

4

4장. 처리율 제한 장치의 설계

처리율 제한 장치

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

ex)

- 사용자는 초당 2회 이상 새 글을 올릴 수 X
- 같은 IP 주소로는 하루에 10개 이상의 계정을 생성 불가능
- 같은 디바이스로는 주당 5회 이상 리워드(reward) 요청 불가능

API 처리율 제한 장치의 장점

- DoS(Denial of Service) 공격에 의한 자원 고갈 방지
- 추가 요청에 대한 처리를 제한 → 서버 많이 두지 않아도 됨 → 비용 절감
- 서버 과부하 방지

시스템 설계 4단계 접근법 적용

1단계 문제 이해 및 설계 범위 확정

요구사항

- 설정된 처리율을 초과하는 요청 제한

- 낮은 응답시간 : 해당 처리율 제한 장치는 HTTP 응답시간에 나쁜 영향을 주어서는 곤란하다.
- 적은 메모리 사용
- 분산형 처리율 제한 : 하나의 처리율 제한 장치를 여러 서버나 프로세스에 공유할 수 있어야 한다.
- 예외 처리 : 요청이 제한되었을 때는 그 사실을 사용자에게 분명하게 보여주어야 한다.
- 높은 결함 감내성 : 제한 장치에 장애가 생기더라도 전체 시스템에 영향을 주어서는 안 된다.

2단계 개략적 설계안 제시 및 동의 구하기

처리율 장치의 위치

- 클라이언트 측
 - 위변조가 쉬워 권장하지 않는다.
- 서버 측
 - API 서버에 두는 방법
 - 처리율 제한 미들웨어를 만들어 해당 미들웨어로 하여금 API 서버로 가는 요청 통제
 - 클라우드 마이크로 서비스의 경우, 처리율 제한 장치는 보통 API 게이트웨이라 불리는 컴포넌트에 구현됨

API 게이트웨이

- 처리율 제한, SSL 종단(termination), 사용자 인증(authentication), IP 허용 목록(white list) 관리 등을 지원하는 완전 위탁 관리형 서비스(fully managed)
- 즉 클라우드 업체가 유지 보수를 담당하는 서비스
- 일단은 처리율 제한을 지원하는 미들웨어라는 정도만 기억하기

