10장: 알림 시스템 설계

난이도	⭐⭐⭐
학습날짜	@2026년 1월 14일

알림 시스템

알림 시스템은 사용자에게 중요할 만한 정보를 비동기적으로 제공한다.

대표적인 알림 시스템으로는 모바일 푸시 알림, SMS 메시지, 그리고 이메일의 3가지 분류할 수 있다.

🤔 알림 시스템의 다양한 종류

1. 모바일 푸시 알림 (Mobile Push Notification)

스마트폰 앱을 통해 사용자에게 팝업을 띄우거나 배지를 표시하는 가장 일반적인 방식입니다. OS(운영체제) 제공자가 알림 전달의 핵심 키를 쥐고 있습니다.

- **iOS (애플): APNS (Apple Push Notification Service)**
  - 애플 기기로 알림을 보내기 위해서는 반드시 애플이 제공하는 이 서비스를 거쳐야 합니다. 보안이 매우 엄격합니다.
- **Android (구글): FCM (Firebase Cloud Messaging)**
  - 과거 GCM(Google Cloud Messaging)의 후속 버전입니다.
  - **특징:** 현재는 안드로이드뿐만 아니라 iOS, 웹(Web)까지 통합하여 알림을 보낼 수 있는 **크로스 플랫폼 기능을 제공**하기 때문에, 많은 기업이 **FCM을 메인 허브로 사용하여 iOS와 안드로이드를 동시에 처리**합니다.

2. SMS 및 메신저 알림 (SMS & Third-party Messengers)

앱이 설치되지 않은 사용자나, 더 확실한 도달률이 필요할 때 사용합니다.

- **SMS/LMS/MMS:** 전통적인 문자 메시지입니다. 비용이 비싼 편입니다.
  - *대표 서비스:* Twilio (글로벌), Nexmo, 국내의 경우 NHN Cloud Notification, Naver Cloud, CoolSMS 등.
- **카카오톡 알림톡 (한국 특화):**
  - 한국에서는 SMS 대체제로 가장 많이 사용됩니다. 비용이 SMS보다 저렴하고 정보성 메시지(주문 내역, 배송 현황 등) 전달에 최적화되어 있습니다.
- **WhatsApp Business API:**
  - 글로벌 서비스에서 SMS 대신 많이 사용되는 메신저 기반 알림입니다.

3. 이메일 (Email)

마케팅, 본인 인증, 정기 뉴스레터 등에 사용됩니다.

- **대표 서비스:**
  - **AWS SES (Simple Email Service):** 클라우드 기반으로 확장성이 좋고 비용이 저렴해 가장 많이 쓰입니다.
  - **SendGrid / Mailgun:** 이메일 전송 성공률(Deliverability) 관리와 마케팅 분석 도구가 강력한 서비스들입니다.

4. 웹 푸시 (Web Push)

모바일 앱이 없어도 **PC나 모바일 브라우저를 통해 알림을 보냅니다**. 브라우저가 켜져 있지 않아도(백그라운드) 알림을 받을 수 있게 해줍니다.

- **기술 스택:** Service Worker와 Web Push API 표준을 사용합니다.
- **전송 방식:** 크롬, 파이어폭스, 사파리 등의 브라우저 벤더 서버를 경유합니다. FCM을 통해서도 구현 가능합니다.

개략적 설계

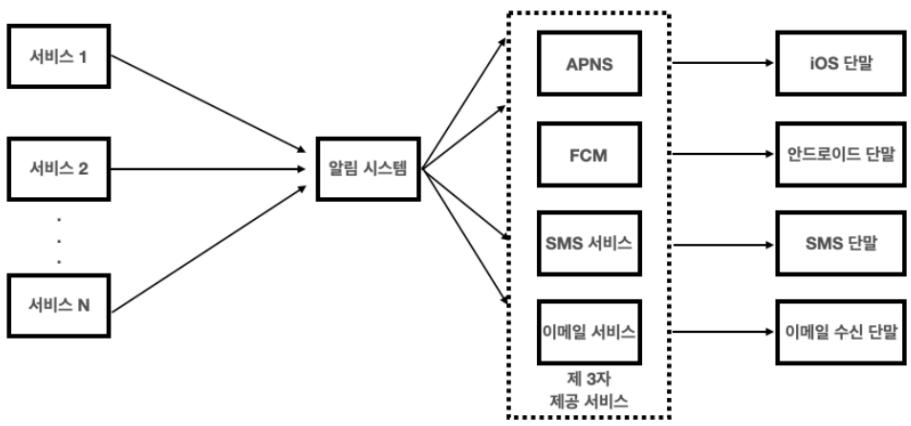
알림 시스템을 설계하기 위해서 어떤 항목들에 대해 설정해야 할까?

