4장 - 처리율 제한 장치의 설계 (Rate Limiter)

들어가기

- 네트워크 시스템에서 처리율 제한 장치(rate limiter)는

  클라이언트 또는 **서비스가 보내는 트래픽의 처리율(rate)을 제어하기 위한 장치**다.

  - HTTP를 예로 들면 특정 기간 내에 전송되는 클라이언트의 요청 횟수를 제한한다.
- API 요청 횟수가 제한 장치에 정의된 임계치(threshold)를 넘어서면

  추가로 도달하는 모든 호출은 처리가 중단(block)된다.

사용 예시

- 사용자는 초당 2회 이상 새 글을 올릴 수 없다.
- 같은 IP 주소로는 하루에 10개 이상의 계정을 생성할 수 없다.
- 같은 디바이스로는 주당 5회 이상 리워드(reward)를 요청할 수 없다.

API에 처리율 제한 장치를 두면 좋은 점

▼ **Dos(Denial of Service) 공격에 의한 자원 고갈(resource starvation)을 방지**

- Rate-limiting strategies and techniques - 참고
- 대형 IT 기업들이 공개한 거의 대부분의 API는 어떤 형태로든 처리율 제한 장치를 갖고 있다.
- ex) 트위터는 3시간 동안 300개의 트윗만 올릴 수 있도록 제한,

  구글 독스 API는 사용자당 분당 300회의 read 요청만 허용.
- 추가 요청에 대한 처리를 중단함으로서 Dos 공격을 방지한다.

▼ **비용 절감**

- 처리를 제한하면 서버를 많이 두지 않아도 되고, 우선순위가 높은 API에 더 많은 자원을 할당할 수 있다.
- 3자(third-pary) API에 사용료를 지불하고 있는 회사들에게는 아주 중요하다.
    - ex) 신용을 확인하거나, 신용카드를 결제 하거나, health check를 확인하는 API 등에 대한 과금이 횟수에 따라 이루어진다면 그 횟수를 제한해야 비용을 절감할 수 있다.

▼ **서버 과부하 방지**

- 봇에서 오는 트래픽이나 사용자의 잘못된 이용 패턴으로 유발된 트래픽을 걸러낼 수 있다.

처리율 제한 장치 설계

1단계 : 문제 이해 및 설계 범위 확정

다음 면접관과 소통하며 어떤 제한 장치를 구현해야 하는지 확인해보자.

- 처리율 제한 장치를 구현하는 데는 여러 알고리즘이 있으므로 선택해야 됨

```
지원자 : 어떤 종류의 처리율 제한 장치를 설계해야 하나요?
        클라이언트 측 제한 장치입니까? 서버 측 제한 장치 입니까?

면접관 : 좋은 질문이네요. 서버측 API를 위한 장치를 설계한다고 가정합시다.

지원자 : 어떤 기준을 사용해서 API 호출을 제어애향 할까요?
        IP 주소를 사용해야 하나요? 아니면 사용자 ID? 아니면 생각하는 다른

면접관 : 다양한 형태의 제어 규칙(throttling rules)을 정의할 수 있는 유

지원자 : 시스템 규모는 어느 정도여야 할까요?
        스타트업 정도 회사를 위한 시스템입니까? 아니면 사용자가 많은 큰 기

면접관 : 설계할 시스템은 대규모 요청을 처리할 수 있어야 합니다.

지원자 : 시스템이 분산 환경에서 동작해야 하나요?

면접관 : 그렇습니다.
```

```
지원자 : 이 처리율 제한 장치는 독립된 서비스입니까? 아니면 애플리케이션 코드

면접관 : 그 결정은 본인이 내려주시면 되겠습니다.

지원자 : 사용자의 요청이 처리율 제한 장치에 의해 걸러진 경우 사용자에게 그

면접관 : 그렇습니다.
```

위 시나리오를 보고, 요구사항을 정의하면 다음과 같다.

요구사항

- 설정된 처리율을 **초과하는 요청은 정확하게 제한**한다.
- **낮은 응답시간**
  - 이 처리율 제한 장치는 HTTP 응답 시간에 나쁜 영향을 주어서는 곤란한다.
- **가능한 한 적은 메모리**를 써야 한다.
- **분산형 처리율 제한(distributed rate limiting)**
  - 하나의 처리율 제한 장치를 여러 서버나 프로세스에 공유될 수 있어야 한다.
- 예외 처리
  - 요청이 제한되었을 때는 그 사실을 사용자에게 분명하게 보여주어야 한다.
- **높은 결함 감내성(fault tolerance)**
  - 제한 장치에 장애가 생기더라도 전체 시스템에 영향을 주어서는 안 된다.

2단계 : 개략적 설계안 제시 및 동의 구하기

기본적인 클라이언트-서버 통신 모델 사용한다. (설계의 복잡함을 줄이자)

처리율 제한 장치는 어디에 둘 것인가?

