4️⃣

230924 4. 처리율 제한 장치의 설계

4. 처리율 제한 장치의 설계

처리율 제한 장치

클라이언트 또는 서비스가 보내는 트래픽의 처리율을 제어하기 위한 장치

ex) 특정 기간 내에 전송되는 클라이언트의 요청 횟수 제한

- 사용자는 초당 2회 이상 새 글을 올릴 수 없음
- 같은 IP 주소로는 하루에 10개 이상의 계정을 생성할 수 없음
- 같은 디바이스로는 주당 5회 이상 리워드를 요청할 수 없음

API에 처리율 제한 장치를 두면 좋은 점

1. DoS 공격에 의한 자원 고갈 방지
2. 비용 절감
3. 서버 과부화 막음

▼ 1. 문제 이해 및 설계 범위 확정

면접관과 소통하며 어떤 제한 장치를 구현해야 하는지 파악하기

처리율 제한 장치를 구현할 수 있는 여러 가지 알고리즘 중에 올바른 알고리즘 선택하기

▼ 2. 개략적 설계안 제시 및 동의 구하기

처리율 제한 장치는 어디에 둘 것인가?

- 클라이언트 측에 두기
    - 클라이언트 요청은 쉽게 위변조가 가능하여 처리율 제한을 안정적으로 걸 수 있는 장소가 못 됨
- 서버 측에 두기

1. API 서버에 처리율 제한 장치 두기
2. 처리율 제한 미들웨어를 통해 API 서버로 가는 요청 자체를 통제하기

⇒ 정답은 없기 때문에 기술 스택, 기존 설계, 인력 등 **조건과 상황에 맞는 처리율 제한 장치를** 찾기

처리율 제한 미들웨어

보통 API 게이트웨이 컴포넌트에 구현

- 처리율 제한, SSL 종단, 사용자 인등, IP 허용 목록 관리 등을 지원하는 완전 위탁관리형 서비스
- 클라우드 업체가 유지 보수를 담당하는 서비스

처리율 제한 알고리즘
▼ 토큰 버킷 알고리즘

동작 원리

1. 토큰 버킷은 지정된 용량을 갖는 컨테이너이며 사전 설정된 양의 토큰이 주기적으로 채워짐
2. 토큰이 꽉 찬 버킷에는 더 이상의 토큰이 추가되지 않고, 버킷이 가득 차면 추가로 공급된 토큰은 버려짐
3. 각 요청은 처리될 때마다 하나의 토큰을 사용함
4. 요청이 도착하면 버킷에 충분한 토큰이 있는지 검사 후 토큰이 있는 경우, 버킷에서 토큰 하나를 꺼내 요청을 시스템에 전달하고 없으면 해당 요청은 버려짐

인자 값

- 버킷 크기
  - 버킷에 담을 수 있는 토큰의 최대 개수
- 토큰 공급률
  - 초당 몇 개의 토큰이 버킷에 공급되는가

버킷은 몇 개나 사용해야 하는지?

- 공급 제한 규칙에 따라 달라짐
- 통상적으로, API 엔드포인트마다 별도의 버킷을 두고 잇음
- IP 주소별로 처리율 제한을 적용해야 한다면 IP 주소마다 버킷을 하나씩 할당해야 함
- 시스템의 처리율을 초당 10000개 요청으로 제한하고 싶다면, 모든 요청이 하나의 버킷을 공유하도록 해야 함

장점

- 구현이 쉬움
- 메모리 사용 측면에서 효율적임
- 짧은 시간에 집중되는 트래픽 처리 가능

단점

- 두 개의 인자를 적절하게 튜닝하는 것이 어려움

▼ 누출 버킷 알고리즘

토큰 버킷 알고리즘과 비슷하지만 요청 처리율이 고정되어 있다는 점이 다름

동작 원리

1. 요청이 도착하면 큐가 가득 차 있는지 보고 빈자리가 있는 경우 큐에 요청을 추가
2. 큐가 가득 차 있는 경우 새 요청을 버림
3. 지정된 시간마다 큐에 요청을 꺼내어 처리