1. **지원해야 하는 알림의 종류**
   - 푸시 알림, SMS 메세지, 이메일, 웹 푸시 등
2. **실시간 시스템 여부**
   - 연성 시스템 (Soft Real-Time) 시스템으로 가정.

     연성 시스템이란, 가능한 빨리 전달되어야 하나 시스템에 높은 부하가 걸렸을 때 약간의 지연을 허용
3. **지원해야 하는 단말의 종류**
   - iOS 단말, 안드로이드 단말, 그리고 렙탑/데스크톱을 지원해야 함
4. **알림을 생성하는 주체**
   - 클라이언트가 생성할 수도, 서버가 스케쥴링을 통해 생성할 수도 있다.

5. **사용자가 알림 허용 여부를 설정할 수 있는지**
    - 허용/거부를 통해 설정할 수 있다.
6. **하루에 몇 건의 알림을 보낼 수 있어야 하는지**
    - 하루에 1000만 건의 알림을 보낼 수 있어야 한다.
        - 1초 ~= 11건 처리

알림 유형별 지원 방안



[구성요소]

1. 서비스 1 ~ 서비스 N :
    - 마이크로서비스일 수도 있고, 서버의 크론잡일 수도 있다.
2. 알림 시스템:
    - 알림 전송/수신 처리의 핵심
    - 서비스에 알림을 전송하기 위한 API 제공, 제3자 서비스에 전달할 알림 페이로드를 만들어 낼 수 있어야 한다.

🤔 알림 시스템의 기능

- 알림 전송 API
- 알림 검증
- 데이터베이스 또는 캐시 질의
- 알림 전송

```
// JSON 예시
{
  "to" : [
     {
       "user_id" : 1234
     }
  ],
  "from" : {
    "email" : "cale.jiho@gmail.com"
  },
  "subject" : "Hello World!",
  "content" : [
   {
     "type" : "text/plain",
     "value" : "Hello world!"
   }
  ]
}
```

3. 제3자 서비스:
    - 사용자에게 알림을 실제로 전달하는 역할
4. 각 단말.

각 단말별 제3자 서비스

1. iOS 단말
    - APNS (Apple Push Notification Service)에서 알림을 받아 처리

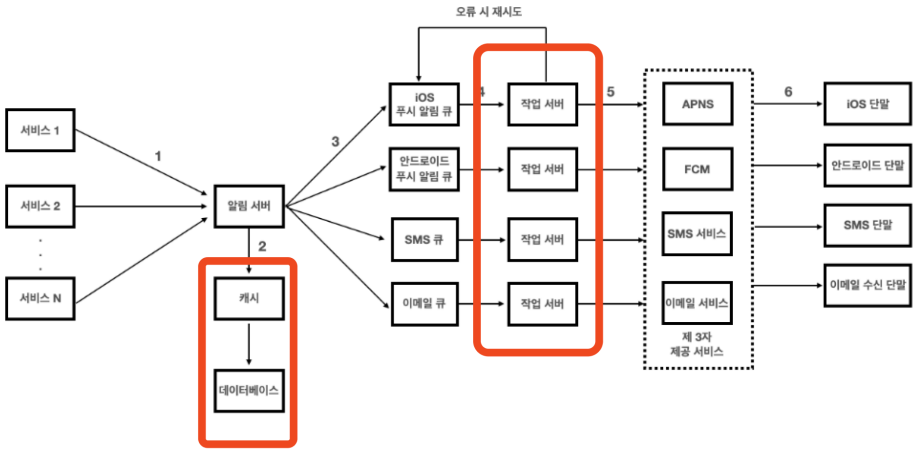
2. 안드로이드 푸시 알림
    - FCM (Firebase Cloud Messaging)
3. SMS 메세지
    - 트윌리오(Twilio), 넥스모 (Nexmo)와 같은 제3자 서비스 사용
4. 이메일
    - 회사 고유 이메일 서버 구축
    - 센드그리드 (Senggrid), 메일침프 (Mailchimp) 등