- `클라이언트`, `서버` 양 측 모두 처리율 제한 장치를 적용할 수 있다. 또는 `미들웨어`를 통해 제어 가능
- `클라이언트` 측에 뒀을 경우

    - 클라이언트 요청은 쉽게 위변조가 가능해 처리율 제한을 안정적으로 걸 수 있는 장소가 아님
    - 모든 클라이언트의 구현을 통제하는 것 어려움
- `서버` 측에 뒀을 경우

그림 4-1

- `미들웨어`를 만들어 API 서버로 가는 요청을 통제하는 것

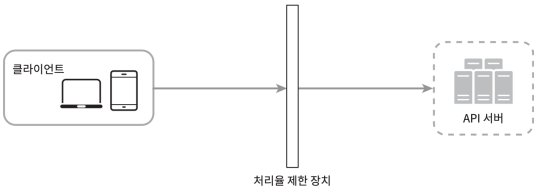

그림 4-2

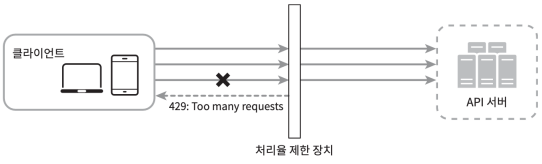

그림 4-3

왼쪽 그림이 어떻게 동작하는 설명하는 그림

API 서버의 처리율이 초당 **2개의 요청으로 제한**하면, **3번째 요청은 처리율 제한 미들웨어에 의해 가로막히고**,
클라이언트로는 HTTP 상태 코드
`429 (Too many requests)`가 반환된다.

클라우드 마이크로서비스의 경우, 처리율 제한 장치는 **보통 API 게이트웨이(gateway)라 불리는 컴포넌트에 구현**

`API 게이트웨이`는 **처리율 제한**, **SSL 종단(termination)**, **사용자 인증(authentication)**, **IP 허용 목록(whitelist) 관리** 등을 지원하는 완전 위탁관리형 서비스(**fully managed**), 즉 **클라우드 업체가 유지 보수를 담당**하는 서비스이다.

지금은 API 게이트웨이가 처리율 제한을 지원하는 미들웨어라는 점을 기억하자.

다시 돌아와서 처리율 제한을 클라이언트에 두어야 되는지 서버 측에 두어야 되는지는 **정답이 없지만**,
일반적으로 적용될 수 있는 몇 가지 지침은 다음과 같다.

- **프로그래밍 언어, 캐시 서비스 등 현재 사용하고 있는 기술 스택을 점검**

- 현재 사용하는 프로그래밍 언어가 서버 측 구현을 지원하기 충분한 지 확인

- **현재 사업 필요에 맞는 처리율 제한 알고리즘**
  - 서버 측에서 모든 것을 구현하면 알고리즘 자유, 게이트웨이를 사용한다면 선택지 제한
- **서비스가 MSA에 기반하고 있고, 사용자 인증, IP 허용목록 관리 등 API 게이트웨이를 사용 중이라면**
  - 처리율 제한 기능도 여기 포함
- **처리율 제한 서비스를 직접 만드는 데는 시간이 듬**
  - 처리율 제한 장치를 구현하기에 충분한 인력이 없다면 사용 API 게이트웨이를 쓰는 것이 좋음

널리 알려진 처리율 제한 알고리즘

▼ 토큰 버킷(token bucket)

- 처리율 제한에 폭넓게 이용되는 알고리즘이다. (간단함, 알고리즘에 대한 세간의 이해도도 높은 편)
- 아마존과 스트라이프가 API 요청을 통제(throttle)하기 위해 이 알고리즘을 사용한다.
- 토큰 버킷 알고리즘의 동작 원리

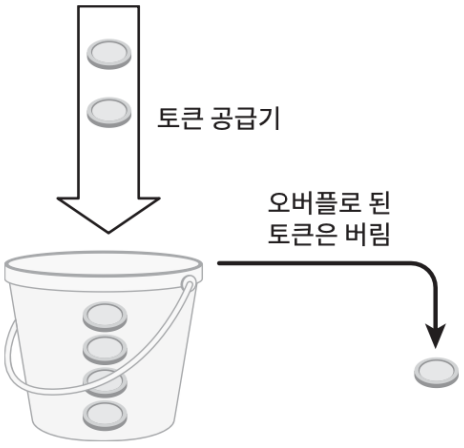

그림 4-4

용량이 4인 버킷

- 토큰 버킷은 지정된 용량을 갖는 컨테이너
  이 버킷에는 사전 설정된 양의 토큰이 주기적으로 채워진다.
- 토큰이 꽉 찬 버킷에는 더 이상 토큰은 추가되지 않는다.
- 토큰 공급기(refiller)는 이 버킷에 매초 2개의 토큰을 추가한다.
- 버킷이 가득차면 추가로 공급된 토큰은 버려(overflow)진다.

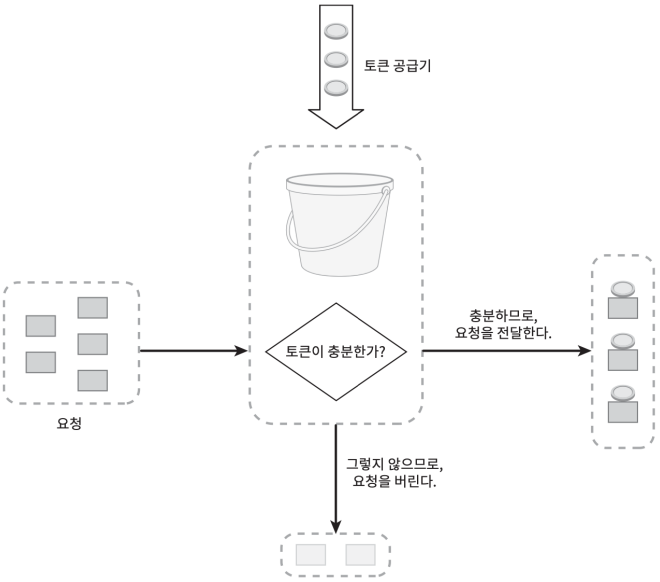

그림 4-5

- 각 요청은 처리될 때마다 하나의 토큰을 사용한다.
- 요청이 도착하면 버킷에 충분한 토큰이 있는지 검사하게 된다.
  - 토큰이 있는 경우 : 버킷에서 토큰 하나를 꺼낸 후 요청을 시스템에 전달
  - 토큰이 없는 경우 : 해당 요청을 버려(dropped)진다.

