4️⃣ 처리율 제한 장치의 설계

이 장을 시작하기 전에..

지난 달 면접 준비할 때 접해봤던 토픽이다. 그런데 그 당시에는 다른 할 것도 너무 많았고 난이도도 높아서 쉽게 와닿지 못했다. 그래도 이번에는 한번 접해봤다고 체감 난이도는 지난번보다는 나아진 것 같다. 다음번 면접에서는 대규모나 분산 질문에 제대로 답해야겠다.

처리율 제한 장치(rate limit)

- 클라이언트 또는 서비스가 보내는 트래픽의 처리율(rate)을 제어하기 위한 장치
- API 요청 횟수가 제한 장치에 의해 정의된 임계치를 넘어서면 이후 요청은 모두 중단(block)됩니다.

처리율 제한 장치를 두면 좋은 점

- **DoS**(Denial of Service) 공격에 의한 자원 고갈을 방지할 수 있다. 추가 요청에 대해서는 처리를 중단시킴으로 DoS 공격 방지 가능
- 비용 절감 → 서버를 많이 두지 않아도 되며 우선순위가 높은 API에 더 많은 자원을 할당할 수 있습니다. 제 3자 API에 사용료응 지불하고 있는 회사들에게는 아주 중요
- 서버 과부하 막음 → 잘못된 이용 패턴으로 접근하는 경우 트래픽 걸러내기 가능

'면접 : 처리율 제한 장치를 구현(설계)해보세요.'에 대한 질문에 4단계로 나눠 답해보자. (위 방법은 3장에서의 설계 면접 공략법의 순서와 유사함)

1단계) 문제 이해 및 설계 범위 확정

면접관은 짧은 질문을 던졌지만 지원자는 여러 사항을 추가 질문으로 내세워야 한다.

가능한 질문

- 처리율 제한 장치를 클리이언트, 서버 중 어떤 종류로 설계해야하는가?
- 어떤 기준(IP주소, 사용자 ID 등)을 이용해서 API 호출을 제어해야하나요?
- 시스템의 규모는 어느정도인가? (스타트업 규모? 대기업 규모?)
- 시스템이 분산 환경에서 동작해야하는가?
- 처리율 장치는 독립된 서비스인가? 어플리케이션 코드에 포함될 수도 있는가?
- 처리율 제한에 의해 거절된 경우 사용자에게 알려야하는가?

위 대화를 통해 얻은 요구사항 (가정)

- 설정된 처리율을 초과하는 요청은 정확하게 제한
- 낮은 응답시간: 이 처리율 제한 장치는 HTTP 응답시간에 나쁜 영향을 주어서는 안된다.
- 가능한 적은 메모리를 써야 한다.
- **분산형 처리율 제한(distributed rate limiting)**: 하나의 처리율 제한 장치를 여러 서버나 프로세스에서 공유할 수 있어야 한다.
- 예외처리: 요청이 제한되었을 때 그 사실을 사용자에게 분명하게 보여주어야 한다.
- **높은 결함 감내성(fault tolerance)**: 제한 장치에 장애가 생기더라도 전체 시스템에 영향을 주어서는 안된다.

2단계) 개략적 설계안 제시 및 동의 구하기

우선 복잡한 설계가 아닌 기본적인 클라이언트-서버 통신 모델을 사용하자.

