

4장. 처리율 제한 장치의 설계

🕒 Created	@September 29, 2022 1:02 PM
📌 Progress	DONE



학습 TODO list

- ☐ DoS(Denial of Service) 공격
- ☐ 자원 고갈(resource starvation)
- ☐ API 게이트웨이
- ☐ API 엔드포인트

4.1. 1단계: 문제 이해 및 설계 범위 확정

4.2. 2단계: 개략적 설계안 제시 및 동의 구하기

4.2.1. 처리율 제한 장치의 위치

4.2.2. 처리율 제한 알고리즘

4.2.3. 개략적인 아키텍처

4.3. 3단계: 상세 설계

4.3.1. 처리율 제한 규칙

4.3.2. 처리율 한도 초과 트래픽 처리

4.3.3. 상세 설계

4.3.4. 분산 환경에서의 처리율 제한 장치의 구현

4.4. 4단계: 마무리

네트워크 시스템에서 처리율 제한 장치(rate limiter)는 클라이언트 또는 서비스가 보내는 트래픽의 처리율(rate)을 제어하기 위한 장치다.

- HTTP 통신 예: 처리율 제한 장치 가 특정 기간 내에 전송되는 클라이언트의 요청 횟수를 제한함
 - API 요청 횟수가 제한 처리율 제한 장치 에 정의된 임계치(threshold)를 넘어서면 추가로 도달한 모든 호출은 처리가 중단(block)됨
 - ex. 사용자는 초당 2회 이상 새 글을 올릴 수 없다.
 - ex. 같은 IP 주소로는 하루에 10개 이상의 계정을 생성할 수 없다.
 - ex. 같은 디바이스로는 주당 5회 이상 리워드(reward)를 요청할 수 없다.

API에 처리율 제한 장치를 두면 좋은 점

- DoS(Denial of Service) 공격 에 의한 자원 고갈(resource starvation) 을 방지할 수 있음
- 비용 절감
 - 서버를 많이 두지 않아도 되고, 우선순위가 높은 API에 더 많은 자원을 할당할 수 있음
 - 제3자(third-party) API에 사용료를 지불하고 있는 회사들의 경우 과금 기준에 따라 API 요청 횟수를 제한할 수 있음
 - 서버 과부하 를 막음: 봇(bot)에서 오는 트래픽이나 사용자의 잘못된 이용 패턴으로 유발된 트래픽을 걸러냄

설계 예시

1단계: 문제 이해 및 설계 범위 확정

2단계: 개략적 설계안 제시 및 동의 구하기

3단계: 상세 설계

4단계: 마무리

4.1. 1단계: 문제 이해 및 설계 범위 확정



어떤 종류의 처리율 제한 장치를 설계해야 하나요? 클라이언트 측 제한 장치입니까, 아니면 서버 측 제한 장치입니까?



좋은 질문이에요. 서버측 API를 위한 장치를 설계한다고 가정합니다.



어떤 기준을 사용해서 API 호출을 제어해야 할까요? IP 주소를 사용해야 하나요? 아니면 사용자 ID? 아니면 생각하는 다른 어떤 기준이 있습니까?



다양한 형태의 제어 규칙(throttling rules)을 정의할 수 있도록 하는, 유연한 시스템이어야 합니다.



시스템의 규모는 어느 정도여야 할까요? 스타트업 정도 회사를 위한 시스템입니까 아니면 사용자가 많은 큰 기업을 위한 제품입니까?



설계할 시스템은 대규모 요청을 처리할 수 있어야 합니다.



시스템이 분산 환경에서 동작해야 하나요?



그렇습니다.



이 처리율 제한 장치는 독립된 서비스입니까 아니면 애플리케이션 코드에 포함될 수도 있습니까?



그 결정은 본인이 내려주시면 되겠습니다.



사용자의 요청이 처리율 제한 장치에 의해 걸려진 경우 사용자에게 그 사실을 알려야 하나요?



그렇습니다.

• 시스템 요구사항 요약

- 설정된 처리율을 초과하는 요청은 정확하게 제한한다.
- 낮은 응답시간: 이 처리율 제한 장치는 HTTP 응답시간에 나쁜 영향을 주어서는 곤란하다.
- 가능한 한 적은 메모리를 써야 한다.
- 분산형 처리율 제한(distributed rate limiting): 하나의 처리율 제한 장치를 여러 서버나 프로세스에서 공유할 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 안된다.