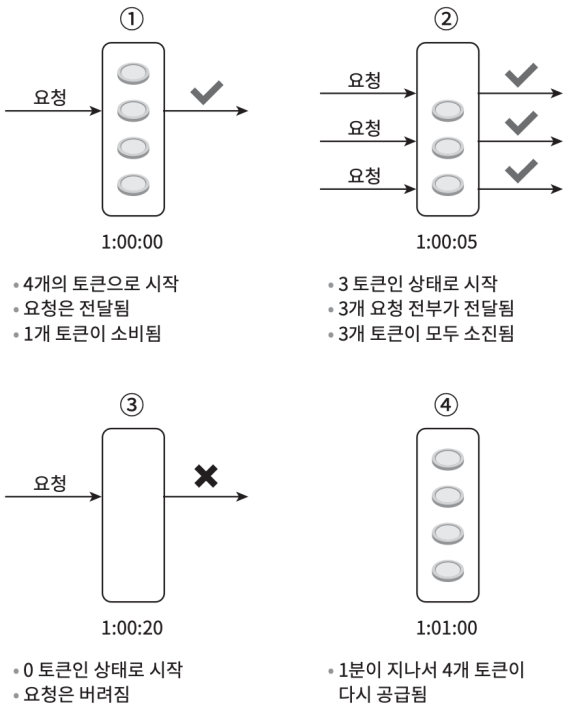

그림 4-6

- 이 예에서 토큰 `버킷의 크기`는 4 이고, `토큰 공급률(refill rate)`은 분당 4이다.
  - **버킷 크기** : 버킷에 담을 수 있는 토큰의 최대 개수
  - **토큰 공급률(refill rate)** : 초당 몇 개의 토큰이 버킷에 공급지
  - 이런 식으로 토큰 버킷 알고리즘은 **2개 인자(parameter)**를 받는다.

- 통상적으로 API 엔트포인트마다 별도의 버킷을 둔다.
  ex)
  `사용자`마다 **하루에 한 번만 포스**팅 할 수 있고, `친구`는 **150명까지 추가**할 수 있고,

  `좋아요` 버튼은 **다섯 번까지만 누를 수** 있다면 사용자마다 `3개의 버킷`을 두어야 한다.
- IP 주소별로 처리율 제한을 적용해야 한다면 IP 주소마다 버킷을 하나씩 할당해야 한다.
- 시스템의 처리율을 초당 10,000개 요청으로 제한하고 싶다면, 모든 요청이 하나의 버킷을 공유하도록

장점

- 구현이 쉽다.
- 메모리 사용 측면에서도 효율적이다.

- 짧은 시간에 집중되는 트래픽(burst of traffic)도 처리 가능하다.
  버킷에 남은 토큰이 있기만 하면 요청은 시스템에 전달

단점

- `버킷 크기`와 `토큰 공급률`이라는 두 개 인자를 가지고 있다. 이 값을 적절하게 튜닝하는 것은 까다롭다.

> API Gateway는 요청에 대해 토큰이 계산되는 토큰 버킷 알고리즘을 사용하여 API에 대한 요청을 제한합니다. 특히 API Gateway는 리전별로 계정의 모든 API에 대한 요청 제출 속도와 버스트를 검사합니다. 토큰 버킷 알고리즘에서 버스트는 이러한 제한의 사전 정의된 초과 실행을 허용할 수 있지만, 다른 요인으로도 제한을 초과할 수 있습니다.
>
> - Amazon API Gateway는 토큰 버킷 알고리즘 사용

▼ 누출 버킷(leaky bucket)

- 토큰 버킷 알고리즘과 비슷하지만 **요청 처리율이 고정**되어 있다.

▼ 이 알고리즘은 보통 FIFO(First-In-First-Out) 큐로 구현한다. (펼치면 동작 원리)

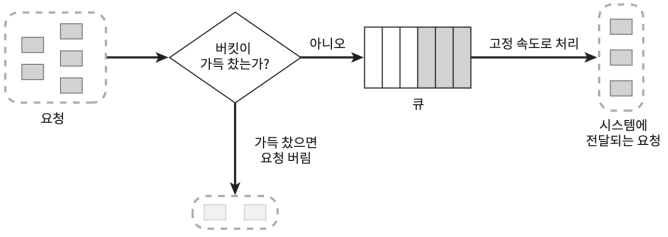

그림 4-7

  - 요청이 도착하면 큐가 가득 차 있는지 확인, 빈자리가 있다면 큐에 요청 추가
  - 큐가 가득 차 있다면 새 요청은 버림
  - 지정된 시간마다 큐에서 요청을 꺼내어 처리
- `버킷 크기`와 `처리율(outflow rate)`이라는 두 인자를 사용한다.
  - **버킷 크기** : 큐 사이즈와 같은 값. 큐에는 처리될 항목들이 보관된다.
  - **처리율** : 지정된 시간당 몇 개의 항목을 처리할지 지정하는 값. (보통 초 단위로 관리)

- 쇼피파이(Shopify)가 이 알고리즘을 사용하여 처리율 제한을 구현하고 있음

장점

- 큐의 크기가 제한되어 있어 메모리 사용량 측면에서 효율적이다.
- 고정된 처리율을 갖고 있기 때문에 안정적 출력(stable outflow rate)이 필요한 경우에 적합하다.

