4장. 처리율 제한 장치의 설계

처리율 제한 장치(rate limiter)

클라이언트 또는 서비스가 보내는 트래픽의 처리율을 제어하기 위한 장치

- 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.

처리율 제한 장치의 장점

- DOS 공격에 의한 자원 고갈을 방지할 수 있다.
- 비용을 절감한다. 서버를 많이 두지 않아도 되고, 우선순위가 높은 다른 API에 더 많은 자원을 할당할 수 있다.
- 서버 과부하를 막는다. 봇에서 오는 트래픽이나 사용자의 잘못된 이용 패턴으로 유발된 트래픽을 걸러낼 수 있다.

1단계 | 문제 이해 및 설계 범위 확정

- **처리율 제한의 주체**는 어디인가? 클라이언트/서버
- **어떠한 것을 기준으로 제한**을 할 것인가? IP 주소, 사용자 ID ..
- **시스템 규모**? 스타트업 규모/대규모 시스템
- **분산 환경**에서 동작?
- **독립된 환경**에서 동작? 애플리케이션 코드에 포함?

책에서 제시한 요구사항

- 설정된 처리율을 초과하는 요청은 정확하게 제한한다.
- 낮은 응답시간: 이 처율 제한 장치는 HTTP 응답시간에 나쁜 영향을 주어서는 곤란하다.
- 가능한 한 적은 메모리를 써야 한다.
- 분산형 처리율 제한: 하나의 처리율 제한 장치를 여러 서버나 프로세스에서 공유할 수 있어야 한다.
- 예외 처리: 요청이 제한되었을 때는 그 사실을 사용자에게 분명히 보여주어야 한다.
- 높은 경험 감내성: 제한 장치에 장애가 생기더라도 전체 시스템에 영향을 주어서는 안된다.

2단계 | 개략적 설계안 제시 및 동의 구하기

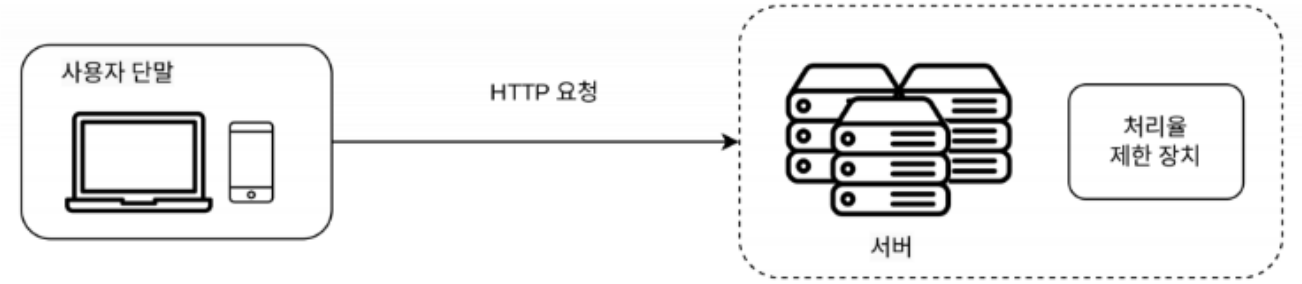
일단 일을 너무 복잡하게 만드는 것을 피하고, 기본적인 **클라이언트-서버 통신 모델** 사용

제한 장치는 어디에 둘 것인가?

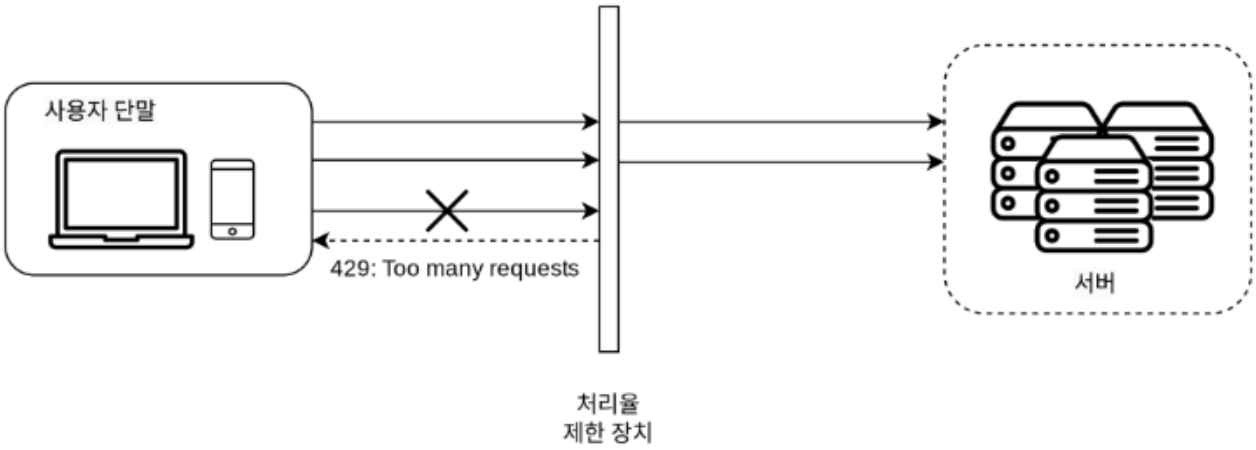
1. 클라이언트

클라이언트 요청은 **쉽게 위변조가 가능**하기 때문에, 일반적으로 클라이언트는 처리율 제한을 안정적으로 걸 수 있는 장소 X

2. 서버



3. 미들웨어



- 클라우드 기반 MSA의 경우, 처리율 제한 장치는 보통 API 게이트 웨이라 불리는 컴포넌트에 구현

어떤 방법을 선택해야 하는가?