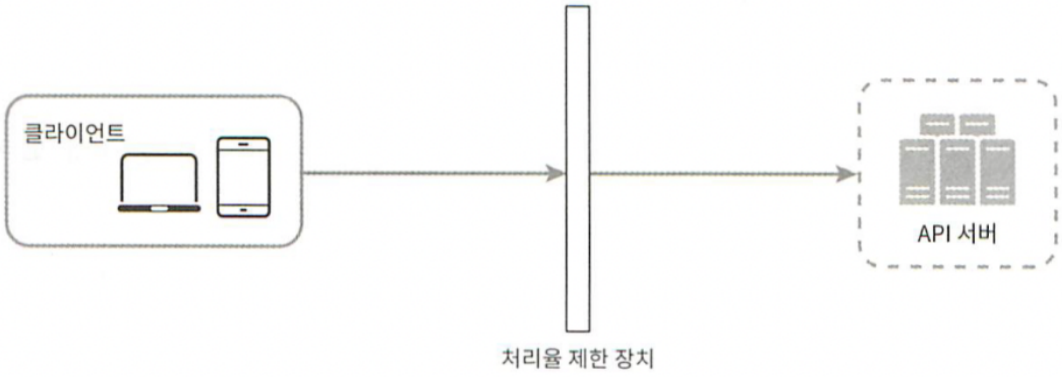
처리율 제한 장치는 어디에 둘 것인가?

- **클라이언트 측에 둔다.**

  클라이언트 요청은 쉽게 위변조가 가능해 안정적으로 처리율 제한을 두기 힘들다.

- **서버 측에 둔다.**
  - 처리율 장치를 API 서버와 동일한 곳에 둔다.

- 처리율 제한 **미들웨어**를 만들어 해당 미들웨어에서 API 서버로 가는 요청을 통제한다.



  - 마이크로서비스의 경우 → API 게이트웨이(처리율 제한을 지원하는 미들웨어)라는 컴포넌트에 구현된다.

서버와 게이트웨이 중에서 어디에 두어야하는지는 정해진 답은 없다. 회사 기술 스택에 따라 달라진다. 다만 고려해볼 점은 있다.

- 제3 사업자가 제공하는 게이트웨이를 사용한다면 위 선택지에 제한이 생길 수 있다.
- 마이크로서비스 기반이고 API 게이트웨이를 이미 설계에 포함되었다면 체리율 제한 기능 또한 게이트웨이에 포함할 수 있어야 한다.
- 처리율 제한 서비스를 직접 만드는 것은 많은 비용이 발생한다. 인력이 부족하다면 상용 API 게이트웨이를 쓰는 것이 좋다.

처리율 제한 알고리즘

처리율 제한 알고리즘에는 여러개가 있다. 각각은 장단점을 갖고 있으며 5개의 알고리즘에 대해 대략적으로 알아보겠습니다.

1. **토큰 버킷 알고리즘 (token bucket)**

그림 4-5

- 토큰 버킷은 지정된 용량을 갖는 컨테이너입니다. 이 버킷에는 사전 설정된 양의 토큰이 주기적으로 채워지며 토큰이 꽉 찬 버킷에는 더 이상의 토큰이 추가되지 않는다.
- 각 요청은 처리될 때마다 하나의 토큰을 사용한다. 요청이 도착하면 버킷에 충분한 토큰이 있는지 검사하게 되며
  - 충분히 있다면 → 버킷에서 토큰 하나를 꺼낸 후 요청을 시스템에 전달한다.
  - 충분하지 않다면 → 해당 요청은 버려진다.

토큰 버킷 알고리즘 인자

- 버킷 크기 : 버킷에 담을 수 있는 토큰의 최대 갯수

- 토큰 공급률(refill rate) : 초당 몇 개의 토큰이 버킷에 공급되는가

버킷은 몇개나 사용해야 할까?

- 통상적으로는 API 엔드포인트마다 별도의 버킷을 둔다.
- IP 주소별로 처리율 제한을 적용
- 시스템 처리율을 초당 10,000개의 요청으로 제한하고 싶다면 모든 요청이 하나의 버킷을 공유하도록 해야한다.