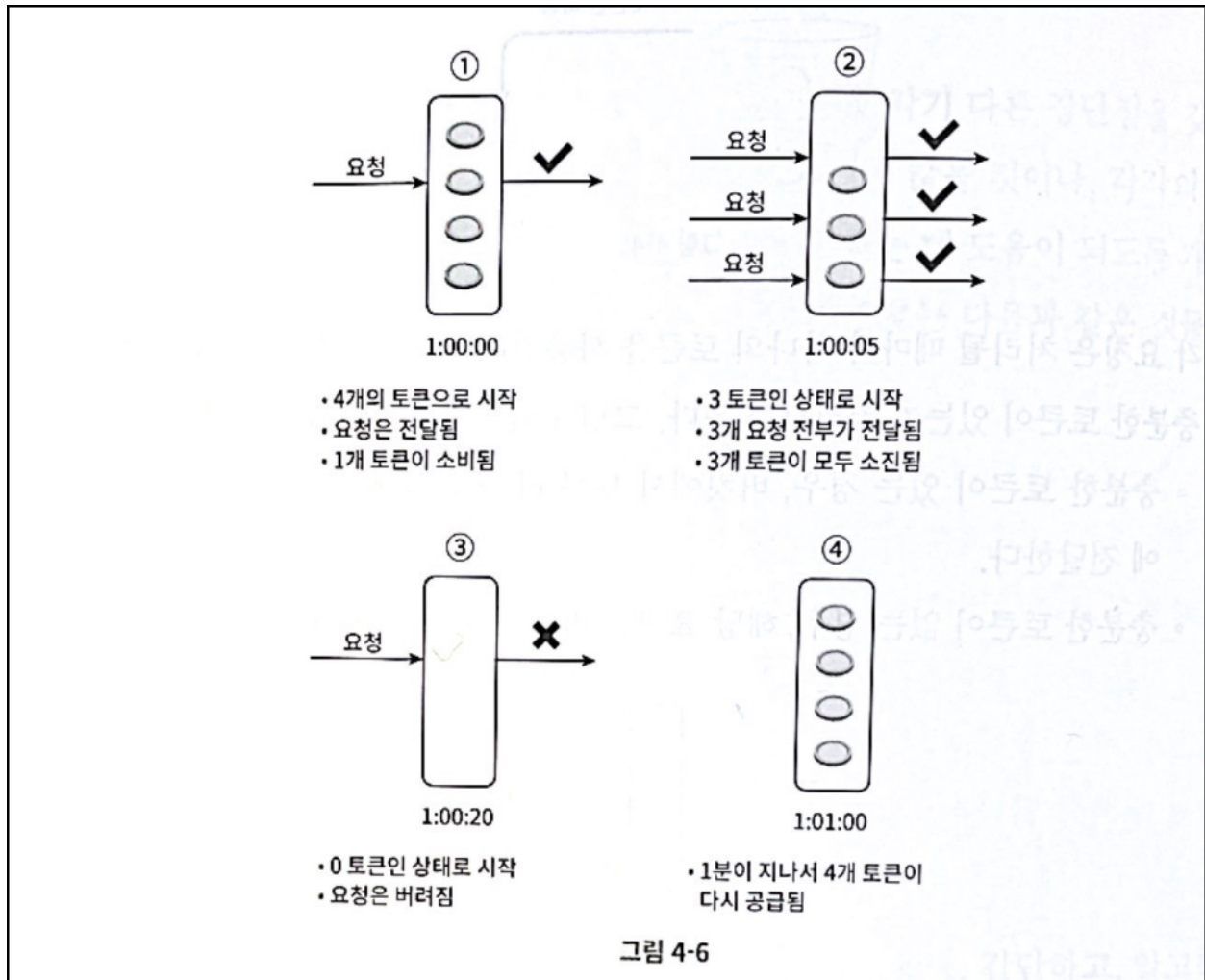
정리

- 서버에 두어야 하나? 게이트 웨이에 두어야 하나?



- 처리율 제한 장치를 어디에 둔다는 것에 대한 정답은 없음. 하지만 일반적인 지침은 있음.
- 현재 기술 스택 점검 → 현재 사용 중인 언어가 서버 측 구현을 지원하기 충분할 정도로 효율이 높은지 확인
- 상황에 맞는 알고리즘 사용 → 만약 제3 사업자가 제공하는 API Gateway 를 사용한다면 선택지는 제한이 될 수 있다.
- MSA 에 기반하고 있다면 인증, IP 허용 같은 기능을 이미 API Gateway 에 적용했을 수 있다. 그러면 처리율 제한도 API Gateway 에 포함시켜야 할 수도 있다.
- 충분한 인력이 없다면 상용 솔루션도 고려해보는 것이 좋다.



가능한 알고리즘



토큰 버킷 알고리즘



- 2개 인자 : 버킷 크기, 토큰 공급률을 인자로 받음
- 토큰이 주기적으로 채워진다.
- 각 요청이 처리될 때마다 하나의 토큰을 사용한다.
- 토큰이 없다면 해당 요청은 버려진다.



한국어 ▼

AWS > ... > 개발자 안내서



처리량 향상을 위해 API 요청 조절

[PDF](#) | [RSS](#)

API에 대한 제한 및 할당량을 구성하여 너무 많은 요청으로 인해 과부하되지 않도록 보호할 수 있습니다. 제한과 할당량은 모두 최선의 방식으로 적용되며 보장된 요청 한도가 아닌 대상으로 간주해야 합니다.

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.