회사의 현재 기술 스택, 엔지니어링 인력, 우선 순위, 목표에 따라 달라질 수 있다.
아래는 일반적으로 적용될 수 있는 지침

- 프로그래밍 언어, 캐시 서비스 등 현재 사용하고 있는 기술 스택을 점검하라. 현재 사용하는 프로그래밍 언어가 서버측 구현을 지원하기 충분할 정도로 효율이 높은지 확인하라.
- 여러분의 사업 필요에 맞는 처리율 제한 알고리즘을 찾아라.
- 여러분의 설계가 MSA에 기반하고 있고, 사용자 인증이나 IP 허용 목록 관리 등을 처리하기 위해 API 게이트웨이를 이미 설계에 포함시켰다면 처리율 제한 기능 또한 게이트웨이에 포함시켜야할 수 있다.

- 처리율 제한 서비스를 직접 만드는 데는 시간이 든다. 처리율 제한 장치를 구현하기에 충분한 인력이 없다면 사용 API 게이트웨이를 쓰는 것이 바람직한 방법일 것이다.

처리율 제한 알고리즘

널리 알려진 알고리즘은 다음과 같다.

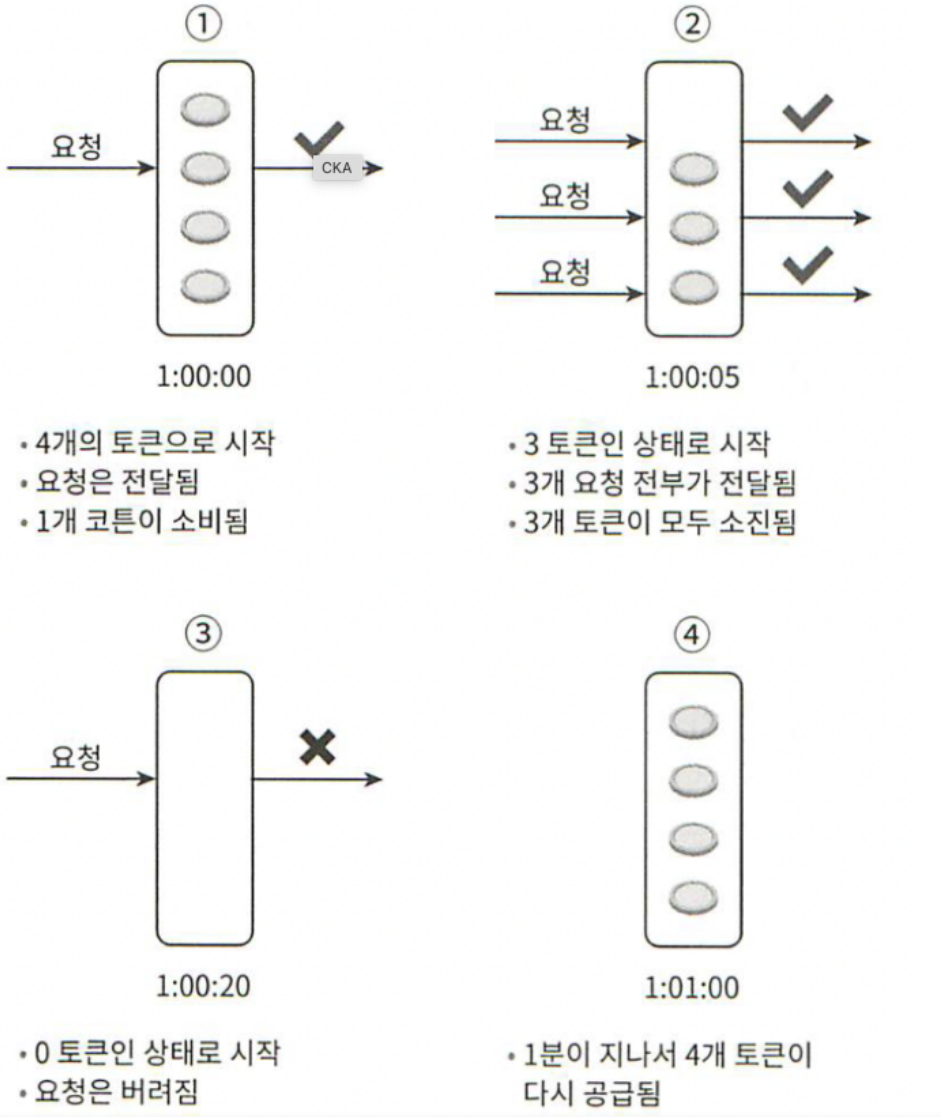
- 토큰 버킷 (token bucket)
- 누출 버킷 (leaky bucket)
- 고정 윈도 카운터 (fixed window counter)
- 이동 윈도 로그 (sliding window log)
- 이동 윈도 카운터 (sliding window counter)

토큰 버킷 알고리즘

토큰 버킷 알고리즘은 처리율 제한에 폭넓게 이용되고 있다.
간단하고, 알고리즘에 대한 세간의 이해도도 높은 편이며 인터넷 기업들(아마존 등)이 보편적으로 사용하고 있다.