🤔 제3자 서비스를 사용할 때 고려해야할 **"확장성"**

1. 특정 제3 서비스는 특정 시장에서는 사용할 수 없을 수 있고, 새로운 서비스로 통합하거나 기존 서비스를 쉽게 제거할 수 있어야 한다.
2. SPOF (Single Point-Of Failure)
   알림 시스템에 서버가 하나밖에 없다면, 해당 서버에 장애가 발생하면 전체 서비스의 장애로 이어질 수 있다.
3. 규모 확장성 / 성능 병목
   알림 작업은 자원을 많이 필요로 할 수 있다.
    - HTML 페이지를 만들고 제3자 서비스의 응답을 기다리는 일은 시간이 오래 걸린다.

개략적 설계 (개선된 버전)

위에서 발견한 문제점들을 토대로 다음과 같이 개선할 수 있다.



1. 데이터베이스와 캐시를 알림 시스템의 주 서버에서 분리하여 **응답 속도를 높인다.**

🤔 알림 시스템에서 캐싱하는 값?

1. 사용자 정보 및 설정

알림을 보내려면 **"누구에게, 어디로, 보내도 되는지"** 확인해야 합니다.

- **캐싱하는 데이터:**
  - **디바이스 토큰:** FCM/APNS에 보낼 사용자의 스마트폰 고유 토큰 (예: `User_123` → `fcm_token_abc...`)
  - **수신 설정 (Preference):** "마케팅 알림 동의 여부", "야간 알림 차단 여부" 등

2. 중복 방지 (Deduplication / Idempotency)

"같은 걸 여러 번 주지 않기 위해"

- **작동 원리:**
  1. 알림 요청이 오면 `이벤트_ID`를 키(Key)로 캐시를 확인합니다.
  2. 캐시에 이미 있다면? → "이미 처리된 알림"이므로 무시합니다.
  3. 캐시에 없다면? → 알림을 발송하고, 캐시에 `이벤트_ID`를 저장(Write)합니다. (보통 TTL을 짧게 설정)
- **효과:** 멱등성(Idempotency, 연산을 여러 번 적용해도 결과가 달라지지 않는 성질)을 보장합니다.

3. 빈도 제어 (Rate Limiting)

한 사용자에게 너무 많은 알림이 쏟아지는 것을 막기 위해 사용합니다. (예: "비밀번호 오류 알림은 1분에 1번만")

- **작동 원리:**
  - Redis의 **Counter** 기능을 사용합니다.
  - `User_123:failed_login_alert` 같은 키를 만들고, 1분 동안 카운트가 5를 넘으면 더 이상 알림을 보내지 않고 버립니다.
- **이유:** 스팸으로 인식되거나 사용자가 앱을 삭제하는 것을 방지합니다.

4. 알림 템플릿 (Notification Templates)

`"안녕하세요 {name}님, 주문하신 {item}이 배송되었습니다."` 같은 문구 양식입니다.

- **캐싱하는 데이터:** 자주 변하지 않는 알림 문구 템플릿.
- **이유:** 매번 파일 시스템이나 DB에서 템플릿을 읽어오면 I/O 비용이 발생하므로 메모리에 올려둡니다.

2. 알림 서버를 증설하고 자동으로 **수평적 규모 확장**이 이루어질 수 있도록 한다.
3. **메시지 큐**를 이용하여 시스템 컴포넌트 사이의 강한 결합을 끊는다.

상세 설계

안정성

- **데이터 손실 방지**

  어떤 상황에서도 알림이 소실되지 않도록 하기

🤔 알림 시스템의 데이터 손실 방지 로직

1. **알림 로그 선(先) 저장 (Notification Log / Outbox Pattern)**

요청이 들어오자마자 큐에 넣는 것이 아니라, **데이터베이스에 먼저 기록**합니다.

- **로직:**
  1. 클라이언트가 알림 요청을 보냄.
  2. 서버는 알림 내용을 DB의 `Notification_Log` 테이블에 저장 (상태: `PENDING`).
  3. 저장이 **성공(Commit)**하면 그제서야 클라이언트에게 "접수되었습니다(HTTP 200)" 응답을 보냄.
  4. 이후에 비동기로 메시지 큐에 넣음.
- **이유:** 서버가 큐에 넣기도 전에 다운되더라도, DB에 기록이 남아있으므로 나중에 배치(Batch) 프로그램이 실패한 건을 찾아 재처리할 수 있습니다.