단점

- 단시간에 많은 트래픽이 몰리는 경우 큐에는 오래된 요청들이 쌓이게 되고,
  해당 요청들을 제때 처리하지 못하면 최신 요청들은 버려지게 되는 상황 발생.
- 토큰 버킷과 마찬가지로 `버킷 크기`와 `처리율` 두 개의 인자의 값을 올바르게 튜닝하기 까다로움

▼ 고정 윈도 카운터(fixed window counter)

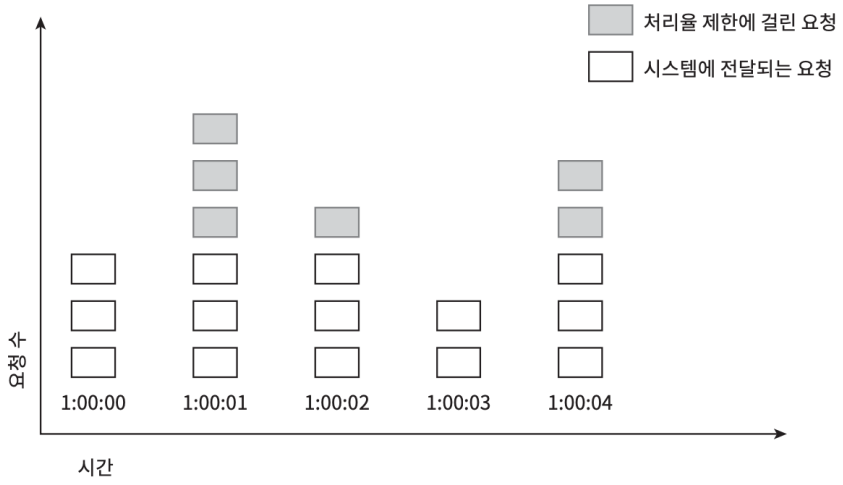

그림 4-8

- 타임라인을 고정된 간격의 윈도로 나누고, 각 윈도마다 카운터를 붙인다.
- 요청이 접수될 때마다 이 카운터 값은 1씩 증가한다.
- 이 카운터의 값이 사전에 설정된 임계칭(threshold)에 도달하면 새로운 요청은 새 윈도가 열릴 때 버려짐
- 이 알고리즘의 가장 큰 문제

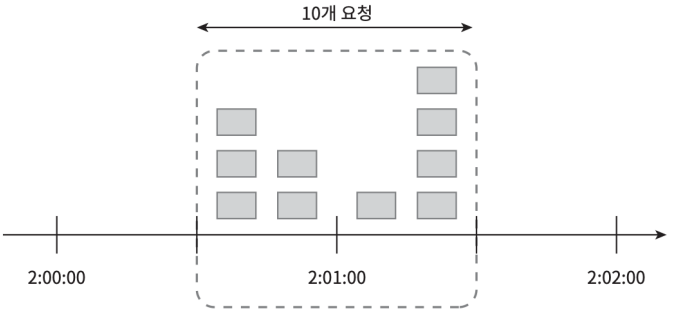

그림 4-9

- 윈도의 경계 부근에 부하가 집중되면 윈도에 할당된 양보다 더 많은 요청을 처리할 수 있게 됨
- 위 그림은 분당 5개의 요청만 허락하지만, 2:00:30 ~ 2:01:30 사이 각각 5개 5개 들어오면
  총 10개를 처리하게 된다. (허용한도의 2개가 처리 됨)

장점

- 메모리 효율이 좋다.
- 이해하기 쉽다.
- 윈도가 닫히는 시점에 카운터를 초기화하는 방식은 특정한 트래픽 패턴을 처리하기에 적합하다.

단점

- 윈도 **경계 부근에서 일시적으로 부하가 올 경우**, 기대했던 시스템의 처리 한도보다 **더 많은 양의 요청을 처리**

▼ 이동 윈도 로그(ssliding window log)

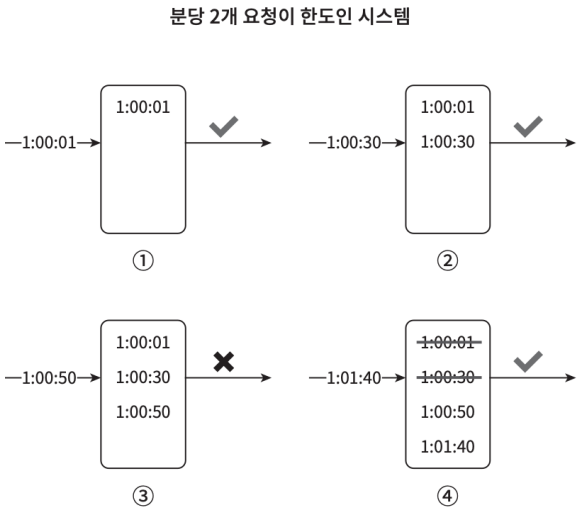

그림 4-10

분당 최대 2회의 요청만 처리하도록 설정

- **이동 윈도 로그**를 사용하면 고정 윈도 카운터를 사용했을 때 윈도 경계 부근에 트래픽이 집중될 경우 **시스템에 설정된 한도보다 더 많은 요청을 처리하는 문제를 해결**한다.
- 요청의 타임스탬프를 추적한다.