동작 원리

- 토큰 버킷은 지정된 용량을 갖는 컨테이너다. 이 버킷에는 사전 설정된 양의 토큰이 주기적으로 채워진다. 토큰이 꽉 찬 버킷에는 더 이상의 토큰은 추가되지 않는다. 버킷이 가득 차면 추가로 공급된 토큰은 버려진다.
- 충분한 토큰이 있는 경우, 버킷에서 토큰 하나를 꺼낸 후 요청을 시스템에 전달한다. 충분한 토큰이 없는 경우, 해당 요청은 버려진다.



- 토큰 버킷 알고리즘은 2개 인자를 받는다.
  - **버킷 크기**: 버킷에 담을 수 있는 토큰의 최대 개수
  - **토큰 공급률(refill rate)**: 초당 몇 개의 토큰이 버킷에 공급되는가

버킷을 몇 개나 사용해야 하나?

공급 제한 규칙에 따라 달라진다. 다음 사례를 보자.

- 통상적으로, API 엔드포인트마다 별도의 버킷을 둔다 예를 들어, 사용자마다 하루에 한 번만 포스팅을 할 수 있고, 친구는 150명까지 추가할 수 있고, 좋아요 버튼은 다섯 번까지만 누를 수 있다면, 사용자마다 3개의 버킷을 두어야 할 것이다.
- 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 해야할 것이다.

장점

- 구현이 쉽다.
- 메모리 사용 측면에서도 효율적이다.

- 짧은 시간에 집중되는 트래픽도 처리 가능하다. 버킷에 남은 토큰이 있기만 하면 요청은 시스템에 전달될 것이다.

단점

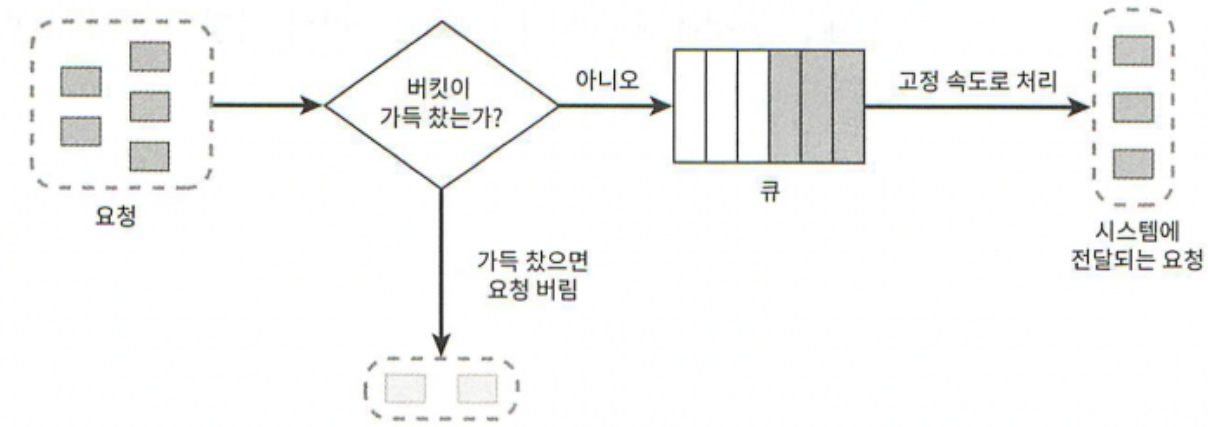
- 이 알고리즘은 버킷 크기와 토큰 공급률이라는 두 개 인자를 가지고 있는데, 이 값을 적절하게 튜닝하는 것이 까다롭다.

누출 버킷 알고리즘

토큰 버킷 알고리즘과 비슷하지만, 요청 처리율이 고정되어 있다는 점이 다르다. 누출 버킷 알고리즘은 보통 FIFO큐로 구현된다. 쇼피파이가 사용하는 알고리즘이다.

동작 과정

- 요청이 도착하면 큐가 가득 차 있는지 본다. 빈자리가 있는 경우에는 큐에 요청을 추가한다.
- 큐가 가득 차 있는 경우 새 요청은 버린다.
- 지정된 시간마다 큐에서 요청을 꺼내어 처리한다.



- **버킷 크기**: 큐 사이즈와 같은 값이다. 큐에는 처리될 항목들이 보관된다.
- **처리율(outflow rate)**: 지정된 시간당 몇 개의 항목을 처리할지 지정하는 값이다. 보통 초 단위로 표현된다.

장점

- 큐의 크기가 제한되어 있어 메모리 사용량 측면에서 효율적이다.
- 고정된 처리율을 갖고 있기 때문에 안정적 출력이 필요한 경우 적합하다.