장점

- 구현이 쉽다
- 메모리 사용 측면에서 효율적이다
- 짧은 시간에 집중되는 트래픽도 처리 가능 (버킷에 남은 토큰이 있기만 하다면)

단점

- 버킷 크기, 토큰 공급률 2개의 인자를 갖고 있어 이 값을 적절히 **튜닝하는 것이 힘듦**

2. **누출 버킷 알고리즘 (leaky bucket)**

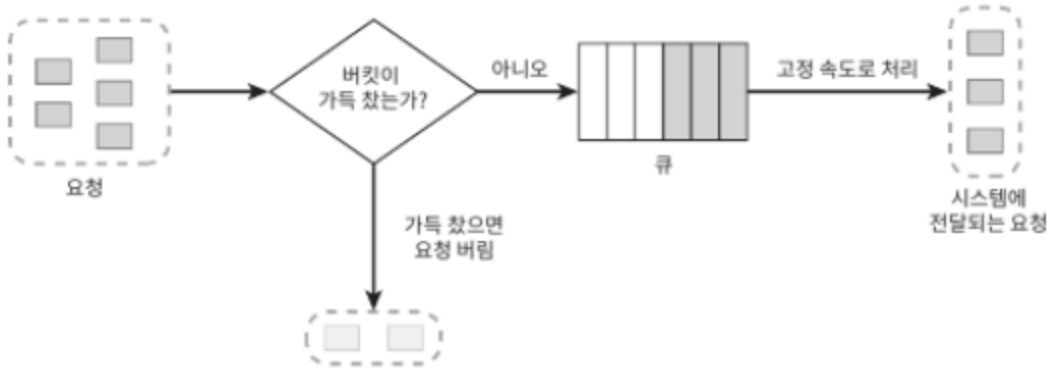


그림 4-7

- 토큰 버킷 알고리즘과 유사하지만 요청 처리율이 고정되어 있다는 점이 다른다.
- 주로 FIFO 큐로 구현한다.
    - 큐가 가득 차있는지 확인

        - 빈자리가 있다면 → 큐에 요청 추가
        - 빈자리가 없다면 → 새 요청은 버린다
    - 지정된 시간마다 큐에서 요청을 꺼내 처리한다.

누출 버킷 알고리즘 인자

- 버킷 크기 : 큐 사이즈와 같은 값, 큐에는 처리될 항목들이 보관된다.
- 처리율 (outflow rate) : 지정된 시간당 몇 개의 항목을 처리할지 지정하는 값

장점

- 큐의 크기가 제한되어 있어 메모리 사용향 측면에서 효율적
- 고정된 처리율을 갖고 있기 때문에 안정적 출력(stable output rate)가 필요한 경우 적합

단점

- 단시간에 많은 트래픽이 몰려 큐에는 오래된 요청들이 쌓이게 되고, 그 요청들을 제때 처리 못하면 최신 요청들은 버려지게 된다.
- 두 개의 인자를 갖고 있어 올바르게 **튜닝하기 까다롭다.**