  타임스탬프 데이터는 보통 **레디스의 정렬 집합(sorted set) 같은 캐시에 보관**한다.
- 새 요청이 오면 만료된 타임스탬프는 제거한다.
  만료된 타임스탬프는 그 값이 현재 윈도의 시작 지점보다 오래된 타임스탬프를 뜻함
- 새 요청의 타임스탬프를 로그에 추가한다.
- 로그의 크기가 허용치보다 같거나 작으면 요청을 시스템에 전달한다.
  그렇지 않은 경우 처리 거부

장점

- 해당 알고리즘이 구현하는 처리율 제한 메커니즘은 아주 정교하다.
  어느 순간의 윈도를 보더라도, 허용되는 요청의 개수는 시스템의 처리율 한도를 넘지 않는다.

단점

- 다량의 메모리를 사용한다. (거부된 요청의 타임스탬프도 보관하기 때문)

▼ 이동 윈도 카운터(ssliding window counter)

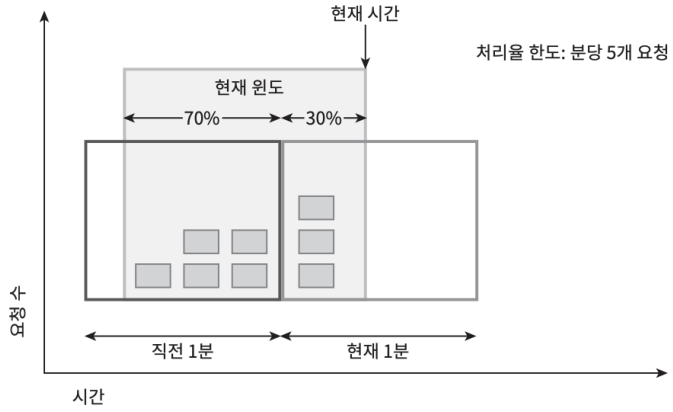

그림 4-11

- `고정 윈도 카운터 알고리즘`과 `이동 윈도 로깅 알고리즘`을 결합한 상태
- 이 알고리즘을 구현하는 방식은 두 가지이고, 한 가지만 다룬다. (다른 하나는 여기 참고)
- 현재 1분의 30% 시점에 도착한 새 요청의 경우, 현재 윈도에 몇 개의 요청이 온 것으로 보고 처리하는 계산
  - 현재 1군간의 요청 수 + 직전 1분간의 요청 수 * 이동 윈도와 직전 1분이 겹치는 비율
  - 이 공식에 따라 현재 윈되에 들어 있는 요청은 3 + 5 * 70% = 6.5개이다. 반올림해서 쓸 수도 있고 내림해서 쓸 수도 있다. (책에선 내림으로 사용)

장점

- 이전 시간대의 평균 처리율에 따라 현재 윈도의 상태를 계산하므로 짧은 시간에 몰리는 트래픽에도 잘 대응
- 메모리 효율이 좋음

단점

- 직전 시간대에 도착한 요청이 균등하게 분포되어 있다고 가정한 상태로 추정치를 계산하기 때문에 느슨하다.

  그렇다고, 느슨한 정도가 심각한 정도의 수준은 아니다.

  클라우드플레어(Cloudflare)가 실시했던 실험에 따르면 40억 개의 요청 가운데, 시스템의 실제 상태와 맞지 않게 허용되거나 버려진 요청은 0.003%에 불과

개략적인 아키텍처

- 얼마나 많은 요청이 접수되었는지를 추적할 수 있는 카운터를 `추적 대상별`로 두고, 이
  `카운터의 값`이 어떤 한도를 넘어서면 한도를 넘어 도착한 요청은 거부하면 된다.
  - 추정 대상 : `사용자별로?`, `IP 주소별로?`, `API 엔드포인트?`, `서비스 단위로?`
- `카운터`는 어디에 보관하면 좋을까?
  - 데이터베이스는 디스크 접근 때문에 속도가 느리므로 **메모리상에서 동작하는 캐시가 유리**하다.
    - 빠르고, 시간에 기반한 캐시 만료 정책을 지원한다.
    - Redis `INCR`과 `EXPIRE`의 두 가지 명령어 지원

      `INCR` : 메모리에 저장된 카운터의 값을 1 증가

      `EXPIRE` : 카운터에 타임아웃 값을 설정 (설정된 시간이 지나면 자동 삭제)

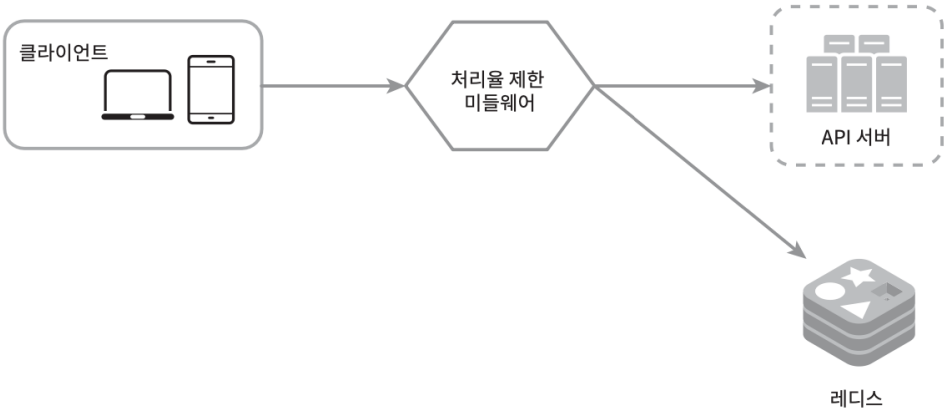

그림 4-12

- 클라이언트가 처리율 제한 미들웨어에게 요청을 보낸다.