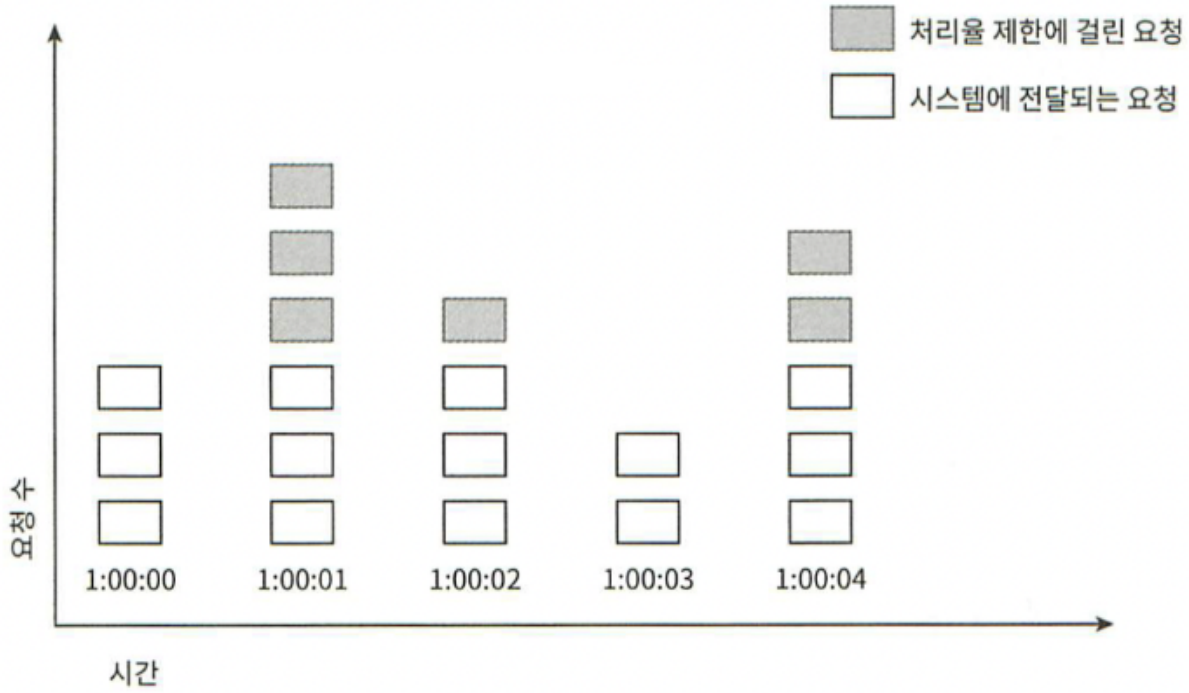
단점

- 단시간에 많은 트래픽이 몰리는 경우 큐에는 오래된 요청들이 쌓이게 되고, 그 요청들을 제때 처리하지 못하면 최신 요청들은 버려지게 된다.
- 두 개 인자를 갖고 있는데, 이들을 올바르게 튜닝하기가 까다로울 수 있다.

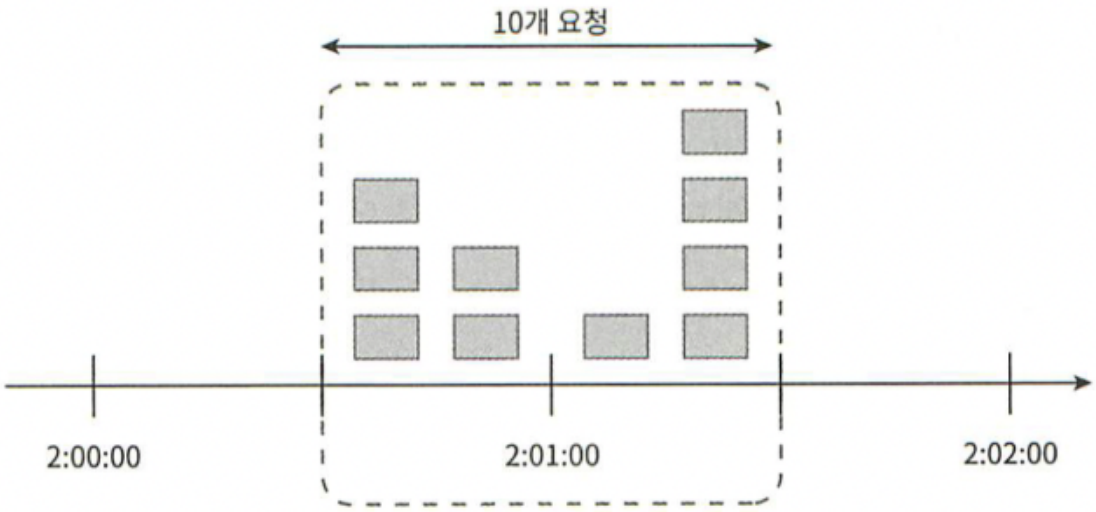
고정 윈도 카운터 알고리즘

동작 과정

- 타임라인을 고정된 가격의 윈도로 나누고, 각 윈도마다 카운터를 붙인다.
- 요청을 접수될 때마다 이 **카운터의 값은 1씩 증가**한다.
- 이 카운터의 값이 사전에 설정된 임계치에 도달하면 새로운 요청은 새 윈도가 열릴 때까지 버려진다.



- 타임라인의 시간 단위는 1초
- 시스템은 초당 3개까지의 요청만을 허용



- 윈도 경계 부근에 순간적으로 많은 트래픽이 집중될 경우 윈도에 할당된 양보다 더 많은 요청이 처리될 수 있다는 문제가 있다.

