을 통해 팀원들이 시스템을 함께 이해하고, 앞으로 더 나은 방향으로 발전시켜 나갈 수 있는 기반을 마련하고자 합니다. '이 시스템이 왜 이렇게 만들어졌을까?'라는 질문에 대한 답을 함께 찾아가는 시간이 되었으면 합니다.

2. 카프카가 없었다면?: 동기식 처리의 악몽

본격적인 아키텍처 분석에 앞서, '만약 우리에게 카프카가 없었다면 어땠을까?'라는 가상 시나리오를 통해 **분산 메시징 시스템의 필요성**을 먼저 짚어보겠습니다.

2.1. 시나리오 1: 병목 현상 (Bottleneck)

만약 포인트크러쉬 어드민 API 서버와 크롤링 서버가 카프카 없이 직접 **동기식(Synchronous) API**로 통신한다고 가정해 봅시다.

- 요청 폭주:** 어드민에서 Tor IP 우회를 사용하는 등 처리 시간이 오래 걸리는 SEO 크롤링 작업을 100건 동시에 요청합니다.
- 서버 블로킹(Blocking):** 어드민 API 서버는 100건의 크롤링이 모두 완료되고 응답을 받을 때까지 다른 요청을 처리하지 못하고 기다립니다.
- 장애 전파:** 바로 이때, 어드민 사용자가 다른 기능을 사용하기 위해 API를 호출하면 어떻게 될까요? 어드민 API 서버가 응답 불능 상태이므로, 이 요청은 타임아웃되고 결국 실패하게 됩니다.

이처럼 크롤링이라는 하나의 작업 지연이, 전혀 상관없어 보이는 어드민 전체 기능에 영향을 미치는 **장애 전파(Error Propagation)**가 발생합니다. 이것이 바로 시스템 간의 의존성이 강한 '**강한 결합(Tightly Coupling)**'이 가진 가장 큰 문제입니다.

2.2. 시나리오 2: 단일 실패 지점 (SPOF)

이번엔 크롤링 서버 인스턴스가 단 한 대뿐이고, 이 서버에 예기치 않은 장애가 발생했다고 가정해 봅시다.

서버가 다운되면 복구 전까지 **모든 크롤링 관련 기능이 중단**됩니다. 이는 서비스의 핵심 기능이 단 하나의 컴포넌트에 의존하고 있어 매우 취약한 구조라는 것을 의미합니다. 바로 '**단일 실패 지점(Single Point of Failure)**' 문제죠.

3. 해결책: 디커플링과 메시지 큐

앞서 본 두 가지 아찔한 상황을 해결하는 핵심 아이디어가 바로 '**디커플링(Decoupling)**', 즉 시스템 간의 의존성을 분리하는 것입니다.

중간에 **메시지 큐(Message Queue)**라는 완충 지대를 두면, 서버와 크롤러는 서로의 상태에 영향을 받지 않게 됩니다. 서버는 요청을 메시지 큐에 '던져놓기'만 하면 되고, 크롤러는 자신이 처리할 수 있을 때 메시지를 '가져가면' 되니까요.

3.1. 왜 Kafka를 선택했는가?

메시지 큐 기술에는 RabbitMQ, AWS SQS 등 다양한 선택지가 있습니다. 이들은 주로 '작업 큐' 관리에 강점이 있고, 메시지가 소비되면 사라지는 특징이 있습니다.

반면 우리 시스템이 채택한 **카프카**는, 단순히 메시지를 전달하는 것을 넘어 **이벤트를 디스크에 저장하는 '분산 이벤트 스트리밍 플랫폼'**입니다. 이 덕분에 대규모 데이터를 안정적으로 처리하고, 문제가 발생했을 때 데이터를 유실하

지 않고 **재처리**할 수 있는 강력한 장점을 갖게 됩니다.

4. Kafka 아키텍처 해부

4.1. 핵심 구성 요소

우리 시스템을 이해하기 위해 꼭 알아야 할 카프카의 핵심 용어들입니다.

용어	설명	비유
이벤트(Event)	"어떤 일이 일어났다"는 사실을 기록한 데이터의 기본 단위	일기장의 한 페이지
토픽(Topic)	이벤트가 발행되는 카테고리 또는 피드 이름	책
파티션(Partition)	하나의 토픽을 여러 개로 나눈 로그. 병렬 처리의 핵심.	책의 챕터
오프셋(Offset)	각 파티션 내에서 메시지가 갖는 고유하고 순차적인 ID	페이지 번호
프로듀서(Producer)	토픽에 이벤트를 발행하는 클라이언트	저자
컨슈머(Consumer)	토픽을 구독하여 이벤트를 처리하는 클라이언트	독자
컨슈머 그룹	여러 컨슈머를 묶어 작업을 병렬로 분담하는 그룹	스터디 그룹
브로커(Broker)	카프카 서버 자체. 여러 브로커가 모여 클러스터를 구성.	사서

4.2. 포인트크러쉬 인프라 현황

- 프로듀서: 1대 (NestJS)
- 브로커: 1대
- 컨슈머: 총 29대 (NestJS)

4.3. 워크플로우 분석

1. **작업 요청 및 발행:** Express.js 기반의 어드민 서버가 NestJS 프로듀서에게 API를 호출합니다. 프로듀서는 해당 토픽으로 이벤트를 `emit` 하고 즉시 응답합니다. (Fire-and-Forget)
2. **처리 및 보고:** 컨슈머가 이벤트를 받아 크롤링을 수행한 후, 어드민 서버의 API를 직접 호출(Callback)하여 결과를 보고합니다.