2. **[메세지 큐]**

a. **메세지 큐의 영속성 (영속성 : *데이터*를 *생성한 프로그램이 종료되어도 사라지지 않는* *데이터*의 *특성*)**
메시지 큐(Kafka, RabbitMQ 등)가 메모리에서만 동작하면 전원 공급이 중단되었을 때 데이터가 날아갑니다.

- **설정:**
  - **디스크 저장 (Persist to Disk):** 메시지를 받으면 메모리뿐만 아니라 디스크(Log file)에도 씁니다.
  - **복제 (Replication):** Kafka의 경우 여러 브로커(서버)에 데이터를 복제해 둡니다. 한 대가 터져도 다른 서버에 데이터가 남아 있습니다.

b. **생산자의 확인**
알림 서버(Producer)가 큐에 메시지를 넣을 때, "잘 들어갔니?"라고 묻는 과정입니다.

- **로직:**
  - 메시지 큐로부터 Ack(Acknowledgement, 수신 확인)를 받아야만 다음 로직을 진행하거나 성공으로 간주합니다.
  - 만약 Ack가 오지 않거나 에러가 나면, 즉시 재시도(Retry)를 하거나 DB에 '큐 전송 실패'로 기록합니다.

c. **소비자의 수동 확인**
워커(Worker)가 큐에서 메시지를 꺼내서 발송하는 도중에 워커 서버가 죽는 상황을 대비합니다.

- **자동 ACK (Auto ACK) vs 수동 ACK (Manual ACK):**
  - **위험한 방식 (Auto ACK):** 큐에서 메시지를 꺼내는 순간 큐에서 삭제함. (꺼내자마자 워커가 죽으면 메시지 증발)
  - **안전한 방식 (Manual ACK):**
    1. 메시지를 꺼낸다. (큐에서 삭제하지 않고 '처리 중'으로 표시)
    2. 실제 알림 발송 로직(FCM 호출 등)을 수행한다.
    3. 성공하면 큐에게 "다 했어(ACK)"라고 신호를 보낸다.
    4. 큐는 그제서야 메시지를 삭제한다.
  - **효과:** 워커가 발송 도중 죽으면, 큐는 ACK를 받지 못했으므로 일정 시간 후 해당 메시지를 다른 워커에게 다시 줍니다. (단, 이 과정에서 중복 발송 가능성이 생기므로 앞서 배운 **중복 방지 로직**이 함께 필요합니다.)

d. 재시도 및 죽은 편지함 (Retry & Dead Letter Queue)

모든 노력을 했음에도 외부 시스템(FCM, 메일 서버) 장애로 계속 실패하는 경우입니다.

- **재시도 (Retry):** 지수 백오프(Exponential Backoff, 1초, 2초, 4초... 간격 늘리기)를 적용해 몇 번 더 시도합니다.
- **Dead Letter Queue (DLQ):** 정해진 횟수(예: 5회)만큼 재시도해도 실패하면, 메시지를 삭제하지 않고 **별도의 보관함(DLQ)**으로 옮깁니다.
- **사후 처리:** 개발자가 DLQ를 모니터링하다가 원인(예: 템플릿 오류, 외부 서버 다운)을 해결한 뒤 재발송(Replay)합니다.

- **알림 중복 전송 방지**

같은 알림이 보내지지 않도록 중복 주의해야 함.

🤔 알림 시스템의 중복 알림 전송 방지 로직
가장 널리 쓰이는 방식은 **Redis의 원자적 연산(Atomic Operation)**을 활용하는 방법

1. 전제 조건: 고유 ID (Deduplication ID) 생성

중복인지 확인하려면 각 알림 메시지에 **'지문'**이 있어야 합니다.
알림을 요청하는 시점(생산자)에 **고유한 ID**를 생성해서 메시지에 포함해야 합니다.

- **예시:** `order_1234_shipped` (주문번호 + 이벤트 타입) 또는 UUID.
- **주의:** 단순히 랜덤한 ID를 만들면 안 되고, 비즈니스 로직과 연관된 ID여야 "같은 사건에 대한 알림"임을 식별할 수 있습니다.