3. **고정 윈도 카운터 알고리즘 (fixed window counter)**

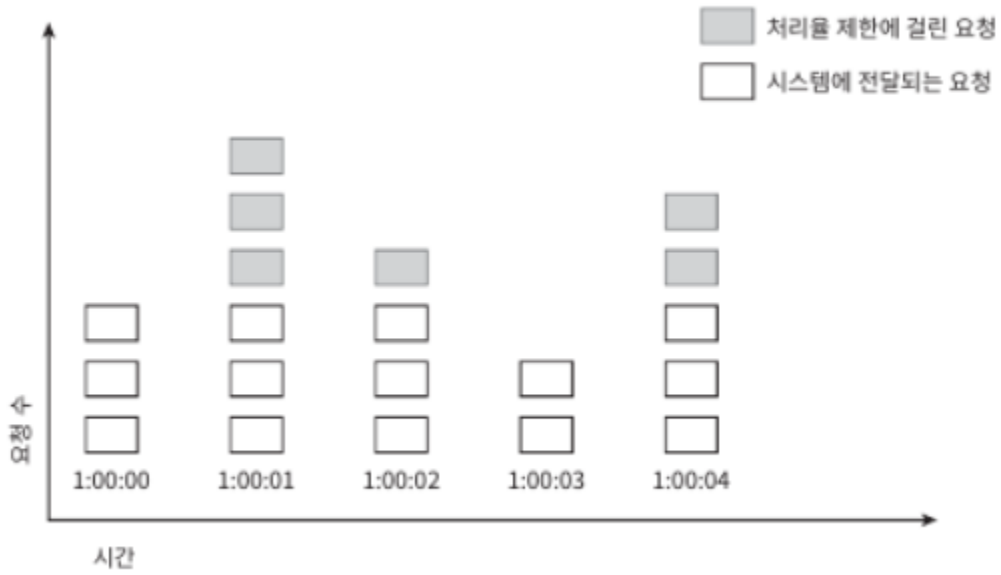


그림 4-8

동작 원리

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

원도 위치를 막 바꿔도 되나..? (p.62 상단글에 대한 의문..) + 혹시 모든 순간의 1분전을 다 파악해야하나?

장점

- 메모리 효율이 좋다
- 이해하기 쉽다
- 윈도가 닫히는 시점에 카운터를 초기화하는 방식은 특정한 트래픽 패턴을 처리하는데 적합

단점

- 윈도 경계 부근에서 일시적으로 많은 트래픽이 몰려드는 경우, 기대했던 시스템의 처리 한도보다 많은 양의 요청을 처리하게 된다. → 사실상 가장 큰 문제