인자 값

- 버킷 크기
  - 큐 사이즈와 같은 값
- 처리율
  - 지정된 시간당 몇 개의 항목을 처리할지 지정하는 값
  - 보통 초 단위로 표현

장점

- 큐의 크기가 제한되어 있어 메모리 사용량 측면에서 효율적
- 고정된 처리율을 갖고 있기 때문에 안정적 출력이 필요한 경우 적합

단점

- 단시간에 많은 트래픽이 몰리는 경우 큐에 오랜된 요청들이 쌓이면서 최신 요청들이 버려지게 됨
- 두 개의 인자를 적절하게 튜닝하는 것이 어려움

▼ 고정 윈도 알고리즘

동작 원리

1. 타임라인을 고정된 간격의 윈도로 나눔
2. 각 윈도마다 카운터를 붙임
3. 요청이 접수될 때마다 카운터의 값 1씩 증가
4. 카운터의 값이 사전에 설정된 임계치에 도달하면 새로운 요청은 새 윈도가 열릴 때까지 버려짐

장점

- 메모리 효율이 좋음
- 이해하기 쉬움
- 윈도가 닫히는 시점에 카운터를 초기화하는 방식은 특정한 트래픽 패턴을 처리하기 적합

단점

- 일시적으로 많은 트래픽이 몰릴 경우 기대했던 시스템의 처리 한도보다 많은 양의 요청을 처리하게 됨

▼ 이동 윈도 로깅 알고리즘

고정 윈도 카운터 알고리즘의 문제를 해결

동작 원리

1. 캐시에 보관된 타임스탬프 데이터를 추적함
2. 새 요청이 오면 만료된 타임스탬프 제거
3. 새 요청의 타임스탬프를 로그에 추가
4. 로그의 크기가 허용치보다 같거나 작으면 요청을 시스템에 전달하고 그렇지 않으면 처리를 거부

장점

- 정규한 메커니즘
- 어느 순간의 윈도를 보더라도 허용되는 요청의 개수는 시스템의 처리율 한도를 넘지 않음

단점

- 거부된 요청의 타임스탬프도 보관하기 때문에 다량의 메모리 사용

▼ 이동 윈도 카운터 알고리즘

고정 윈도 카운터 알고리즘 + 이동 윈도 로깅 알고리즘

현재 윈도에 몇 개의 요청이 온 것으로 보고 처리해야 하는지?

현재 1분간의 요청 수 + (직전 1분간의 요청 수 * 이동 윈도와 직전 1분이 겹치는 비율)

올림에서 사용할 수도, 내림하여 사용할 수도 있음

```
한도: 분당 7개
이전 1분 요청: 5개
현재 1분 요청: 3개

현재 1분의 30% 시점에 도착한 새 요청
3 + (5 * 70%) = 6.5
```

장점

- 이전 시간대의 평균 처리율에 따라 현재 윈도의 상태를 계산하므로 짧은 시간에 몰리는 트래픽에도 잘 대응함

- 메모리 효율이 좋음

단점

- 직전 시간대에 도착한 요청이 균등하게 분포되어 있다고 가정한 상태에서 추정치를 계산하기 때문에 다소 느슨

처리율 제한 알고리즘의 기본 아이디어

얼마나 많은 요청이 접수되었는지를 추적할 수 있는 카운터를 추적 대상별로 두고, 이 카운터의 값이 어떤 한도를 넘어 도착한 요청은 거부하는 것

카운터를 어디에 보관할 것인가?