2. 핵심 로직: Redis를 이용한 '검사 후 선점' (Check-and-Set)

단순히 "캐시에 있나 확인하고(GET), 없으면 저장(SET)"하는 방식은 동시성 이슈(Race Condition)가 발생해 중복 발송될 수 있습니다. 반드시 **원자적 명령어**를 써야 합니다.

단계별 흐름

1. **워커(Worker)가 큐에서 메시지를 꺼냄.** (메시지 안에 `event_id` 포함)
2. **Redis에 `SETNX` (Set if Not Exists) 명령 수행.**
    - 명령어: `SET event_id "processing" NX EX 3600`
    - 의미: "`event_id`라는 키가 **없을 때만** 저장해라. 성공하면 값을 쓰고, 유효시간(TTL)은 1시간으로 해라."
3. **결과 확인:**
    - **성공(Return 1/True):** "내가 이 알림을 처음 처리하는 놈이구나!" → **알림 발송 진행.**
    - **실패(Return 0/False):** "누군가 이미 처리 중이거나 처리했구나!" → **메시지 폐기(Discard) 또는 로그만 남김.**

이 `SETNX` 방식이 중복 방지의 90%를 담당합니다. 아주 빠르고 정확합니다.

3. 방어벽: 데이터베이스 유니크 키 (DB Unique Constraint)

Redis가 다운되거나 데이터가 날아갔을 때를 대비한 2차 방어선입니다.

- **로직:**
    - 알림 발송 이력을 저장하는 테이블(`NotificationHistory`)에 `event_id` 컬럼을 만들고 **UNIQUE INDEX**를 겁니다.
- **작동:**
    - 워커가 알림을 보내고 DB에 저장(INSERT)을 시도합니다.
    - 만약 이미 처리된 건이라면 DB 수준에서 **Duplicate Key Error**가 발생합니다.
    - 이 에러를 잡아서(Catch) "이미 처리됨"으로 간주하고 로직을 종료합니다.

4. 엣지 케이스 (Edge Case): 발송은 했는데 마킹에 실패하면?

가장 골치 아픈 상황입니다.

> 워커가 FCM에 발송은 성공함 → 하지만 Redis 업데이트나 DB 저장을 하기 직전에 워커 서버가 다운됨.

1. 큐(Queue)는 "어? 처리가 안 끝났네?" 하고 잠시 후 다른 워커에게 동일한 메시지를 줍니다.
2. 다른 워커는 Redis/DB를 확인합니다. 기록이 없으므로(이전 워커가 저장 못 하고 죽었으므로) **또 보냅니다.**

해결책:
완벽한 1회 전송(Exactly-once)은 분산 시스템에서 매우 어렵습니다. 하지만 이를 완화하기 위해 외부 공급자(Provider)의 기능을 활용합니다.

- **FCM/APNS의 중복 방지 기능 활용:**
    - FCM API를 호출할 때도 우리가 만든 `event_id`를 같이 보냅니다. (FCM에서는 `collapse_key`나 `message_id` 등으로 활용 가능)
    - FCM 서버 쪽에서 "어, 이거 아까 받은 ID인데?" 하고 자체적으로 중복을 걸러주기도 합니다.

알림 템플릿

대부분의 알림은 유사한 형태를 띈다.

따라서, 인자나 스타일, 추적 링크만을 조정하면 사전에 지정한 형식에 맞춰 알림을 만들어 내는 틀을 만들어두면 된다.

> 여러분이 꿈궈온 그 상품을 우리가 준비했습니다. [item_name]이 다시 입고되었습니다! [date]까지만 주문 가능합니다!

알림 설정

사용자가 알림 설정 여부를 설정할 수 있다.

필드명	타입	비고
user_id	bigint	
channel	varchar	알림이 전송될 채널. 푸시 알림, 이메일, SMS 등
opt_in	boolean	해당 채널로 알림을 받을 것인지의 여부