장점

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

단점

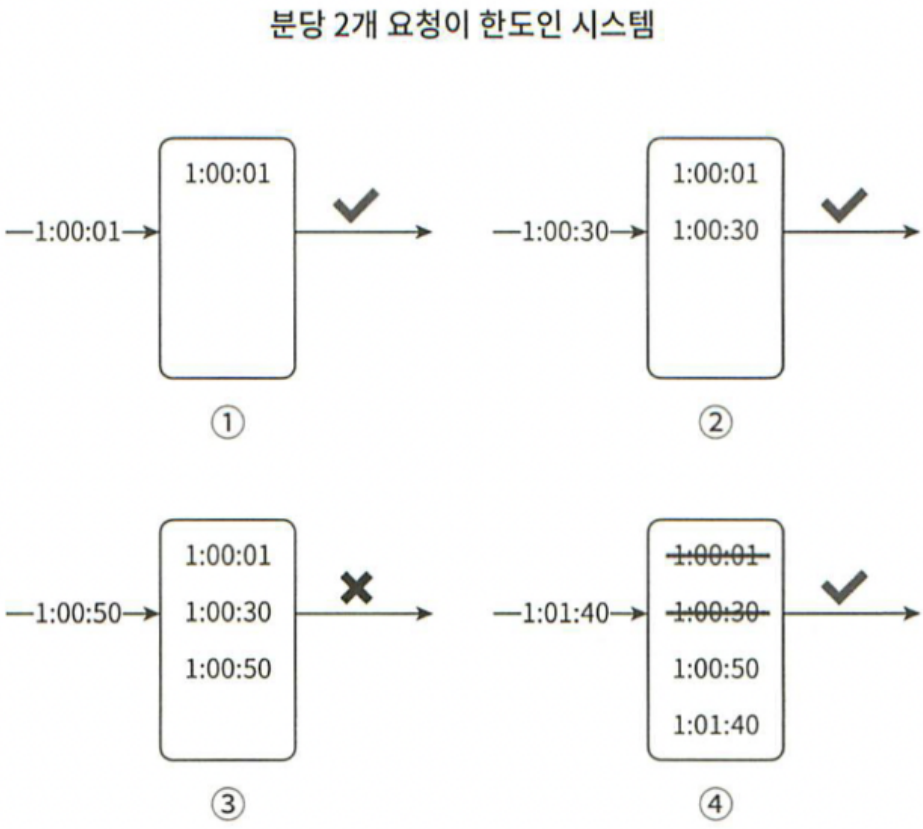
- 윈도 경계 부근에서 일시적으로 많은 트래픽이 몰려드는 경우, 기대했던 시스템의 처리 한도보다 많은 양의 요청을 처리하게 된다.

이동 윈도 로깅 알고리즘

고정 윈도 카운터 알고리즘에서 트래픽이 집중되는 경우 시스템에 설정된 한도보다 많은 요청을 처리하는 **단점을 보완한 알고리즘**이다.

동작 원리

- 이 알고리즘은 요청의 타임스탬프를 추적한다. 타임스탬프 데이터는 보통 레디스의 정렬 집합 같은 캐시에 보관한다.
- 새 요청이 오면 만료된 타임스탬프는 제거한다. 만료된 타임스탬프는 그 값이 현재 윈도의 시작 시점보다 오래된 타임스탬프를 말한다.
- 새 요청의 타임스탬프를 로그에 추가한다.
- 로그의 크기가 허용치보다 같거나 작으면 요청을 시스템에 전달한다. 그렇지 않은 경우에는 처리를 거부한다.



- 요청이 1:00:01에 도착했을 때, 로그는 비어 있는 상태다. 따라서 요청은 허용된다.

- 새로운 요청이 1:00:30에 도착한다. 해당 타임스탬프가 로그에 추가된다. 추가 직후 로그의 크기는 2이며, 허용 한도보다 크지 않은 값이다. 따라서 요청은 시스템에 전달된다.
- 새로운 요청이 1:00:50에 도착한다. 해당 타임스탬프가 로그에 추가된다. 추가 직후 로그의 크기는 3으로, 허용 한도보다 큰 값이다. 따라서 타임스탬프는 로그에 남지만 요청은 거부된다.
- 새로운 요청이 1:01:40에 도착한다. 1:00:40 ~ 1:01:40 범위 안에 있는 요청은 1분 윈도 안에 있는 요청이지만, 1:00:40 이전의 타임스탬프는 전부 만료된 값이다. 따라서 두 개의 만료된 타임스탬프 1:00:01과 1:00:30을 로그에서 삭제한다. 삭제 직후 로그의 크기는 2이다. 따라서 1:01:40의 신규 요청은 시스템에 전달된다.

장점

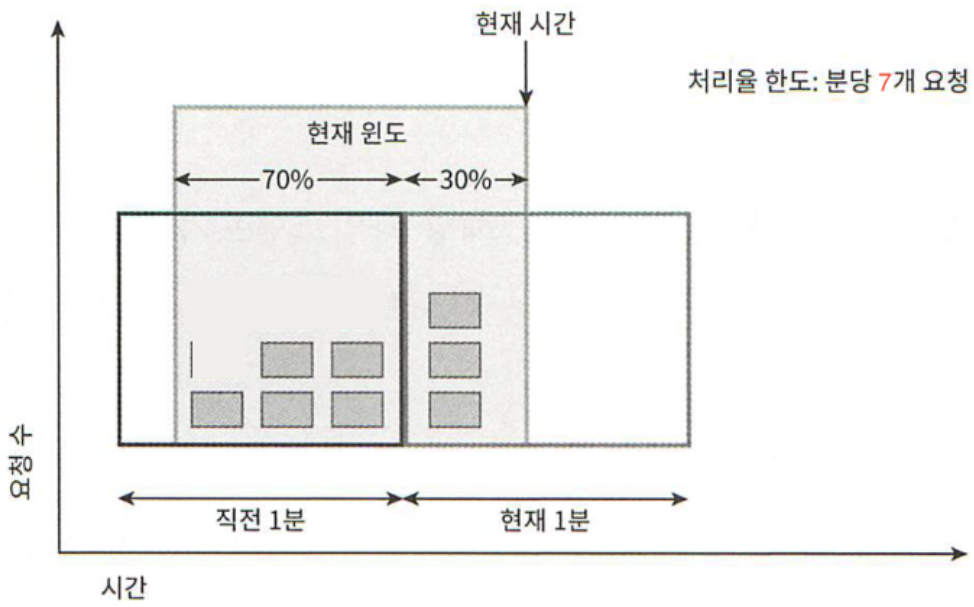
- 이 알고리즘이 구현하는 처리율 제한 메커니즘은 아주 정교하다. 어느 순간의 윈도우를 보더라도, 허용되는 요청의 개수는 시스템의 처리율 한도를 넘지 않는다.