요청 제출이 정상 상태의 요청 속도 및 버스트 제한을 초과할 경우 API Gateway에서 요청을 제한하기 시작합니다. 이 시점에서 클라이언트는 429 Too Many Requests 오류 응답을 받을 수 있습니다. 이러한 예외를 포착하면 클라이언트는 속도 제한 방식으로 실패한 요청을 다시 제출할 수 있습니다.

- 많은 기업들(아마존, 스트라이프 등)이 보편적으로 사용하는 알고리즘
- 통상적으로 API 엔드포인트마다 별도의 버킷을 둔다.
 - ex) 사용자마다 하루에 한 번만 포스팅 가능, 친구 150명 까지 추가 가능, 좋아요 버튼은 다섯 번까지만 누를 수 있다면, 사용자마다 3개의 버킷 필요
- IP 주소별로 처리율 제한을 적용해야 한다면 IP 주소마다 버킷을 하나씩 할당해야 한다.
- 시스템의 처리율을 초당 10,000 개 요청으로 제한한다면, 모든 요청이 하나의 버킷을 공유하도록 해야 한다.

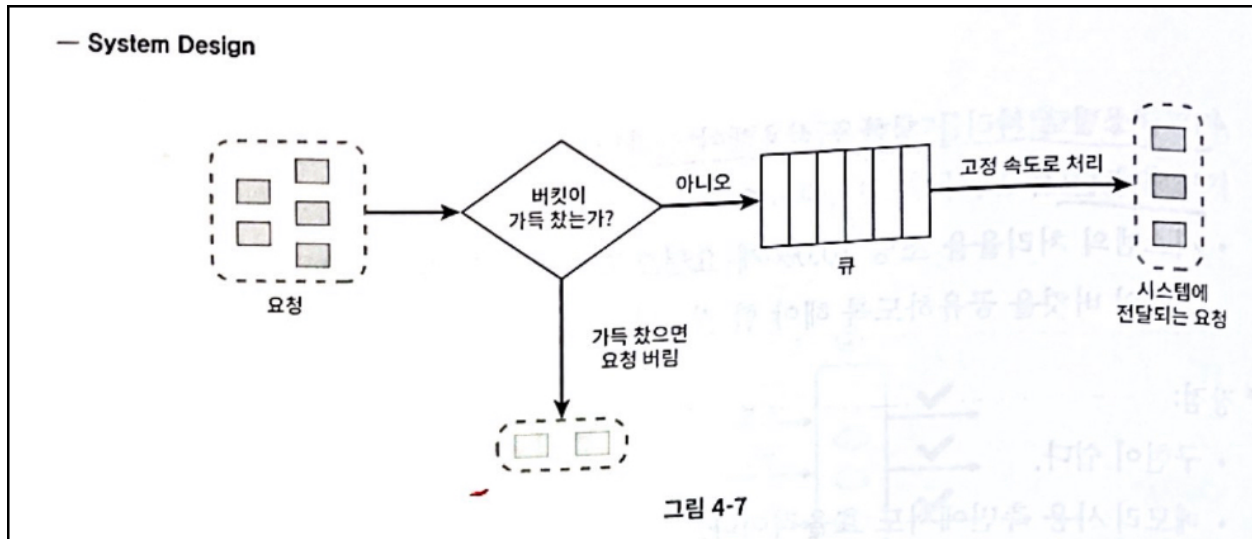
장점

- 구현이 쉬움
- 메모리 효율적
- 짧은 시간에 집중 되는 트래픽도 잘 처리

단점

- 버킷 크기 & 토큰 공급률 두 개의 인자를 필요로 하는 알고리즘이기 때문에 적절하게 튜닝하는 것이 어렵다.

누출 버킷 알고리즘



- 요청이 들어오면 큐가 가득 차 있는지 체크한다.
- 빈 자리가 있다면 큐에 요청을 추가한다.
- 만약 큐가 가득 차 있다면 요청은 버린다.
- 지정된 시간마다 큐에서 요청을 꺼내어 처리한다.
- 토큰 버킷 알고리즘과 비슷하지만, 요청 처리율이 고정되어 있다는 점이 다르다.
 - 큐에서 1초에 1개씩 꺼내!
- 보통 FIFO 큐로 구현한다.