4.2. 2단계: 개략적 설계안 제시 및 동의 구하기

기본적인 클라이언트-서버 통신 모델을 사용하여 설계해보자.

4.2.1. 처리율 제한 장치의 위치

- 클라이언트 측에 둘 경우: 클라이언트 요청은 위변조가 가능하고 모든 클라이언트의 구현을 통제하는 것은 어려움 → 일반적으로 클라이언트에서는 처리율 제한을 안정적으로 걸 수 없음
- 서버 측에 둘 수 있음
- 처리율 제한 미들웨어(middleware)를 만들어 해당 미들웨어가 API 서버로 가는 요청을 통제할 수 있음
 - ex. API 서버의 처리율이 초당 2개의 요청으로 제한된 상황에서, 클라이언트가 3번째 요청을 같은 초 범위 내에서 전송하였을 경우 → 앞 두 요청은 API 서버로 전송되고 세 번째 요청은 처리율 제한 미들웨어(middleware)에 의해 가로막히고 클라이언트로 HTTP 상태코드 429가 반환됨
 - ex. 클라우드 마이크로서비스의 경우 보통 API 게이트웨이(gateway) 컴포넌트에 처리율 제한 장치 가 구현됨

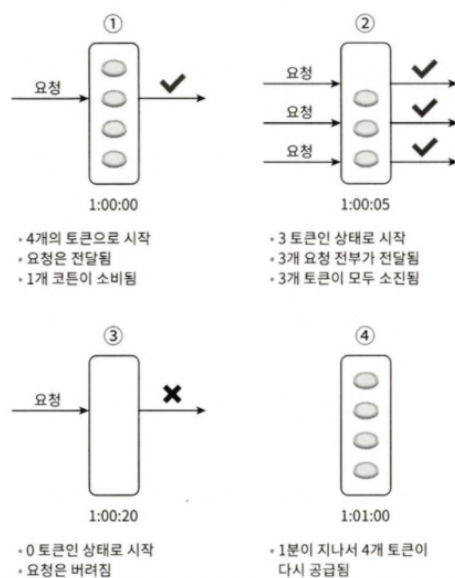


처리율 제한 장치의 위치 설계 시 고려 사항

- 프로그래밍 언어, 캐시 서비스 등 현재 사용하고 있는 기술 스택 점검
 - 현재 사용하는 프로그래밍 언어가 서버 측 구현을 지원하기 충분할 정도로 효율이 높은가?
- 사업 필요에 맞는 처리율 제한 알고리즘 찾기
 - 서버 측에서 모든 것을 구현하는가? → 알고리즘을 자유롭게 선택
 - 제3 사업자가 제공하는 게이트웨이를 사용하기로 했는가? → 선택지 제한
- 설계가 마이크로서비스에 기반하고 사용자 인증이나 IP 허용목록 관리 등을 처리하기 위한 API 게이트웨이가 있는가? → 처리율 제한 기능을 게이트웨이에 포함
- 처리율 제한 장치를 구현할 인력이 충분한가? → 상용 API 게이트웨이 고려

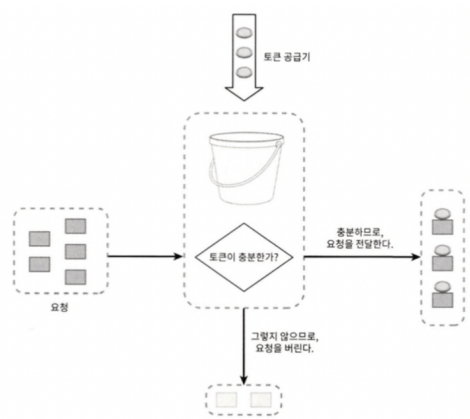
4.2.2. 처리율 제한 알고리즘

- 토큰 버킷(token bucket) 알고리즘: 아마존, 스트라이프가 API 요청을 통제(throttle)하기 위해 사용함
 - 동작 원리



처리 제한 로직 작동 예: 토큰 공급률(refill rate) = 분당 4

- 토큰 버킷: 지정된 용량을 갖는 컨테이너 → 사전 설정된 양의 토큰이 주기적으로 채워짐
 - 토큰이 꽉 찬 버킷에는 더 이상의 토큰이 추가되지 않음 → 버킷이 가득 차면 추가로 공급된 토큰은 버려짐(overflow)
- 각 요청은 처리될 때마다 하나의 토큰을 사용함 → 요청이 도착하면 버킷에 충분한 토큰이 있는지 검사함