단점

- 이 알고리즘은 다량의 메모리를 사용하는데, 거부된 요청의 타임 스탬프도 보관하기 때문이다.

이동 윈도 카운터 알고리즘

고정 윈도 카운터 알고리즘과 이동 윈도 로깅 알고리즘을 결합한 것이다.



처리율 제한 장치의 한도가 분당 7개 요청으로 설정되어 있다고 하고, 이전 1분 동안 5개의 요청이, 그리고 현재 1분 동안 3개의 요청이 왔다고 해 보자.
현재 1분의 30% 시점에 도착한 새 요청의 경우, 현재 윈도에 몇 개의 요청이 온것으로 보고 처리해야 할까? 다음과 같이 계산한다.

- 현재 1분간의 요청 수 + 직전 1분간의 요청 수 X 이동 윈도와 직전 1분의 겹치는 비율

이 공식에 따르면 현재 윈도에 들어 있는 요청은 3 + 5 X 70% = 6.5개다. 반올림해서 쓸 수도 있고 내림하여 쓸 수도 있는데, 본 예제에서는 내림하여 쓰겠다. 따라서 그 값은 6이다.
본 예제의 경우 처리율 제한 한도가 분당 7개 요청이라고 했으므로, 현재 1분의 30% 시점에 도착한 신규 요청은 시스템으로 전달될 것이다. 하지만 그 직후에는 한도에 도달하였으므로 더 이상의 요청은 받을 수 없을 것이다.

장점

- 이전 시간대의 평균 처리율에 따라 현재 윈도의 상태를 계산하므로 짧은 시간에 몰리는 트래픽에도 잘 대응한다.
- 메모리 효율이 좋다.

단점

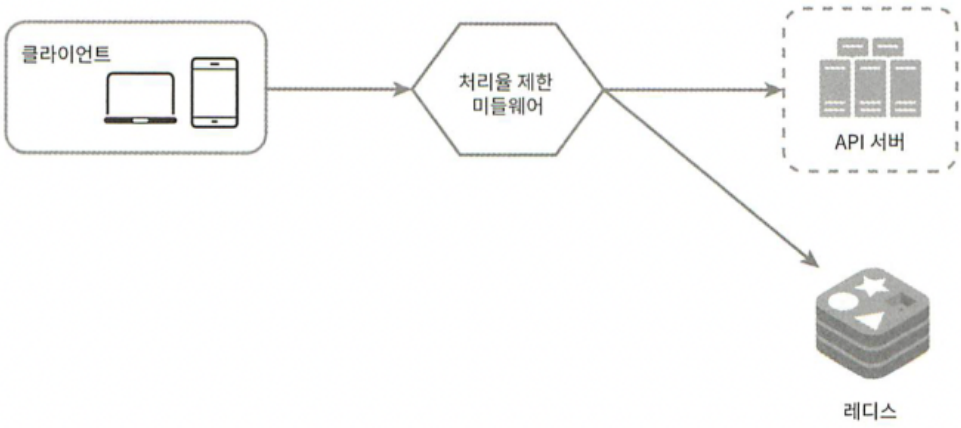
- 직전 시간대에 도착한 요청이 균등하게 분포되어 있다고 가정한 상태에서 추정치를 계산하기 때문에 다소 느슨하다. 하지만 이 문제는 생각만큼 심각한게 아닌데, 클라우드플레어가 실시했던 실험에 따르면 40억 개의 요청 가운데 시스템의 실제 상태와 맞지 않게 허용되거나 버려진 요청은 0.003%에 불과하였다.

개략적 아키텍처

카운터 추적 보관

데이터베이스는 디스크 접근 때문 느림 → 메모리상에서 동작하는 캐시가 바람직
→ **Redis**: 메모리 기반 저장 장치로서, **INCR**, **EXPIRE** 2가지 명령어를 지원

- **INCR** : 메모리에 저장된 카운터의 값을 1만큼 증가시킨다.
- **EXPIRE** : 카운터에 타임아웃 값을 설정한다. 설정된 시간이 지나면 카운터는 자동으로 삭제 된다.



- 클라이언트가 처리율 제한 미들웨어에게 요청을 보낸다.
- 처리율 제한 미들웨어는 레디스의 지정 버킷에서 카운터를 가져와서 한도에 도달했는지 검사한다.
    - 한도에 도달했다면 요청은 거부

- 한도에 도달하지 않았다면 요청은 API 서버로 전달 + 미들웨어는 카운터의 값을 증가시킨 후 다시 레디스에 저장

3단계 | 상세 설계

처리율 제한 규칙

리프트(Lyft)는 처리율 제한 오픈소스를 사용

```
domain : messaging
descriptors :
      - key : message_type
        value : marketing
        rate_limit :
            unit : day
            requests_per_unit: 5
```

- 시스템이 처리할 수 있는 마케팅 메시지의 최대치를 하루 5개로 제한

```
domain : auth
descriptors :
      - key : auth_type
        value : login
        rate_limit :
            unit : minute
            requests_per_unit: 5
```

- 클라이언트가 분당 5회 이상 로그인 할 수 없도록 제한

이런 규칙들은 보통 설정 파일 형태로 디스크에 저장된다.