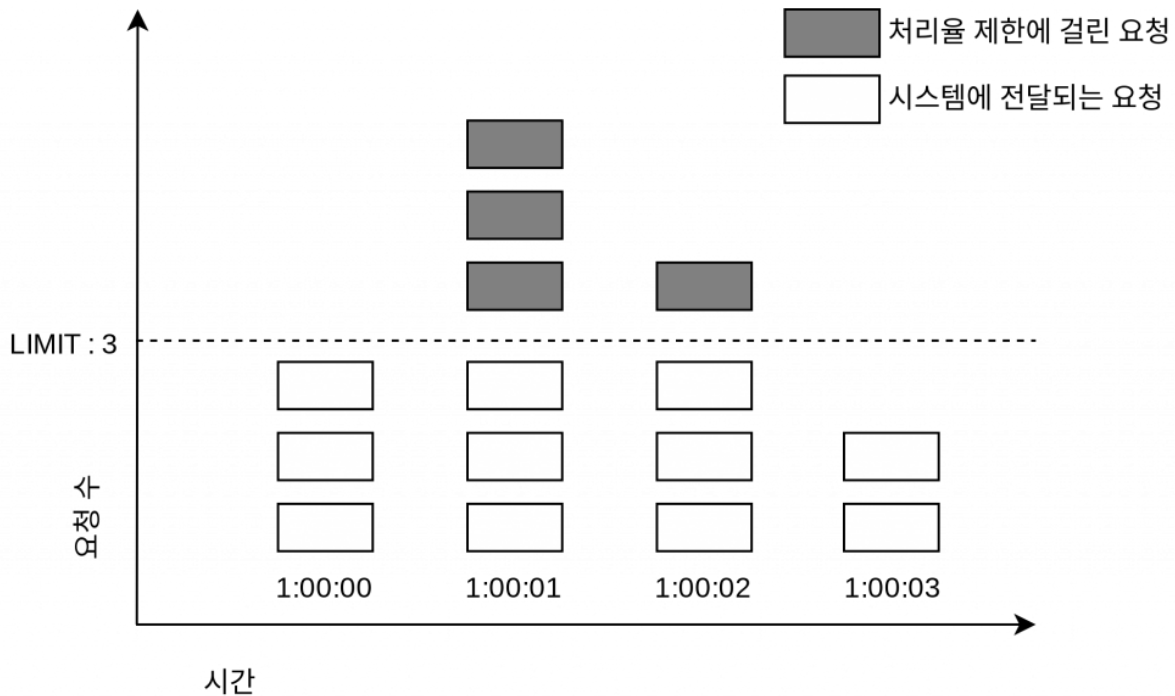
장점

- 큐의 크기 제한 -> 메모리 효율적 사용
- 고정된 처리율을 가지고 있어 안정적 출력 (stable outflow rate) 이 필요한 경우 적합

단점

- 단기간에 많은 트래픽이 몰리는 경우 최신 요청들이 버려지게 될 수 있음
- 토큰 버킷 알고리즘처럼 튜닝이 어렵다. (버킷 크기 & 처리율)

고정 윈도우 카운터 알고리즘



- 타임라인(timeline)을 고정된 간격의 윈도우(window)로 나누고, 각 윈도우마다 카운터(counter)를 붙인다.
- 요청 접수 -> 카운터 + 1
- 카운터가 임계치에 도달하면 새로운 요청은 새 윈도우가 열릴 때까지 버려진다.