전송률 제한

→ 사용자에게 너무 많은 알림을 보내지 않도록 제한

재시도 방법

푸시 알림과 보안

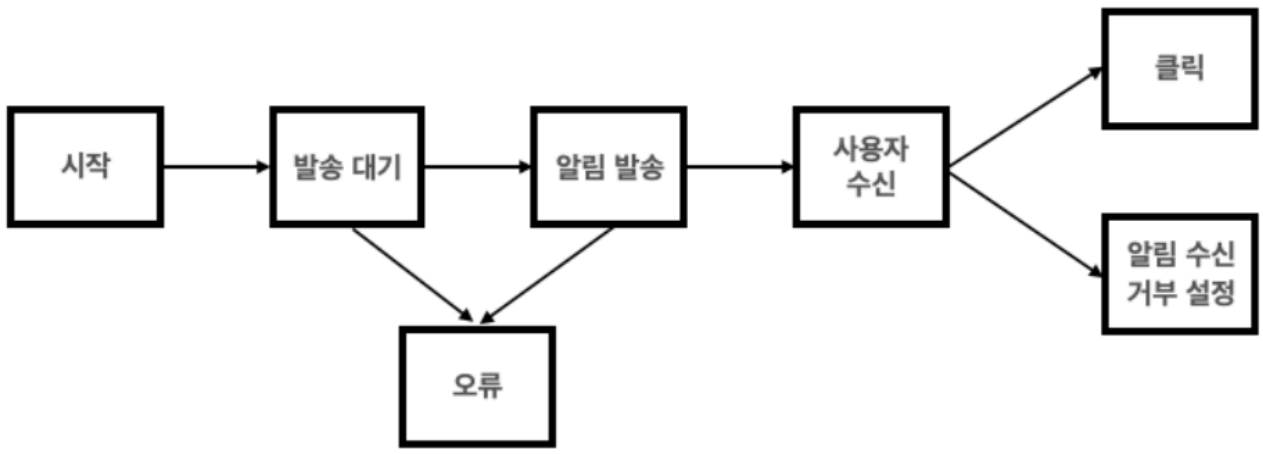
알림 전송 API에 비밀키 (appKey, appSecret)를 사용해야 함.

큐 모니터링

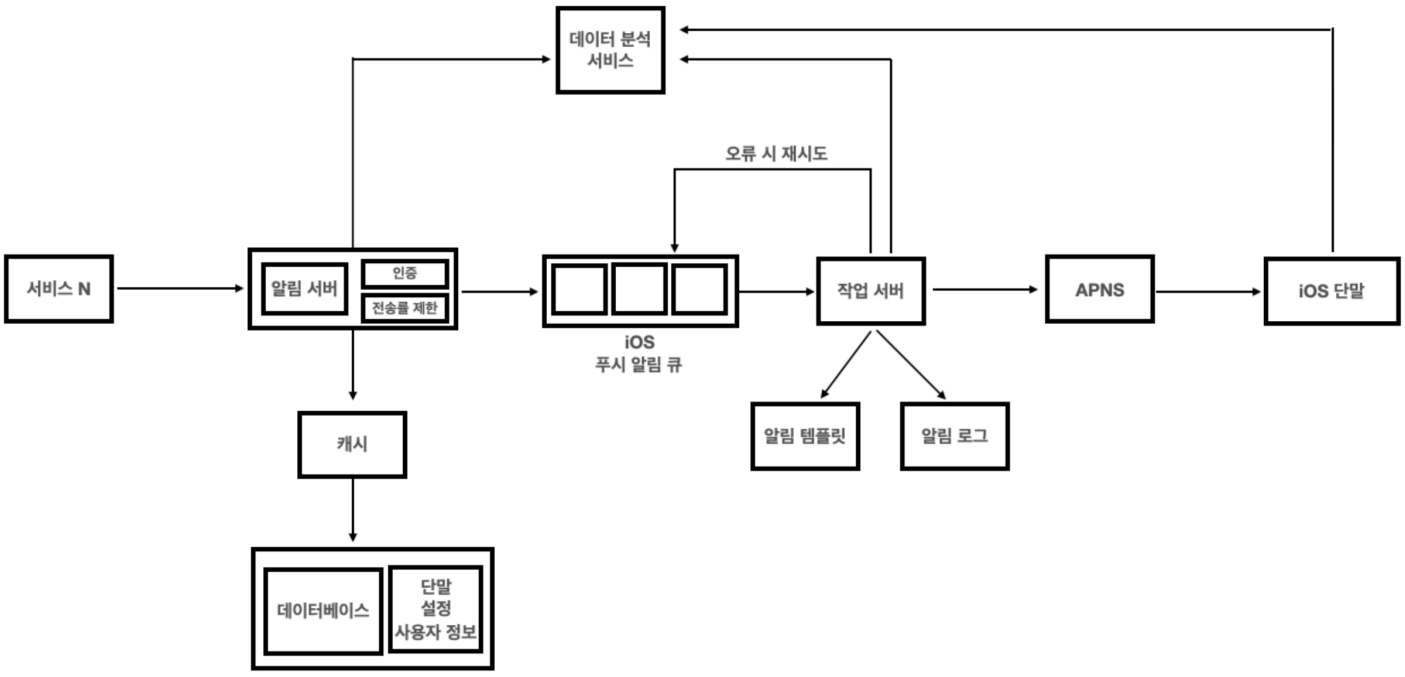
큐에 쌓인 알림의 개수 모니터링.