처리율 한도 초과 트래픽의 처리

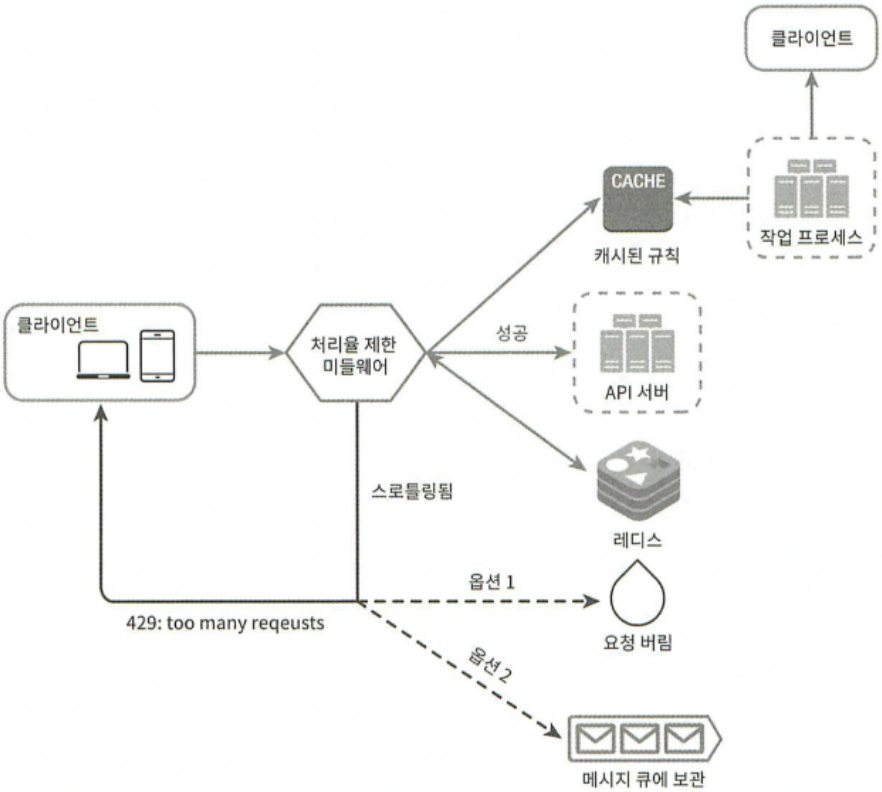
어떤 요청이 한도 제한에 걸리면 API는 HTTP 429 응답(too many requests)을 클라이언트에게 보낸다.
경우에 따라서는 한도 제한에 걸린 메시지를 나중에 처리하기 위해 큐에 보관할 수도 있다.

처리율 제한 장치가 사용하는 HTTP 헤더

클라이언트는 **HTTP 응답 헤더**를 통해 처리율 제한에 걸리기까지 얼마나 많은 요청을 보낼 수 있는지 어떻게 알 수 있다.
처리율 제한 장치는 다음의 HTTP 헤더를 클라이언트에게 보낼 수 있다.

- 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 뒤에 요청을 다시 보내야 하는지 알림
  사용자가 너무 많은 요청을 보내면 429 too many requests 오류를 X-Ratelimit-Retry-After 헤더와 함께 반환하도록 한다.

상세 설계



- 처리율 제한 규칙은 디스크에 보관한다. 작업 프로세스는 수시로 규칙을 디스크에서 읽어 캐시에 저장한다.
- 클라이언트가 요청을 서버에 보내면 요청은 먼저 처리율 제한 미들웨어에 도달한다.
- 처리율 제한 미들웨어는 제한 규칙을 캐시에서 가져온다. 아울러 카운터 및 마지막 요청의 타임스탬프를 레디스 캐시에서 가져온다. 가져온 값들에 근거하여 해당 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. 한편 해당 요청은 그대로 버릴 수도 있고 메시지 큐에 보관할 수 있다.

분산 환경에서의 처리율 제한 장치의 구현

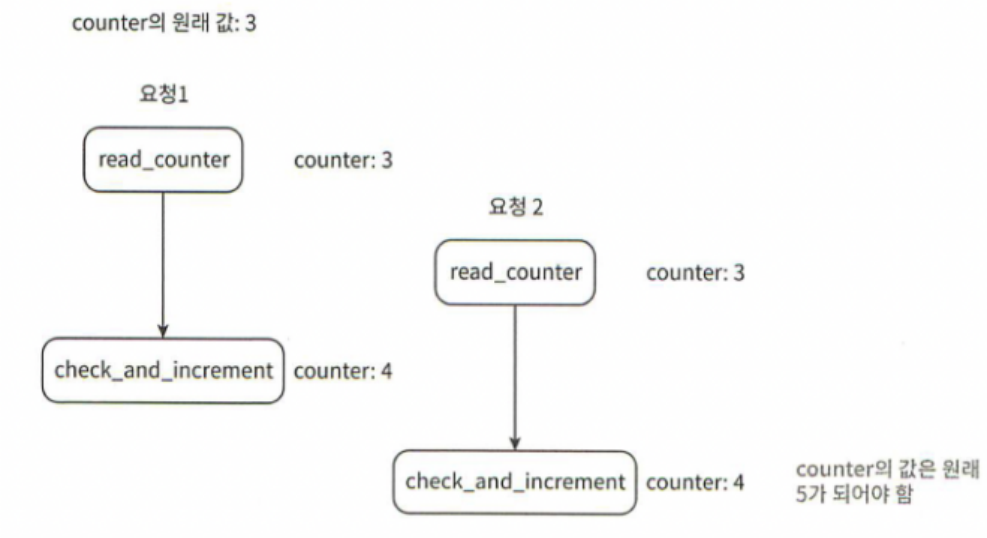
여러 대의 서버와 병렬 스레드를 지원하도록 시스템을 확장하려면 다음 두 가지 어려운 문제를 풀어야 한다.