4. **이동 윈도 로그 알고리즘 (sliding window log)**

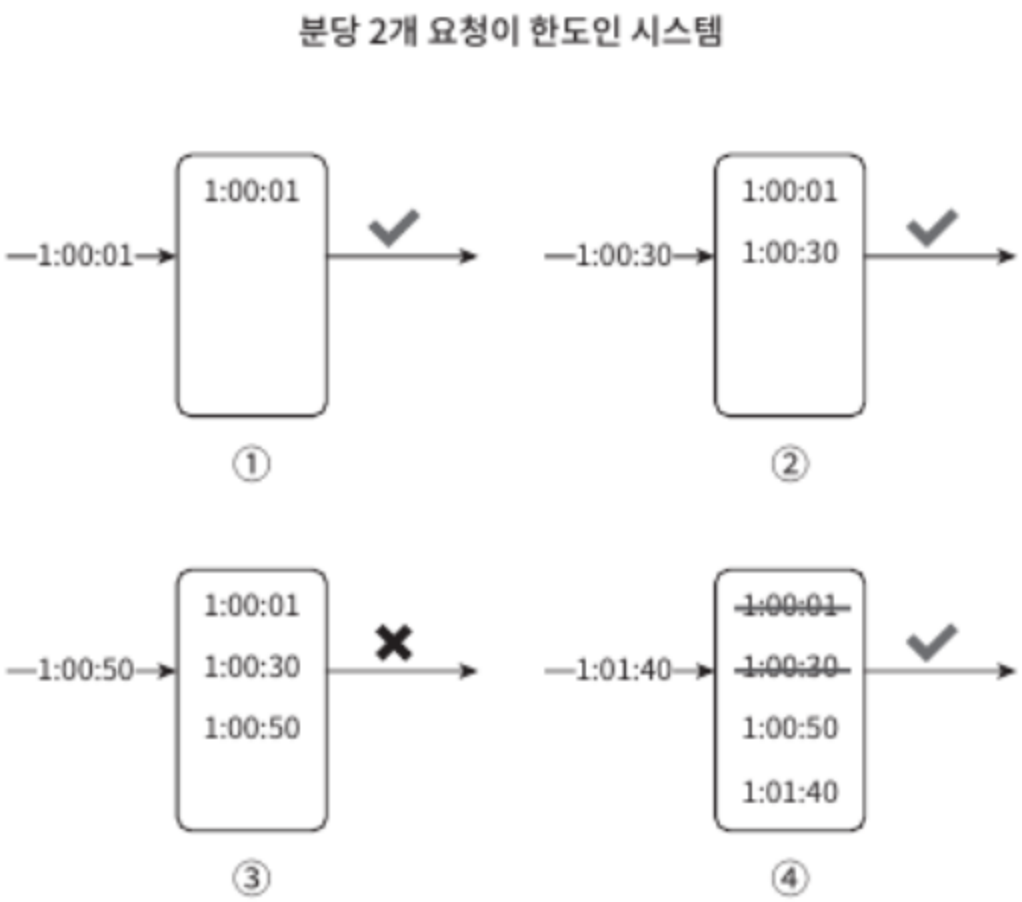


그림 4-10

3번 알고리즘의 큰 문제점을 해결할 수 있는 알고리즘

- 요청의 타임스탬프를 추적한다. (레디스의 정렬 집합같은 캐시에 보관)
- 새 요청 진입 시, 만료된 타임스탬프는 제거된다.
- 새 요청의 타임스탬프를 로그에 추가
- 로그의 크기가 허용치보다 같거나 작으면 요청을 시스템에 전달하고 그렇지 않은 경우 처리를 거부한다.

장점

- 어느 순간의 윈도를 보더라도, 허용되는 요청의 갯수는 시스템의 처리율 한도를 넘지 않음 (메커니즘이 아주 정교함)

단점

- 거부된 요청의 타임 스탬프 또한 보관하므로 다량의 메모리를 사용

5. **이동 윈도 카운터 알고리즘 (sliding window counter)**

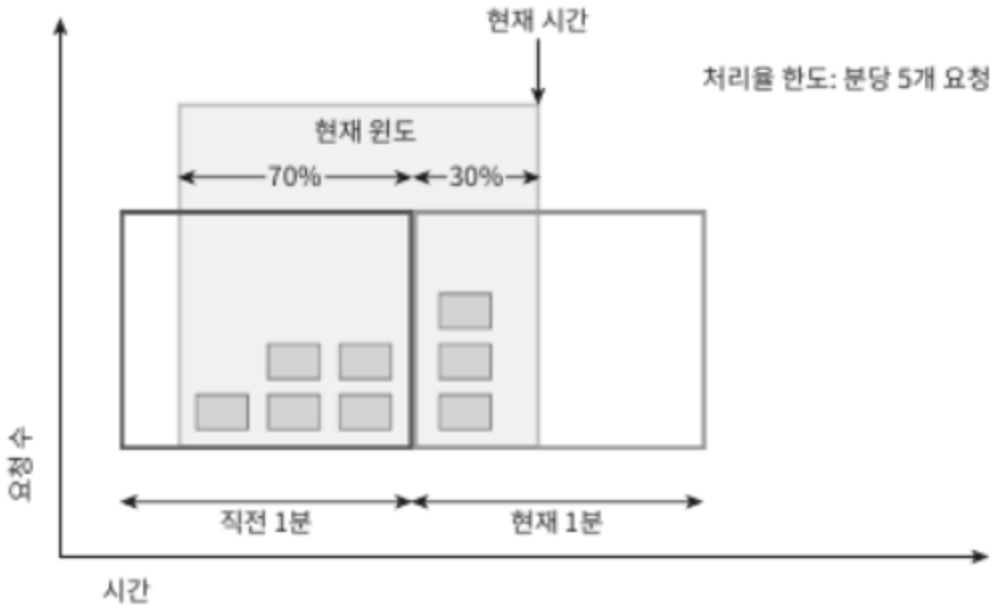


그림 4-11

→ 고정 윈도 카운터 알고리즘 + 이동 윈도 로깅 알고리즘

- 현재 1분간의 요청 + 직전 1분간의 요청 수 x 이동 윈도우와 직전 1분이 겹치는 비율을 이용해 헌재 윈도에 들어 있는 요청의 수를 계산할 수 있다.