장점

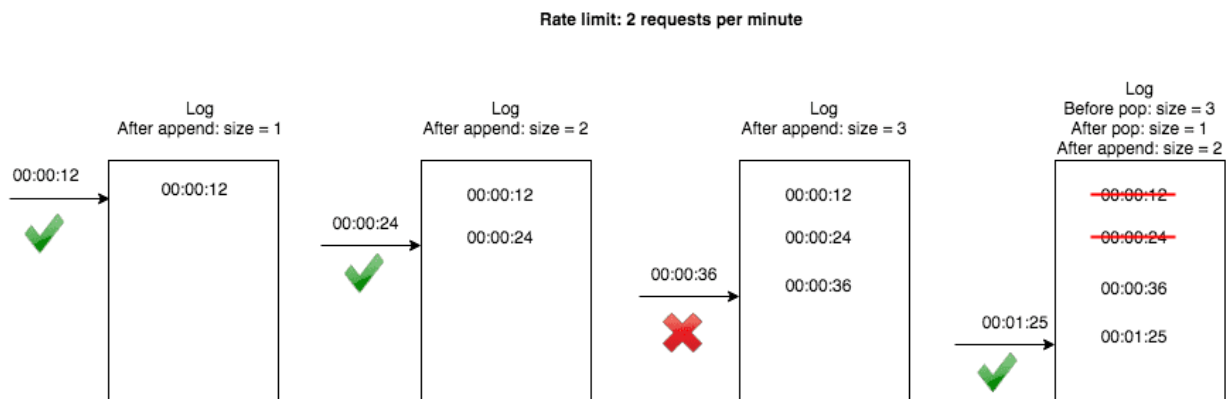
- 메모리 효율이 좋다.
- 이해하기 쉽다.
- 윈도우가 닫히는 시점에 카운터를 초기화하는 방식은 특정한 트래픽 패턴을 처리하기에 적합하다.

단점

- 윈도우 경계 부근에 트래픽이 몰리면 설정한 임계치를 초과할 수 있다.
 - ex) 분당 최대 5개 요청 → 2:00:00와 2:01:00 사이에 5개 요청, 2:01:00와 2:02:00 사이에 또 다섯 개의 요청이 들어왔다. 2:00:30 부터 2:01:30까지의 1분 동안 처리 요청 개수는 5개가 아닌 10개

이동 윈도우 로깅 알고리즘

고정 윈도우 카운터의 윈도우 경계 부근의 트래픽 집중 몰림 현상 해결



- 타임스탬프를 추적하는 알고리즘이다.
- 타임스탬프 데이터는 보통 레디스의 정렬 집합(sorted set) 같은 캐시에 보관
- 새 요청이 오면 만료된 타임스탬프는 제거 (만료된 타임스탬프 : 그 값이 현재 윈도우의 시작 지점보다 오래된 타임스탬프)
- 새 요청의 타임스탬프를 로그에 추가
- 로그의 크기가 허용치보다 같거나 작으면 시스템에 전달
- 허용치보다 크면 처리 거부

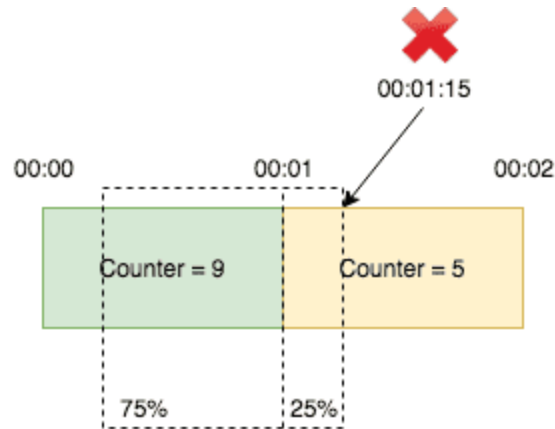
장점

- 처리율 제한 메커니즘이 매우 정교하다.
- 어느 순간의 윈도우를 보더라도 처리율 한도를 넘지 않는다.

단점

- 거부된 요청의 타임스탬프도 보관하기 때문에 메모리를 많이 사용한다.

이동 윈도우 카운터 알고리즘



- 고정 윈도우 카운터 + 이동 윈도우 로깅
- 현재 1분간의 요청 수 + 직전 1분간의 요청 수 \times 이동 윈도우와 직전 1분이 겹치는 비율

장점

- 이전 시간대의 평균 처리율에 따라 현재 윈도우의 상태를 계산하므로 짧은 시간에 몰리는 트래픽 대응에 용이하다.
- 메모리 효율이 좋다.