- 충분한 토큰이 있는 경우: 버킷에서 토큰 하나를 꺼낸 후 요청을 시스템에 전달
- 충분한 토큰이 없는 경우: 해당 요청은 버려짐(dropped)
- 인자(parameter): 버킷 크기, 토큰 공급률(refill rate)

| 버킷의 개수를 정하는 기준: 공급 제한 규칙

- 통상적으로 API 엔드포인트(endpoint)마다 별도의 버킷을 둬
 - ex. 사용자마다 하루에 한 번만 포스팅 할 수 있고, 친구는 150명까지 추가할 수 있고, 좋아요 버튼을 다섯 번까지만 누를 수 있을 경우 → 3개의 버킷
- IP 주소별로 처리율 제한을 적용해야할 경우: IP 주소마다 버킷을 하나씩 할당
- 시스템 처리율을 초당 10,000개 요청으로 제한하고 싶을 경우: 모든 요청이 하나의 버킷을 공유



장점

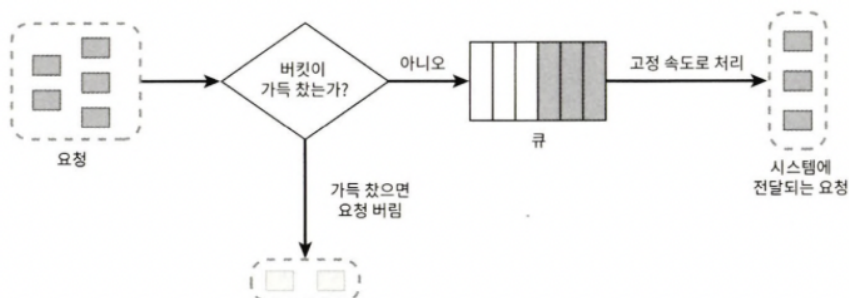
- 구현이 쉬움
- 메모리 사용 측면에서 효율적
- 짧은 시간에 집중되는 트래픽(burst of traffic)도 처리 가능 → 버킷에 남은 토큰이 있거나 하면 요청은 시스템에 전달될 것



단점

- 버킷 크기와 토큰 공급률, 두 인자를 적절하게 튜닝하는 것이 까다로움

- **누출 버킷(leaky bucket) 알고리즘**: 요청 처리율이 고정되어있음, FIFO(First-In-First-Out) 큐로 구현. Shopify가 누출 버킷 알고리즘을 사용하여 처리율 제한을 구현하고 있음
 - 동작 원리



- 요청이 도착하면 큐가 가득 차 있는지 확인 → 빈자리가 있는 경우 큐에 요청을 추가함
- 큐가 가득 차 있는 경우 새 요청을 버림
- 지정된 시간마다 큐에서 요청을 꺼내어 처리함
- 인자(parameter): 버킷 크기(=큐 사이즈), 처리율(outflow rate)



장점

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

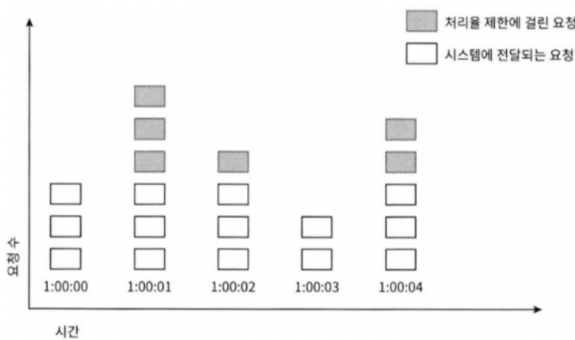


단점

- 단시간에 많은 트래픽이 몰리는 경우 큐에는 오래된 요청들이 쌓이게 되고, 그 요청들을 제때 처리 못 하면 최신 요청들은 버려짐
- 두 개 인자를 올바르게 튜닝하기 까다로울 수 있음