- 처리율 제한 미들웨어는 레디스의 지정 버킷에서 카운터를 가져와 한도에 도달했는지 검사한다.
  - 한도에 도달했다면 : 요청 거부
  - 도달하지 않았다면 : 요청은 API 서버로 전달. (미들웨어는 카운터 값을 증가 시킨 후 레디스에 저장)

위 개략적 설계에서까지 왔을 때 해결하지 못한 부분

- `처리율 제한 규칙`은 어떻게 만들어지고 어디에 저장되는가?
- 처리가 제한된 요청들은 어떻게 처리되는가?

처리율 제한 규칙

리프트(Lyft)는 처리율 제한에 오픈 소스를 사용하고 있다.
이 컴포넌트를 들여다보고, 어떤 처리율 제한 규칙이 사용되고 있는지 살펴보자.

```
--- 마케팅 메시지의 최대치를 하루 5개로 제한
domain: messaging
descriptors:
    - key: message_type
      value: marketing
      rate_limit:
          unit: day
          requests_per_unit: 5

--- 클라이언트가 분당 5회 이상 로그인 할 수 없도록 제한
domain: messaging
descriptors:
    - key: message_type
      value: marketing
      rate_limit:
          unit: day
          requests_per_unit: 5
```

- 이런 규칙들은 보통 설정 파일(configuration file) 형태로 디스크에 저장

처리율 한도 초과 트래픽 처리

한도 제한에 걸리면 API는 HTTP 429 응답(too many requests)을 클라이언트에 반환
경우에 따라 한도 제한에 걸린 메시지를 나중에 처리하기 위해 큐에 보관하기도 함

> 처리율 제한 장치가 사용하는 **HTTP 헤더**

클라이언트는 자신의 요청이 처리율 제한에 걸리는지 어떻게 감지할 수 있을까?
또한 처리율 제한에 걸리기까지 얼마나 많은 요청을 보냈는지 어떻게 알 수 있을까?

HTTP 응답 헤더(response header)를 통해 해당하는 정보들을 확인할 수 있다.
다음(`3단계 : 상세 설계`)에서 설계할 처리율 제한 장치는 **다음 HTTP 헤더**를 클라이언트에게 보낸다.

- X-Ratelimit-Remaining: 윈도 내에 남은 처리 가능 요청의 수.
- X-Ratelimit-Limit: 매 윈도마다 클라이언트가 전송할 수 있는 요청의 수.
- X-Ratelimit-Retry-After: 한도 제한에 걸리지 않으려면 몇 초 뒤 요청을 다시 보내야 하는지 알림
    - 사용자가 너무 많은 요청을 보내면 `429 too many requests` 오류를 해당하는 헤더와 함께 반환