이벤트 추적

알림 확인율, 클릭율, 실제 앱 사용으로 이어지는 비율 등 메트릭을 분석.



수정된 설계안


데이터 분석
서비스
오류 시 재시도
서비스 N
알림 서버
인증
전송률 제한
iOS
푸시 알림 큐
작업 서버
APNS
iOS 단말
캐시
알림 템플릿
알림 로그
데이터베이스
단말
설정
사용자 정보

📌 🔥 부록5 : **이상적으로 1번만 보내는 시스템 설계 방법 (링크)** 🔥

이 아티클 **"You Cannot Have Exactly-Once Delivery (정확히 한 번 전송이란 존재하지 않는다)"**는 분산 시스템 엔지니어링의 핵심 진실을 다루고 있는 매우 유명한 글입니다.
우리가 앞서 이야기했던 "데이터 손실 방지(At-least-once)"와 "중복 처리(Idempotency)"가 왜 필수적인지 이론적 배경을 제공합니다.
핵심 내용을 요약해 드립니다.

1. 핵심 주장: "정확히 한 번(Exactly-once) 전송은 불가능하다"

분산 시스템(서버, DB, 큐가 네트워크로 연결된 환경)에서 메시지를 누락 없이, 그리고 중복도 없이 '항상 딱 한 번'만 전달하는 것은 수학적/물리적으로 불가능합니다.

- **이유 (두 장군 문제, Two Generals Problem):**
  - 메시지를 보내는 쪽과 받는 쪽이 불확실한 네트워크(Unreliable Link)를 통해 완벽하게 합의하는 것은 불가능하다는 이론입니다.
  - A가 메시지를 보내고 B가 받았습니다. B는 "받았다(ACK)"는 신호를 보냅니다. 하지만 이 ACK가 네트워크에서 사라진다면?
  - A는 B가 못 받았다고 생각해서 **다시 보냅니다(중복 발생).**
  - 반대로 A가 다시 보내지 않으면? 만약 진짜로 B가 못 받은 거라면 **누락(데이터 손실) 발생.**
  - 즉, **누락(0회)과 중복(2회 이상) 사이에서 하나를 선택**해야만 합니다.

2. 세 가지 전송 방식 (Delivery Semantics)

글쓴이는 전송 보장 수준을 세 가지로 분류합니다.

1. **At-most-once (최대 1회):**
   - "보내고 잊어버리기(Fire and forget)".
   - 빠르지만, 메시지가 중간에 사라져도 모릅니다. (알림 시스템에서는 부적합)
2. **At-least-once (최소 1회):**
   - "확인받을 때까지 계속 보내기".
   - 메시지 손실은 없지만, 필연적으로 **중복 전송**이 발생합니다.
   - **현존하는 대부분의 신뢰성 있는 메시지 큐(RabbitMQ, Kafka 등)가 채택한 방식**입니다.
3. **Exactly-once (정확히 1회):**
   - 이상적인 유니콘 같은 존재입니다.
   - 마케팅 용어로는 존재할지 몰라도, 기술적으로 네트워크 레벨에서 **이를 보장하는 것은 거짓말이거나 분산 시스템을 오해하고 있는 것**입니다.

3. 현실적인 해결책: "가짜(Fake) Exactly-once 만들기"

물리적으로는 메시지가 두 번(이상) 전송되었지만, 결과적으로는 한 번만 처리된 것처럼 보이게 하기.