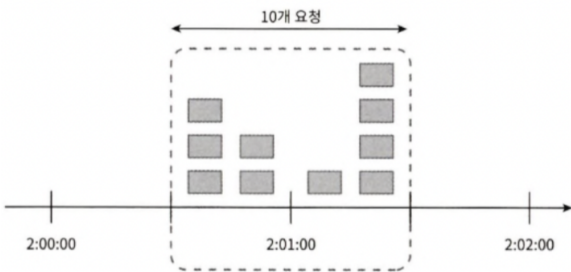
- 고정 윈도우 카운터(fixed window counter) 알고리즘

- 동작 원리



타임라인: 1초, 임계치: 초당 3개의 요청

- 타임라인(timeline)을 고정된 간격의 윈도우(window)로 나누고, 각 윈도우마다 카운터(counter)를 붙임
 - 요청이 접수될 때마다 이 카운터 값은 1씩 증가함
 - 카운터의 값이 사전에 설정된 임계치(threshold)에 도달하면 새로운 요청은 새 윈도우가 열릴 때까지 버려짐
- 가장 큰 문제점: 윈도우의 경계 부근에 순간적으로 많은 트래픽이 집중될 경우 윈도우에 할당된 양보다 더 많은 요청이 처리될 수 있음



타임라인: 1초, 임계치: 초당 5개의 요청



장점

- 메모리 효율이 좋음
- 이해하기 쉬움
- 윈도우가 닫히는 시점에 카운터를 초기화하는 방식은 특정한 트래픽 패턴을 처리하기에 적합함

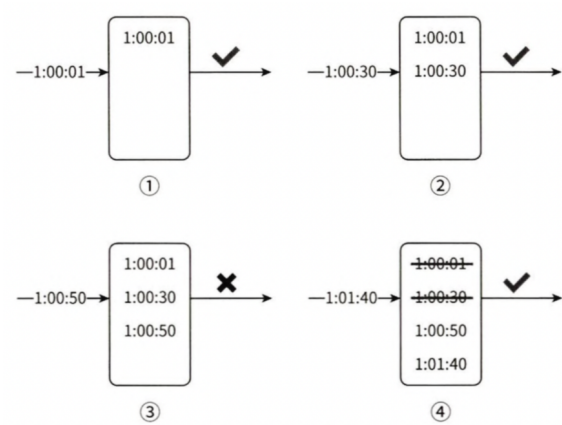


단점

- 윈도우 경계 부근에서 일시적으로 많은 트래픽이 몰려드는 경우, 기대했던 시스템의 처리 한도보다 많은 양의 요청을 처리하게 됨

- 이동 윈도우 로그(sliding window log) 알고리즘 : 고정 윈도우 카운터 알고리즘의 문제점 해결

◦ 동작 원리



임계치: 분당 2회의 요청 → 1:00:50 요청만 거부됨

- 요청의 타임스탬프(timestamp)를 추적함 → 타임스탬프 데이터는 보통 레디스(Redis)의 정렬 집합(sorted set) 같은 캐시에 보관함
- 새 요청이 오면 만료된 타임스탬프를 제거함
- 새 요청의 타임스탬프를 로그(log)에 추가함
- 로그의 크기가 허용치보다 같거나 작으면 요청을 시스템에 전달함 → 그렇지 않은 경우에는 처리를 거부



장점

- 어느 순간의 윈도우를 보더라도, 허용되는 요청의 개수는 시스템의 처리율 한도를 넘기지 않음

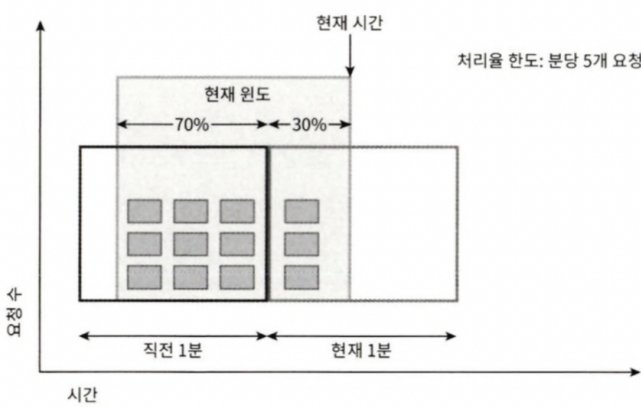


단점

- 다량의 메모리를 사용함(거부된 요청의 타임스탬프도 보관)