단점

- 직전 시간대에 도착한 요청이 균등하게 분포되어 있다고 가정
- 따라서 추정치를 계산하기 때문에 100% 정확하지는 않다.
- 하지만 클라우드플레어에서 수행한 실험에 의하면 오류 확률은 0.003%에 불과했다고 한다.

정리

- 얼마나 많은 요청이 접수되었는지를 추적할 수 있는 카운터를 추적 대상별로 두고(사용자별? IP 주소별? API 엔드포인트?)
- 카운터를 어디에 보관할 것 인가?

- db는 디스크 접근 때문에 느리니까 부적합
- 메모리가 빠르데다가 시간에 기반한 만료 정책 지원하기 때문에 적합

ex) Redis - INCR, EXPIRE 명령어

- INCR : 메모리에 저장된 카운터의 값을 1만큼 증가
- EXPIRE: 카운터의 타임아웃 값 → 일정 시간 지나면 카운터 삭제

3단계 상세 설계

처리율 제한 규칙

```
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(configuration file) 형태로 디스크에 저장

처리율 한도 초과 트래픽의 처리

- 요청 한도 제한 걸릴 시, HTTP 429 응답(too many requests)을 클라이언트에게 전송
- 메시지를 나중에 처리하기 위해 큐에 보관할 수도 있다.
- 클라이언트가 본인 요청이 처리율 제한에 걸리고 있는 감지하는 방법: HTTP 응답 헤더
 - X-Ratelimit-Remaining: 윈도우 내에 남은 처리 가능 요청의 수
 - X-Ratelimit-Limit: 매 윈도우마다 클라이언트가 전송할 수 있는 요청의 수
 - X-Ratelimit-Retry-After: 한도 제한에 걸리지 않으려면 몇 초 뒤에 요청을 다시 보내야 하는지 알림

분산 환경에서의 처리율 제한 장치의 구현

단일 서버가 아닌, 여러 대의 서버와 병렬 스레드를 지원하도록 시스템 확장하는 것은 또 다른 문제다. 다음 두 문제를 풀어보자.

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

이슈

경쟁 이슈가 발생하는 동작

해결 방법

레디스에서 카운터의 값을 읽는다.

counter +1 의 값이 임계치를 넘는지 본다.

넘지 않는다면 레디스에 보관된 카운터 값을 1만큼 증가시킨다.

- 가장 널리 알려진 해결책은 락 → 하지만 락은 시스템 성능을 상당히 떨어뜨림
- 락 대신 사용 가능한 솔루션 2가지 → 루아 스크립트(Lua script), 정렬 집합(sorted set)

A better approach – sorted sets

Fortunately, Redis has another data structure that we can use to prevent these race conditions – the **sorted set**. Here's the algorithm we came up with:

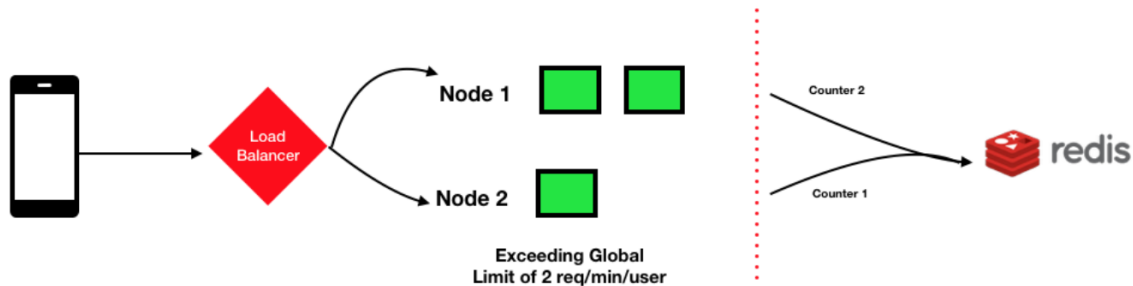
- Each user has a sorted set associated with them. The keys and values are identical, and equal to the (microsecond) times when actions were attempted.
- When a user attempts to perform an action, we first drop all elements of the set which occurred before one interval ago. This can be accomplished with Redis's `ZREMRANGEBYSCORE` command.
- We fetch all elements of the set, using `ZRANGE(0, -1)`.
- We add the current timestamp to the set, using `ZADD`.
- We set a TTL equal to the rate-limiting interval on the set (to save space).