- 경쟁 조건(race condition)
- 동기화(synchronization)

경쟁 조건

처리율 제한 장치는 대략 다음과 같이 동작한다.

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

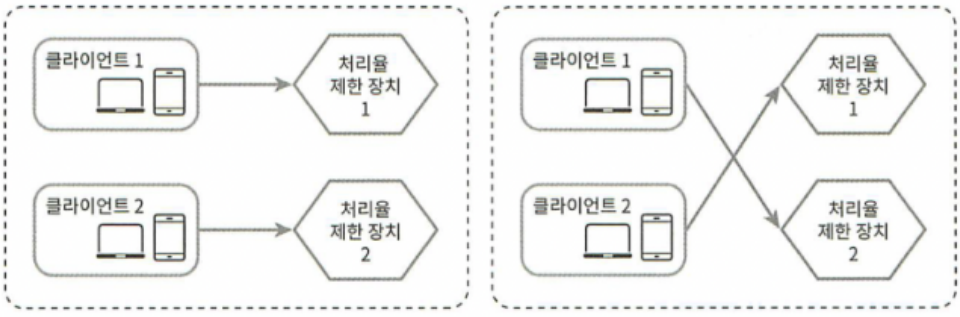


해결 방법

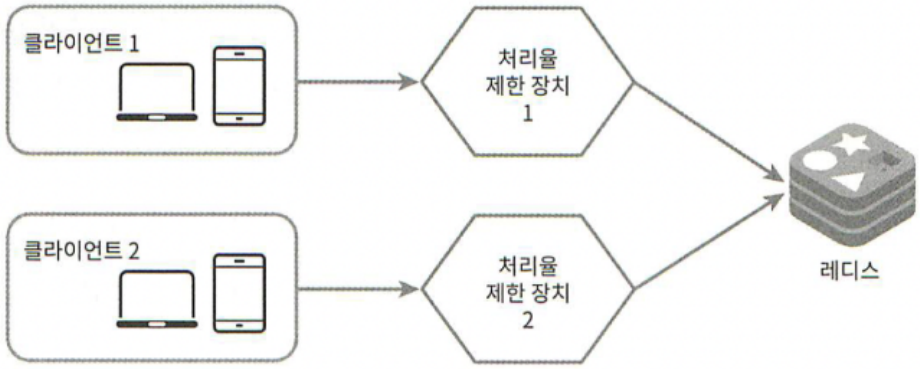
- 락(lock) → 가장 널리 알려진 해결책이지만, 시스템의 성능을 상당히 떨어뜨린다는 문제
- 루아 스크립트(Lua script)
- 레디스 자료구조 - 정렬집합(sorted set)

동기화 이슈

처리율 제한 장치를 여러 대 두게 되면 동기화가 필요



- 고정 세션으로 해결할 수 있지만, 유연하지 않은 방법이라 비추



- 레디스와 같은 **중앙 집중형 데이터 저장소** 사용

성능 최적화

1. 사용자의 트래픽을 가장 가까운 **엣지 서버로 전달**하여 지연시간 Down
2. 제한 장치 간에 데이터를 동기화할 때 **최종 일관성 모델**을 사용

모니터링

처리율 제한 장치를 설치한 이후에는 효과적으로 동작하고 있는지 보기 위해 데이터를 모을 필요가 있다.
기본적으로 모니터링을 통해 확인하려는 것은 다음 두 가지이다.

- 채택된 처리율 제한 알고리즘이 효과적이다.
- 정의한 처리율 제한 규칙이 효과적이다.
  모니터링을 통해 유동적으로 조절할 필요가 있다.

4단계 | 마무리

경성(hard) 또는 연성(soft) 처리율 제한

- 경성 처리율 제한 : 요청의 개수는 임계치를 절대 넘어설 수 없다.
- 연성 처리율 제한 : 요청 개수는 잠시 동안만 임계치를 넘어설 수 있다.

다양한 계층에서의 처리율 제한

지금 까지의 내용은 애플리케이션 계층(OSI 7계층)에서의 처리율 제한에 대해서만 살펴보았다.
하지만 다른 계층에서도 처리율 제한이 가능 → Iptables를 사용하면 IP 주소(OSI 3계층)에 처리율 제한을 적용하는 것이 가능

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 한다.
- 예외나 에러를 처리하는 코드를 도입하여 클라이언트가 예외적 상황으로부터 우아하게 (gracefully) 복구될 수 있도록 한다.
- 재시도 로직을 구현할 때는 충분한 백오프(back-off) 시간을 둔다.