장점

- 이전 시간대의 평균 처리율에 따라 현재 윈도의 상태를 계산하며 짧은 시간에 몰리는 트래픽에 잘 대응한다.
- 메모리 효율에 좋디

단점

- 직전 시간대에 도착한 요청이 균등하게 분포되어 있다고 가정한 상태에서 추정치를 계산하므로 다소 느슨하다. 하지만 이는 심각한 오류는 아니다.

개략적인 아키텍쳐

얼마나 많은 요청이 접수되었는지 추적할 수 있는 카운터를 추적 대상별로 둘 것 인가? IP 주소별로 둘 것인가? 등등에 대한 고민을 해야한다. 그렇다면 카운터는 어디에 보관해야 할까?

데이터베이스면 디스트에 접근해야하므로 속도가 느려진다. **매모리 상에서 동작하는 캐시가 바람직한데, 빠르고 시간에 기반한 만료 정책을 지원하기 때문이다.**

레디스는 처리율 제한 장치를 구현하는데 많이 사용되는 메모리 기반 저장장치로 INCR, EXPIRE 두가지 명령어를 지원

INCR : 메모리에 저장된 카운터의 값을 1만큼 증가시킨다.

EXPIRE : 카운터에 타임아웃 값을 설정한다. 설정된 시간이 지나면 카운터는 자동 삭제된다.

동작원리

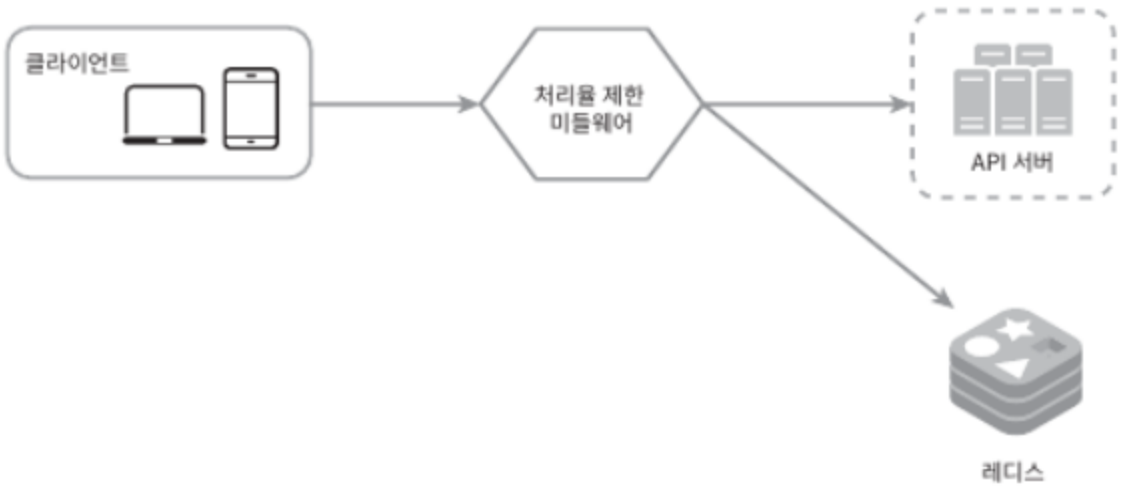


그림 4-12

- 클라이언트가 처리율 제한 미들웨어에게 요청을 보낸다
- 처리율 제한 미들웨어는 레디스의 지정 버킷에서 카운터를 가져와서 한도에 도달했는지 아닌지 검사한다.
    - 한도에 도달했다면 요청은 거부된다.
    - 한도에 도달하지 않았다면 API 서버로 전달된다. (미들웨어는 카운터의 값을 증가시킨 후 다시 레디스에 저장)

3단계) 상세 설계

- 처리율 제한 규칙은 어떻게 만들어지고 어디에 저장되는가?
- 처리가 제한된 요청들은 어떻게 처리되는가?