3단계 : 상세 설계

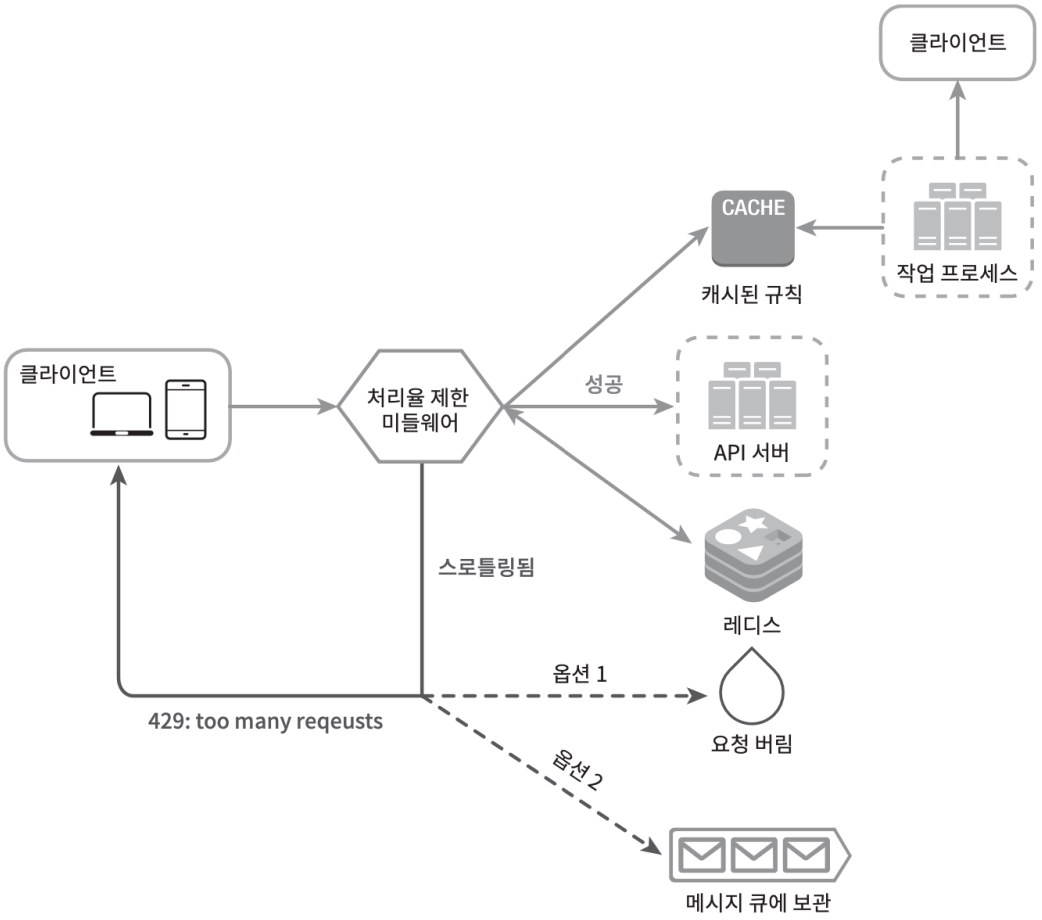

그림 4-13

- 처리율 제한 규칙은 디스크에 보관한다.
  작업 프로세스(worker)는 수시로 규칙을 디스크에서 읽어 캐시에 저장한다.
- 클라이언트가 요청을 서버에 보내면 요청은 먼저 처리율 제한 미들웨어에 도달한다.
- 처리율 제한 미들웨어는 제한 규칙을 캐시에서 가져온다.또한 카운터 및 마지막 요청의 타임스탬프를
  레디스 캐시에서 가져온다. 가져온 값들에 근거하여 미들웨어는 다음과 같은 결정을 내린다.
  - 해당 요청이 처리율 제한에 걸리지 않은 경우에는 API 서버로 보낸다.
  - 해당 요청이 처리율 제한에 걸렸다면 429 too many requests 에러를 클라이언트에 보낸다.
    또한 이 요청은 그대로 버릴 수도 있고, 메시지 큐에 보관할 수도 있다.

분산 환경에서 처리율 제한 장치의 구현

- 단일 서버를 지원하는 처리율 제한 장치를 구현하는 것은 어렵지 않다.
  하지만 여러 대의 서버와 병렬 스레드를 지원하도록 시스템을 확장하는 것은 다음 두 가지 문제를 풀어야 한다.
  1. **경쟁 조건(race condition)**
  2. **동기화(synchronization)**

경쟁 조건(race condition)

처리율 제한 장치는 대략 다음과 같이 동작한다.

1. 레디스에서 카운터의 값을 읽는다. (counter)
2. counter + 1의 값이 임계치를 넘는지 본다.
3. 넘지 않는다면 레디스에 보관된 카운터 값을 1만큼 증가시킨다.

이러한 중에 **동시에 여러 요청이 오는 상황**이 심할 경우 다음 그림과 같은 **경쟁 조건 이슈가 발생**한다.

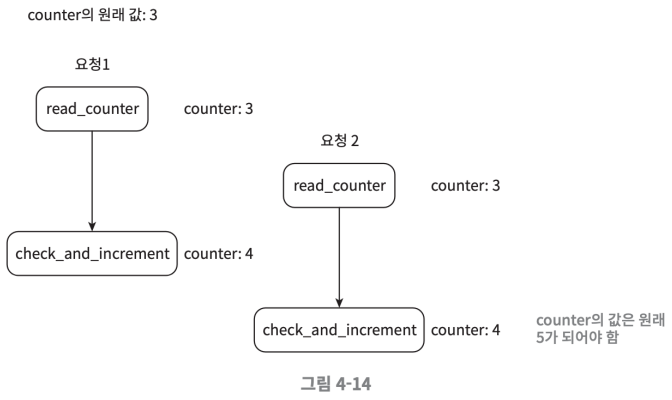

그림 4-14

- 위 그림처럼 두 개 요청을 처리하는 스레드가 각각 **병렬로 counter 값을 읽고**, 그 둘 가운데 어느 쪽도 아직 변경된 값을 저장하지 않은 상태(위 그림에서는 counter의 값이 두 스레드 모두 3인 상태)인 상황
  두 스레드의 처리가 완료되면 counter 값이 `한 쪽은 4` , `한 쪽은 5`가 되어야 하는데, `각각 4`가 된다.
- 이런 경쟁 조건 문제를 해결하는 **가장 널리 알려진 해결책은 락(lock)**이다.
  - 하지만 락은 시스템의 성능을 상당히 떨어뜨린다.
  - 위 설계의 경우 락 대신 쓸 수 있는 해결책이 두 가지가 존재한다.
    1. **루아 스크립트(Lua script) 사용**

2. **정렬 집합(sorted set) 자료구조 사용**

동기화(synchronization)

수백만 사용자를 지원하려면 한 대의 처리율 제한 장치 서버로는 충분하지 않다.
그래서 처리율 제한 장치
서버를 여러 대 두게 되면 동기화가 필요해진다.