- 이동 윈도우 카운터(sliding window counter) 알고리즘 : 고정 윈도우 카운터 알고리즘 + 이동 윈도우 로그 알고리즘

◦ 동작 원리: 현재 윈도우의 요청 수 계산 방법



임계치: 분당 7회의 요청

- 현재 1분간의 요청 수 + 직전 1분간의 요청 수 × 이동 윈도우와 직전 1분이 겹치는 비율
 - ex. $3 + 5 \times 70\% = 6.5\text{개}$ → 현재 1분에서 30% 시점에 도착한 신규 요청은 시스템에 전달될 것



장점

- 이전 시간대의 평균 처리율에 따라 현재 윈도우의 상태를 계산 → 짧은 시간에 몰리는 트래픽에도 잘 대응함
- 메모리 효율이 좋음

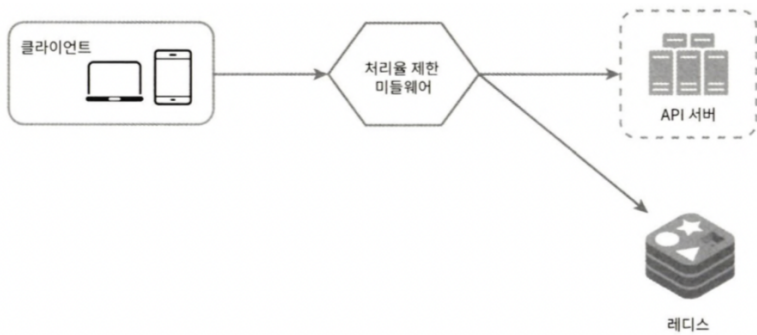


단점

- 직전 시간대에 도착한 요청이 균등하게 분포되어 있다고 가정한 상태에서 추정치를 계산하므로 다소 느슨함

4.2.3. 개략적인 아키텍처

- 처리율 제한 알고리즘**의 기본 아이디어: 얼마나 많은 요청이 접수되었는지를 추적할 수 있는 카운터를 추적 대상별로 두고(사용자별, IP 주소별, API 엔드포인트별, 서비스 단위별 등) 이 카운터의 값이 어떤 한도를 넘어서면 한도를 넘어 도착한 요청은 거부함
 - 카운터 보관 장소: 메모리상에서 동작하는 캐시
 - ex. 레디스(Redis)는 **처리율 제한 장치**를 구현할 때 자주 사용되는 메모리 기반 저장장치로, **INCR**과 **EXPIRE** 두 가지 명령어를 지원함



- 클라이언트가 **처리율 제한 미들웨어(rate limiting middleware)**에게 요청을 보냄
- 처리율 제한 미들웨어**: 레디스의 지정 버킷에서 카운터를 가져와서 한도에 도달했는지 아닌지를 검사 → 한도에 도달하면 요청 거부
- 한도에 도달하지 않으면 요청은 API 서버로 전달됨 → **처리율 제한 미들웨어**는 카운터의 값을 증가시킨 후 다시 레디스에 저장함