위 단계까지만 진행한다면 두 질문에 대한 답을 알 수 없다.

처리율 제한 규칙

- 마케팅 메세지의 최대치를 하루 5개로 제한하고 있다.

처리가 제한된 요청의 처리 전략

- 어떤 요청이 한도 제한이 걸리면 API는 **HTTP 429 응답 클라이언트에게** 보낸다.
- 경우에 따라 한도 제한에 걸린 메세지를 나중에 처리하기 위해 큐에 보관할 수 있다.

처리율 제한 장치가 사용하는 HTTP 헤더

- 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 하는 지 알림

사용자가 너무 많은 요청을 보내면 429 too many reqquests 오류를 헤더와 함께 반환하도록 한다.

상세설계

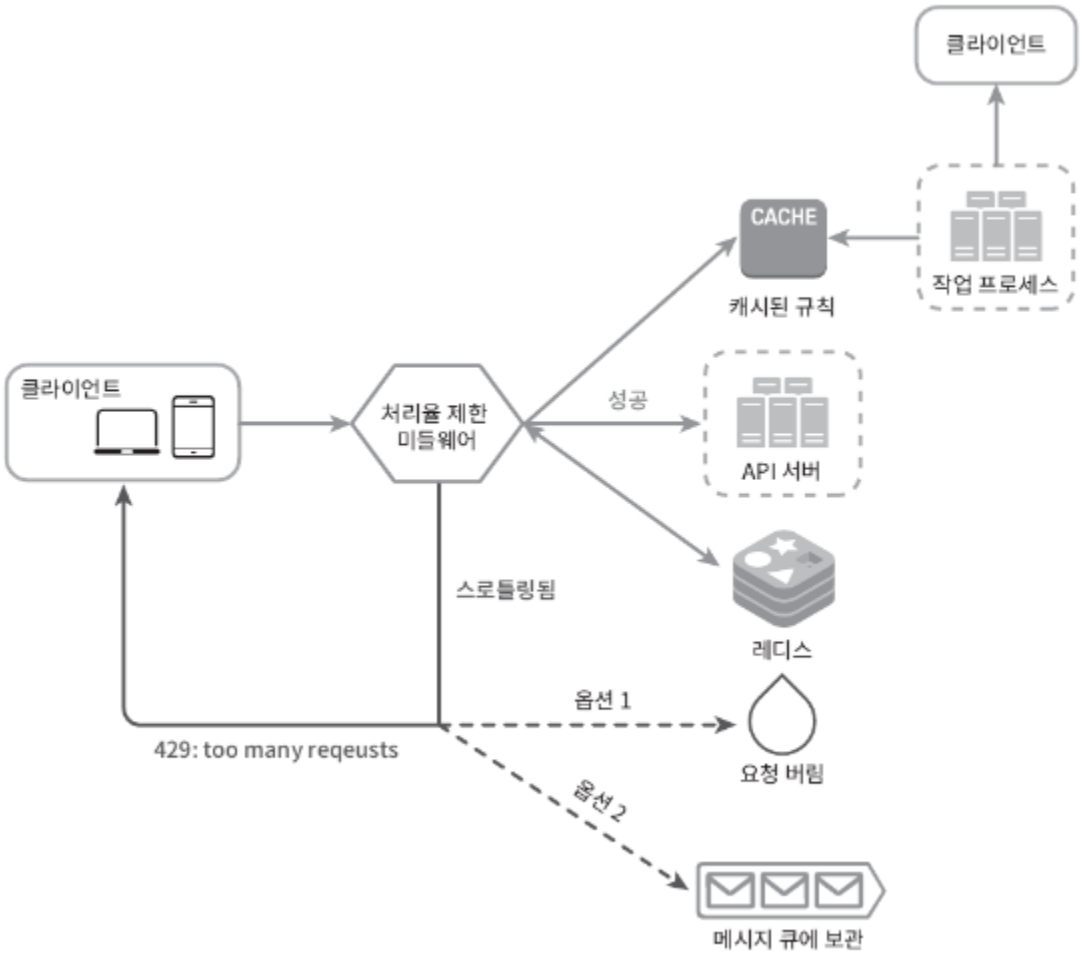


그림 4-13

- **처리율 제한 규칙은 디스크에 보관** → 작업 프로세스는 수시로 규칙을 디스크에서 읽어 캐시에 저장
- 클라이언트가 요청을 서버에 보내면 **요청은 먼저 처리율 제한 미들웨어에 도달**
- 처리율 제한 미들웨어는 제한 규칙을 캐시에서 가져온다. 그 후 레디스 캐시에서 카운터 및 마지막 요청의 타임스탬프를 가져온다. 가져온 값을 이용해 미들웨어는 두가지 방향으로 결정한다.
  - 해당 요청이 처리율 제한에 걸리지 않은 경우 → API 서버로 보낸다
  - 해당 요청이 처리율 제한에 걸린 경우 → 429 too many request 에러를 클라이언트에 보내며 상황에 따라 위 요청은 버려지거나 메세지 큐에 보관된다.

분산 환경에서의 처리율 제한 장치의 구현

단일 서버를 지원하는 처리율 제한 장치를 구현하는 것은 어렵지 않지만 여러 대의 서버와 병렬프로세스를 지원하도록 시스템을 확장하는 것은 **'경쟁 조건(race condition)과 동기화(synchronization)'**을 고려해야한다.

경쟁조건

처리율 제한 장치는

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