- ex) 클라이언트 1은 제한 장치 1에 요청, 2는 제한 장치 2에 요청을 보내는 상황

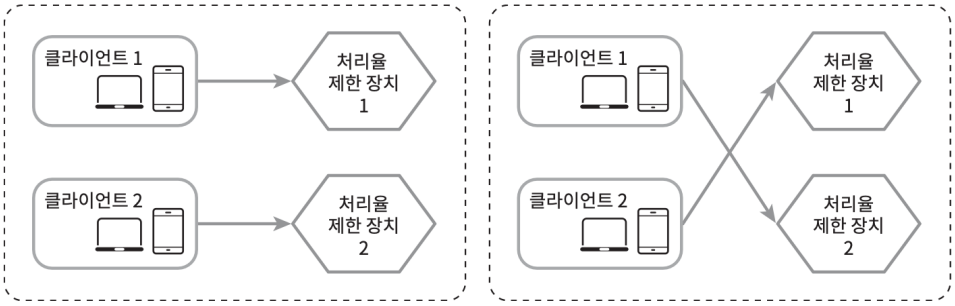

그림 4-15

웹 계층은 무상태(stateless)이므로 클라이언트는 다음 요청을 각기 다른 제한 장치로 보내게 될 수 있음
이때 동기화를 하지 않는다면 제한 장치 1은 클라이언트 2에 대해 아무것도 모르므로 처리율 제한을 올바르게 수행 할 수 없게 된다.

- 다른 방식에서 언급했듯 이때 한 가지 해결책으로는 고정 세션(sticky session)을 활용할 수 있다.
  같은 클라이언트로부터 요청은 항상 같은 처리율 제한 장치로 보내는 방법. (저자는 추천하지 않음)
- 다른 해결책으로는 다음 그림처럼 레디스와 같은 **중앙 집중형 데이터 저장소를 사용**하는 것이다.

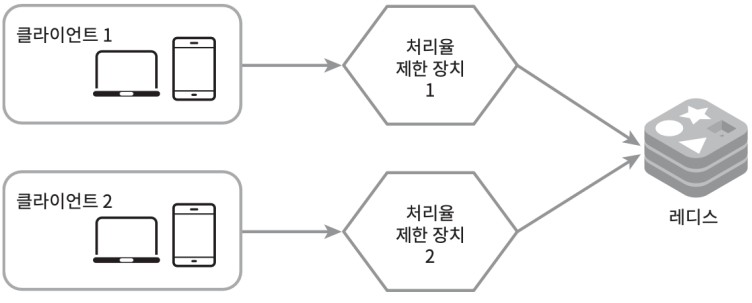

그림 4-16

성능 최적화

지금까지 살펴본 설계는 두 가지 지점에서 개선이 가능하다.

1. **데이터센터의 물리적인 거리를 좁혀 지연시간(latency)를 줄이는 방식**

   대부분의 클라우드 서비스 사업자는 세계 곳곳에 엣지 서버(edge server)를 심어놓고 있다.
   ex) 2020년 5월 20일
   클라우드플레어는 지역으로 분산된 194곳의 위치에 엣지 서버를 설치한 상태
   사용자의 트래픽을 가장 가까운 엣지 서버로 전달하여 지연시간을 줄인다.

2. 제한 장치 간에 **데이터 동기화할 때 최종 일관성 모델(eventual consistency model)을 사용하는 것**이다.
   - 이 일관성 모델이 생소하다면 <6장 키-값 저장소 설계-데이터 일관성>을 참고하자.

모니터링

기본적으로 모니터링을 통해 확인하려는 것은 다음 두 가지다.

1. 채택된 처리율 제한 아록리즘이 효과적인지
2. 정의한 처리율 제한 규칙이 효과적인지

ex) 처리율 제한 규칙이 빡빡하게 설정되었다면 많은 유효 요청이 버려지게 된다. 이럴땐 규칙을 완화해야 된다.

4단계 : 마무리

처리율 제한을 구현하기 위해 다뤘던 알고리즘

- 토큰 버킷
- 누출 버킷
- 고정 윈도 카운터
- 이동 윈도 로그

- 이동 윈도 카운터

추가적으로 살펴보면 도움이 될 만한 것들

- **경성(hard) 또는 연성(soft) 처리율 제한**
  - 경성 처리율 제한 : 요청의 개수는 임계치를 절대 넘어설 수 없다.
  - 연성 처리율 제한 : 요청 개수는 잠시 동안은 임계치를 넘어설 수 있다.
- **다양항 계층에서의 처리율 제한**
  - 이번 장에서는 애플리케이션 계층(HTTP: OSI 7L - 7계층)에서의 처리율 제한에 대해서만 살펴보았다.
    하지만 다른 계층에서도 처리율 제한이 가능하다.
    ex)
    Iptables를 사용하면 IP 주소(3계층)에 처리율 제한을 적용하는 법은 여기를 참고하자.
- 처리율 제한을 회피하는 방법. 클라이언트를 어떻게 설계하는 것이 최선인가?
  - 클라이언트 측 캐시를 사용하여 API 호출 횟수를 줄인다.
  - 처리율 제한의 임계치를 이해하고, 짧은 시간 동안 너무 많은 메시지를 보내지 않도록
  - 예외나 에러를 처리하는 코드를 도입하여 클라이언트가 예외적 상황으로 부터 우아하게 복구될 수 있도록
  - 재시도(retry) 로직을 구현할 때는 충분한 백오프(back-off) 시간을 둔다.

Chapter 4: DESIGN A RATE LIMITER

1. Rate-limiting strategies and techniques
2. Twitter Rate limits
3. Google Docs Usage limits
4. IBM Microservices
5. Throttle API requests for better throughput
6. Stripe rate limiters

7. Shopify API rate limits
8. Better Rate Limiting With Redis Sorted Sets
9. System Design — Rate limiter and Data modelling
10. How we built rate limiting capable of scaling to millions of domains
11. Redis website
12. Lyft rate limiting
13. Scaling your API with rate limiters
14. What is edge computing?
15. Rate Limit Requests with Iptables
16. OSI model