4.3. 3단계: 상세 설계



개략적 설계에서 상세 설계로 넘어가기 위한 고려 사항

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

4.3.1. 처리율 제한 규칙

- 리프트(Lyft)의 처리율 제한 오픈소스 컴포넌트 → 설정 파일 형태로 디스크에 저장

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

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

4.3.2. 처리율 한도 초과 트래픽 처리

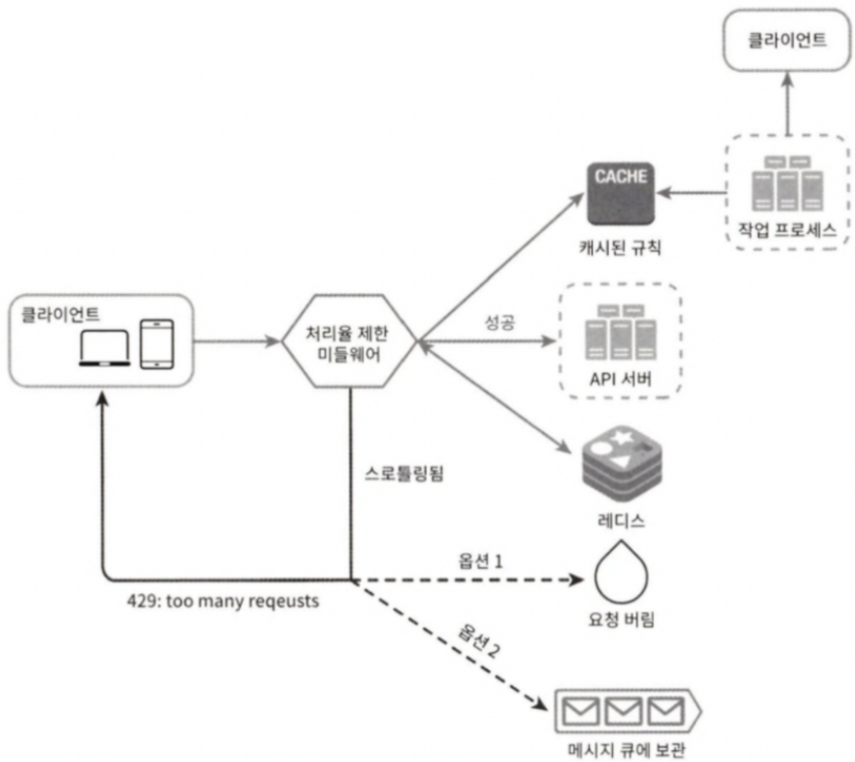
- 어떤 요청이 한도 제한에 걸리면 API는 **HTTP 429** 응답을 클라이언트에게 보냄

- 한도 제한에 걸린 메시지를 나중에 처리하기 위해 큐에 보관할 수도 있음

처리율 제한 장치가 사용하는 HTTP 헤더: 클라이언트가 요청이 처리율 제한에 걸리고 있는지 (throttle) 감지하는 방법

- 처리율 제한 장치 가 클라이언트에 보내는 HTTP 응답 헤더(response header)
 - X-Ratelimit-Remaining
 - X-Ratelimit-Limit
 - X-Ratelimit-Retry-After

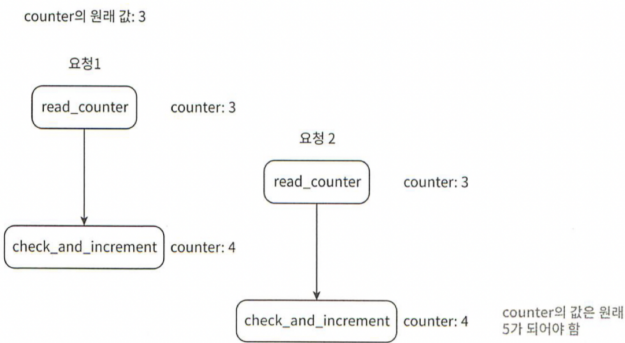
4.3.3. 상세 설계



- 처리율 제한 규칙은 디스크에 보관함. 작업 프로세스(workers)는 수시로 규칙을 디스크에서 읽어 캐시에 저장함
- 클라이언트 요청시 처리율 제한 미들웨어 에서 처리
- 처리율 제한 미들웨어 는 제한 규칙을 캐시에서 가져오고, 카운터 및 마지막 요청의 타임스탬프를 레디스 캐시에서 가져옴
 - 해당 요청이 처리율 제한에 걸리지 않은 경우에는 API 서버로 보냄
 - 해당 요청이 처리율 제한에 걸리면 429 too many requests 에러를 클라이언트에 보냄 → 요청을 버리거나 메시지 큐에 보관