- **공식:**

  > At-least-once Delivery (최소 1회 전송) + Idempotency (멱등성) = Exactly-once Processing (정확히 1회 처리)

- **방법:**
  - 메시지 큐는 중복이 되든 말든 일단 확실하게 보내줍니다(At-least-once).
  - 받는 쪽(Worker)에서 **중복 제거(Deduplication)** 로직을 통해 이미 처리한 ID면 무시합니다.
  - 또는, 연산 자체가 멱등하도록 만듭니다. (예: `x = x + 1` 대신 `x = 5`로 설정하는 방식)

엄밀히 말하면 "네트워크 레벨의 중복 전송(At-least-once)"을 "애플리케이션 레벨의 방어 로직(Idempotency)"으로 덮어씌워서, 최종 사용자나 데이터베이스 입장에서는 마치 사고가 없었던 것처럼 눈속임(Illusion)을 하는 것입니다.
이 과정이 어떻게 일어나는지 **"1,000원 결제 알림"** 시나리오로 아주 상세하게 분해해 드리겠습니다.

시나리오: 1,000원 결제 알림 (메시지 ID: `msg_100`)

상황: 워커(Worker)가 메시지 큐에서 알림 요청을 가져와 처리를 시도합니다.

1. 첫 번째 시도 (물리적 전송 1회) -> 성공했으나 "오해" 발생

1. **전송 (Delivery):** 큐가 워커에게 `msg_100`을 보냅니다.
2. **검사 (Check):** 워커는 DB/Redis를 봅니다. `msg_100` 처리 기록이 **없습니다.**
3. **처리 (Process):**
    - 사용자에게 푸시 알림 발송 🚀
    - DB/Redis에 `msg_100: 처리완료` 저장 💾
4. **사고 발생 (Failure):** 워커가 큐에게 "나 다 했어(ACK)"라고 신호를 보내는데, **네트워크 오류로 이 신호가 큐에 도달하지 못하고 사라집니다.**
5. **큐의 오해:** 큐는 "어? 워커가 응답이 없네? 처리하다 죽었나보다. 다시 보내야지."라고 판단합니다.

> 현황:
>
> - 실제 결과: 알림 발송됨 (1회), DB 저장됨.
> - 큐의 인식: 실패함.

2. 두 번째 시도 (물리적 전송 2회 - 중복!) -> "방어" 발동

1. **재전송 (Redelivery):** 큐가 일정 시간 뒤 다른 워커(혹은 같은 워커)에게 **또** `msg_100`을 보냅니다. **(여기서 At-least-once 발생)**
2. **검사 (Check - 멱등성 로직):**
    - 워커는 다시 받은 `msg_100`을 들고 DB/Redis를 봅니다.
    - **발견!** "잠깐, `msg_100`은 이미 `처리완료` 상태네?"
3. **스킵 (Skip - Fake 처리):**
    - **중요:** 워커는 알림 발송 로직을 **실행하지 않습니다.** (Business Logic Skipped)
    - 아무 일도 안 했지만, 마치 성공한 것처럼 행동합니다.
4. **확인 사살 (ACK):**
    - 워커는 큐에게 즉시 "나 다 했어(ACK)"를 보냅니다. (이미 처리된 건이니 큐에서 지워도 된다는 뜻)
5. **종료:** 큐는 메시지를 삭제합니다.

결과 비교: "무엇이 Fake인가?"

이 과정을 표로 비교해보면 왜 "Fake Exactly-once"인지 명확해집니다.

관점	실제로 일어난 일	최종 인식	비고
네트워크 (전송)	메시지가 **2번** 왔다갔다 함	**2회 전송** (At-least-once)	물리적 현실
비즈니스 로직 (처리)	첫 번째만 실행, 두 번째는 **스킵**	**1회 처리** (Exactly-once)	**우리가 만든 환상**
데이터베이스 (상태)	기록 1회 (`INSERT` 후 `IGNORE`)	**1회 저장**	결과적 일관성
사용자 (경험)	알림 1번 받음	**1회 수신**	만족

핵심: "결과적 멱등성 (Effective Idempotency)"

결국 "Fake Exactly-once"의 정체는 **"전송 횟수(N번)와 상관없이, 시스템의 상태 변화(State Change)는 딱 1번만 일어나도록 강제하는 것"**입니다.

요약 및 결론

이 글은 **"시스템이 알아서 딱 한 번만 보내주겠지?"라는 환상을 버리라**고 말합니다.
우리가 앞서 설계했던 **Redis를 이용한 중복 방지 로직**이 바로 이 글에서 말하는 "At-least-once 위에서 Idempotency를 구현하여 Exactly-once를 흉내 내는 방식"입니다. 엔지니어로서 아주 정석적인 접근을 하신 겁니다.