이 **비대칭 통신 패턴**은 작업 요청 시에는 빠른 응답을 보장하고, 결과 보고 시에는 데이터의 정합성을 확보하는 효과적인 방법입니다.

5. 데이터의 흐름과 신뢰성 보장

5.1. 파티셔닝 전략: 어떻게 작업을 분배하는가?

카프카는 프로듀서가 보낸 메시지를 어떤 파티션에 저장할지 결정하는 **파티셔닝 전략**을 가집니다.

- **라운드-로빈 (현재 우리 방식):** 메시지에 별도의 키가 없으면, 파티션에 순서대로 균등하게 분배합니다. 처리량 극대화에 유리합니다.

- **키 기반 해시:** 메시지 키의 해시 값을 계산하여 특정 파티션에 할당합니다. 동일 키를 가진 메시지는 항상 순서가 보장됩니다.
- **커스텀 파티셔너:** 개발자가 직접 비즈니스 로직에 따라 분배 규칙을 정할 수 있습니다.

5.2. 전송 보장 수준: 어디까지 믿을 수 있는가?

카프카는 데이터의 신뢰성을 위해 세 가지 수준의 전송을 보장합니다.

- **At-Most-Once (최대 한 번):** 메시지를 보내고 성공 여부를 확인하지 않습니다. 속도가 가장 빠르지만 메시지 유실 가능성이 있습니다. (현재 우리 프로듀서의 **emit** 방식)
- **At-Least-Once (최소 한 번):** 성공 응답(ack)을 받을 때까지 재전송합니다. 유실은 없지만 중복 처리 가능성이 있습니다.
- **Exactly-Once (정확히 한 번):** 유실과 중복 없이 정확히 한 번만 처리됨을 보장합니다.

5.3. 수동 커밋: 신뢰성의 마지막 방어선

우리 컨슈머는 수동 커밋(**Manual Commit**) 방식을 사용합니다. 이는 "작업이 끝났다"고 카프카에 알리는 시점을 개발자가 직접 제어하는 것입니다.

```
// 1. 메시지 처리 (크롤링)
const result = await doCrawling(message);

// 2. 서버에 결과 보고 (API 콜백)
const report = await reportToServer(result);

// 3. 서버 응답 성공 시에만 오프셋 커밋
if (report.success) {
  await consumer.commitOffsets();
}



// 4. 실패 시 커밋하지 않음 (재처리 유도)
```

이 패턴 덕분에, 어떤 단계에서든 오류가 발생해도 오프셋이 커밋되지 않아 해당 작업은 다른 컨슈머에 의해 안전하게 재처리될 수 있습니다.


6. 현황 리뷰: 장점과 기술 부채

제가 이 시스템을 운영하며 느낀 장점과 앞으로 우리가 함께 개선해야 할 지점들입니다.

6.1. 무릎을 탁 친 순간 (Pros)

-  **위기 대응 능력 (탄력적 확장성):** 미션 등록이 급증했을 때, 컨슈머 인스턴스를 몇 대 추가하는 것만으로 시스템 중단 없이 트래픽을 감당해냈습니다.
-  **용이한 디버깅 (로그 연속성):** 카프카는 처리된 이벤트를 보관하므로, 특정 작업 실패 시 UI 대시보드를 통해 해당 메시지를 다시 보며 원인을 분석하기 용이합니다.

6.2. 개선이 필요한 지점 (Cons / 기술 부채)

-  **'깜깜이' 운영 (랙 확인 불가):** **emit** 방식으로 인해 컨슈머가 얼마나 밀려있는지(랙)를 확인할 수 없어, 직관적인 부하 모니터링이 어렵습니다.

- **⚠️ 되돌릴 수 없는 파티션 설정:** 과거에 순위 크롤링 토픽의 파티션이 잘못 증설되어, 현재 컨슈머 수와 파티션 할당 비율이 맞지 않는 리소스 낭비가 발생하고 있습니다.
- **⚠️ 외부 의존성으로 인한 장애:** SEO 크롤링에 사용되는 Tor IP 갱신 과정에서 타임아웃이 발생하면, 해당 컨슈머 전체가 장애로 이어지는 경우가 있습니다.

7. 우리의 과제: TODO List

마지막으로, 이 시스템을 함께 발전시키기 위한 우리의 과제입니다.

1. **프로듀서 리팩토링:** `emit` 방식에서 `send` 방식으로 전환하여 `acks` 설정 및 멱등성 확보 방안을 모색합니다.
2. **토픽 마이그레이션 전략 수립:** 파티션 수가 잘못된 토픽의 데이터를 신규 토픽으로 이전하고, 기존 토픽을 제거하는 장기적 계획을 수립합니다.
3. **Dead Letter Queue (DLQ) 도입:** 반복적으로 실패하는 메시지를 별도 토픽으로 분리하여, 전체 시스템의 안정성을 향상시킵니다.
4. **모니터링 고도화:** 컨슈머 랙(Lag)을 정밀하게 추적하고 임계치 초과 시 알림을 받는 시스템을 구축합니다.

이 글을 통해 우리 팀이 카프카의 원리를 이해하고, 현재 시스템의 장점과 한계를 명확히 인지하여 앞으로 더 나은 시스템을 함께 만들어나가는 데 자신감을 얻기를 바랍니다.