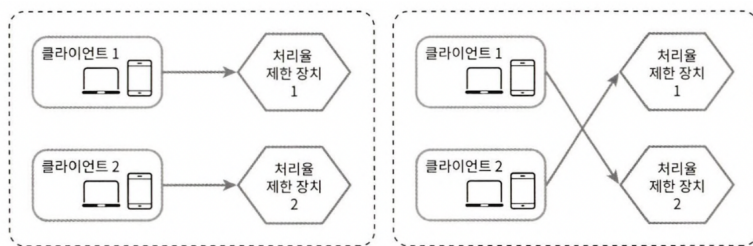
4.3.4. 분산 환경에서의 처리율 제한 장치의 구현

- 처리율 제한 장치 의 동작
 - 레디스에서 카운터의 값을 읽음(counter)
 - counter+1의 값이 임계치를 넘는지 봄
 - 넘지 않으면 레디스에 보관된 카운터 값을 1만큼 증가시킴
- 여러 대의 서버와 병렬 스레드를 지원하도록 시스템을 확장하는데 고려해야할 사항
 - 경쟁 조건(race condition) : 두 개의 요청을 처리하는 스레드(thread)가 각각 병렬로 counter 값을 읽었을 경우



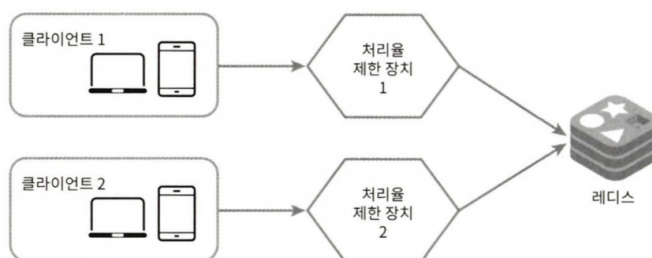
- 해결 방법

- 락(lock) → 시스템 성능을 떨어뜨린다는 문제가 있음
- 루아 스크립트(Lua script)
- 정렬 집합(sorted set)
- 동기화(synchronization) : 처리율 제한 장치 서버를 여러 대 두게 되면 동기화가 필요함



제한 장치 1은 클라이언트 2에 대한 정보가 없음

- 고정 세션(sticky session) : 같은 클라이언트로부터의 요청은 항상 같은 처리율 제한 장치로 보낼 수 있도록 함
→ 규모 확장 불가, 유연하지 않음
- 레디스와 같은 중앙 집중형 데이터 저장소 사용



성능 최적화(개선 가능한 부분들)

- 여러 데이터센터를 지원
 - 데이터센터에서 멀리 떨어진 사용자를 지원하다보면 지연시간(latency)가 증가함
 - 대부분의 클라우드 서비스는 세계 곳곳에 에지 서버(edge server)를 심어놓음 → 사용자의 트래픽을 가장 가까운 에지 서버로 전달하여 지연시간을 줄임
- 제한 장치 간에 데이터를 동기화할 때 최종 일관성 모델(eventual consistency model)을 사용



모니터링

- 채택된 처리율 제한 알고리즘이 효과적인가?
 - 이벤트 때문에 트래픽이 급증할 때: 처리율 제한 장치가 비효율적으로 동작하면 해당 트래픽 패턴을 잘 처리할 수 있도록 알고리즘을 바꾸는 것을 고려해야 함 → 토큰 버킷 알고리즘
- 정의한 처리율 제한 규칙이 효과적인가?
 - 처리율 제한 규칙이 너무 빡빡할 경우: 많은 유효 요청이 처리되지 못하고 버려질 것 → 규칙 완화 필요

4.4. 4단계: 마무리

- 경성(hard) 또는 연성(soft) 처리율 제한
 - 경성 처리율 제한 : 요청의 개수는 임계치를 절대 넘어서지 않음
 - 연성 처리율 제한 : 요청의 개수는 잠시 동안은 임계치를 넘어서지 않을 수 있음
- 다양한 계층에서의 처리율 제한
 - 이 책에서 다룬 부분은 애플리케이션 계층에서의 처리율 제한
 - IP 주소에서 처리율 제한 방법: Iptables 사용
- 처리율 제한을 회피하는 방법. 클라이언트를 어떻게 설계하는 것이 최선인가?

- 클라이언트 측 캐시를 사용하여 API 호출 횟수를 줄임
- 처리율 제한의 임계치를 이해하고, 짧은 시간 동안 너무 많은 메시지를 보내지 않도록 함
- 예외나 에러를 처리하는 코드를 도입하여 클라이언트가 예외적 상황으로부터 우아하게(gracefully) 복구될 수 있도록 함
- 재시도(retry) 로직을 구현할 때는 충분한 백오프(back-off) 시간을 둠