counter 값에 대해 두개의 요청이 한번에 접근할 경우 race condition이 발생한다.

해결책

- Lock.. 가장 유명 하지만 시스템의 성능을 상당히 떨어뜨린다.
- 루아 스크립트
- 레디스 자료구조 중 하나인 정렬 집합

동기화 이슈

수백만 사용자를 지원하려면 처리율 제한 장치 서버를 여러 대 두고 동기화해야 한다.

각 웹 계층은 stateless라서 클라이언트는 각기 다른 서버에 값을 보낼 수 있지만 서버간 동기화가 되지 않아 처리율 제한을 올바르게 수행할 수 없을 것

해결책

고정 세션 (sticky session) → 클라이언트로부터 요청은 항상 같은 처리율 제한 장치로 보냄

하지만 비추, 왜? → 확장, 유연성 X,

More Better! 레디스와 같은 중앙 **집중형 데이터 저장소 사용** (Q. 그럼 이 레디스에는 무엇을 저장해? counter 같은 걸 저장하나?)

성능 최적화

- 여러 데이터센터를 지원하는 문제는 처리율 제한 장치에 매우 중요함
- 제한 장치 간에 데이터 동기화를 할 때 최종 일관성 모델을 사용하는 것

모니터링

모니터링을 통해 아래 두가지를 확인해 볼 수 있다.

- 채택된 처리율 제한 알고리즘이 효과적이다.
- 정의한 처리율 제한 규칙이 효과적이다.

4단계) 마무리

이 외에도 아래 3가지를 더 고려해 볼 수도 있다.

- **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.
- **다양한 계층에서의 처리율 제한**

  이번장에서는 어플리케이션 계층에서만 다뤘지만 OS 7계층의 각 층에서도 처리율 제한이 가능하다.
- **처리율 제한을 회파하는 방법, 클라이언트를 어떻게 설계하는 것이 최선인가?**
  - 클라이언트 측 캐시를 사용하여 API 호출 횟수를 줄인다.
  - 처리율 제한의 임계치를 이해하고, 짧은 시간동안 너무 많은 메세지를 보내지 않도록 함
  - 예외나 에러를 처리하는 코드를 도입하여 보다 유연하게 예외적 상황을 극복할 수 있게 한다.
  - 재시도 로직을 구현할 때는 충분한 백오프 시간을 둔다.