- 데이터베이스
  - 디스크 접근 때문에 느림
- 캐시
  - 메모리에서 동작하기 때문에 빠름
  - 시간에 기반한 만료 정책 지원

Redis 캐시를 통해 구현한 처리율 제한 장치 동작법

1. 클라이언트가 처리율 제한 미들웨어에게 요청
2. 처리율 제한 미들웨어는 레디스의 지정 버킷에서 카운터를 가져와 한도에 도달했는지 아닌지 검사

▼ 3. 상세 설계

개략적 설계를 통해 알 수 없는 사항

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

처리율 제한 규칙(Lyft)

설정 파일 형태로 디스크에 저장

```
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

처리율 한도 초과 트래픽의 처리

- 어떤 요청이 와도 한도 제한에 걸리면 429 응답
- 한도 제한에 걸린 메세지를 나중에 처리하기 위에 큐에 보관할 수도 있음

처리율 제한 장치가 사용하는 HTTP 헤더

클라이언트가 자기 요청이 처리율 제한에 걸리고 있는지 감지할 수 있도록 HTTP 응답 헤더를 클라이언트에게 전송

- 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

상세 설계

1. 처리율 제한 규칙은 디스크에 보관(작업 프로세스는 수시로 규칙을 디스크에서 읽어 캐시에 저장)
2. 클라이언트가 요청을 서버에 보내면 요청은 먼저 처리율 제한 미들웨어에 도달
3. 제한 규칙 캐시에서 가져오기
4. 가져온 값들에 근거하여 미들웨어가 서버 호출 or 클라이언트에 응답

▼ 분산 환경에서의 처리율 제한 장치의 구현

여러 대의 서버와 병렬 스레드로 지원하도록 확장할 경우 해결해야 할 문제

1. 경쟁 조건
2. 동기화

▼ 경쟁 조건

경쟁 조건 문제를 해결하는 가장 널리 알려진 해결책은 **Lock**

하지만 시스템의 성능을 상당히 떨어뜨린다는 문제가 있음

다른 해결책 두 가지

1. 루아 스크립트
2. 정렬 집합

▼ 동기화 이슈

수백만 사용자를 지원하려면 한 대의 처리율 제한 장치 서버로는 부족

처리율 제한 장치 서버를 여러 대 두게 되면 동기화가 필요해짐

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

고정 세션을 활용하여 같은 클라이언트로부터 항상 같은 처리율 제한 장치로 보내게 하기

규모면에서 확장이 가능하지도 않고 유여하지도 않기 때문에 비추

중앙 집중형 데이터 저장소 사용

ex) Redis

▼ 성능 최적화

시스템 설계 면접의 단골 주제

여러 데이터센터를 지원하여 지연시간 증가 문제 해결

처리율 제한 장치 간에 데이터를 동기화할 때 최종 일관성 모델 사용

▼ 모니터링

모니터링을 통해 확인하려는 것

- 채택된 처리율 제한 알고리즘이 효과적인지
- 정의한 처리율 제한 규칙이 효과적인지
    - 제한 규칙이 너무 빡빡하면 많은 유효 요청이 처리되지 못함 → 규칙 완화

▼ 4. 마무리

시간이 허락할 경우 언급해보면 도움이 될 이슈

- 경성 또는 연성 처리율 제한
    - 경성 처리율 제한: 요청의 개수는 임계치를 절대 넘어설 수 없다
    - 연성 처리율 제한: 요청 개수는 잠시 동안은 임계치를 넘어설 수 없다
- 다양한 계층에서의 처리율 제한
- 처리율 제한을 회피하는 방법
    - 클라이언트를 어떻게 설계하는 것이 최선인가?

▼ 토론

처리율 제한 장치를 클라이언트 쪽에 두지 않는다고 했는데 서버 쪽과 동시에 이중으로 두는 건 어떻게 생각하는지? 오버 스펙이라고 생각하는지?

한 쪽에서 문제가 발생하더라도 다른 쪽에서 계속 관리 가능

클라이언트 측과 서버 측의 처리율 제한 장치 간의 독립성을 확보함으로써 보안과 신뢰성을 강화

두 개의 처리율 제한 장치를 사용하면 로드 밸런싱을 구현하여 서버의 부하를 균등하게 분산

관리 복잡

비용

두 장치 간의 동기화와 일관성을 유지 어렵

보안 강화가 필요할 때

다중 플랫폼을 지원할 때