- After all operations are completed, we count the number of fetched elements. If it exceeds the limit, we don't allow the action.
- We also can compare the largest fetched element to the current timestamp. If they're too close, we also don't allow the action.

<https://engineering.classdojo.com/blog/2015/02/06/rolling-rate-limiter/>

- 각 유저는 자신과 연결된 정렬된 집합 가지고 있음
 - 사용자가 작업 수행하려 하면, 한 interval 전에 발생한 집합 요소 모두 삭제
 - 그리고 집합의 모든 요소를 가져와서 현재 타임스탬프를 세트에 추가
- 모든 작업을 Redis를 이용해 원자적 작업으로 수행 가능하다는 장점 존재

Race Condition

A problem which occurs with shared datastore is **"Race Condition"** in highly concurrent environment which is result of **get then set** approach. We will get value form datastore ,increment it and set it. But Problem occurs when in meanwhile some other request come and increment and set a wrong value.



A better approach is to use a **"set-then-get"** mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.

- get then set → set then get 방법 사용

이슈

여러대의 처리율 제한 장치를 사용할 경우 동기화가 필요해진다

웹 계층은 무상태이므로 각 클라이언트는 각기 다른 제한 장치로 요청을 보낼 수 있음

해결 방법

- Sticky session 을 사용하여 같은 클라이언트로부터의 요청은 항상 같은 처리율 제한 장치로 보낼 수 있도록 할 수 있다.
 - 규모면에서 확장 가능하지도 않고 유연하지도 않아서 비추천
- 레디스와 같은 중앙 집중형 데이터 저장소 필요

이슈

성능 최적화

해결 방법

- 지연시간을 줄이기 위해 사용자의 트래픽을 가장 가까운 Edge 서버로 전달하여 지연시간을 줄인다.
- 제한 장치 간 동기화할 때 최종 일관성 모델을 사용해야한다.

4단계 마무리

추가적으로 알아두면 좋은 정보

경성 또는 연성 처리율 제한

- 경성 처리율 제한 : 요청의 개수는 임계치를 절대 넘어설 수 없다.
- 연성 처리율 제한 : 요청 개수는 잠시 동안은 임계치 넘기기 가능

다양한 계층에서의 처리율 제한

- 애플리케이션 계층(7번 계층)에서의 처리율 제한 외에도 다른 계층에서 제어
 - ex) Iptables를 사용하여 IP 주소(3번 계층)에 처리율 제한을 적용

처리율 제한을 회피하는 방법

- 클라이언트 측 캐시를 사용하여 API 호출 횟수 줄이기
- 예외나 에러를 처리하는 코드를 도입하여, 클라이언트가 예외적 상황으로부터 우아하게 복구될 수 있도록 한다.
 - 서버가 죽어서 안 되는건지, 요청이 많아서 안 되는 건지 클라에서 알 수 있도록 하자.
 - 에러를 받았을 때, 재처리 / 재시도 할 수 있도록 한다. 우아하게 서킷 브레이커 사용하자.
- 재시도 로직을 구현할 때는 충분한 백오프 시간을 두도록 한다.

참고링크

<https://velog.io/@haron/가상-면접-사례로-배우는-대규모-시스템-4장>

<https://newwisdom.tistory.com/117>

<https://velog.io/@dev-log/처리율-제한-장치-설계-툭아보기>

<https://jonghoonpark.com/2023/05/17/처리율-제한-장치-설계>