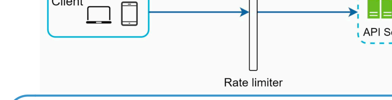
Alex Xu - System Design Interview (2nd ed.)

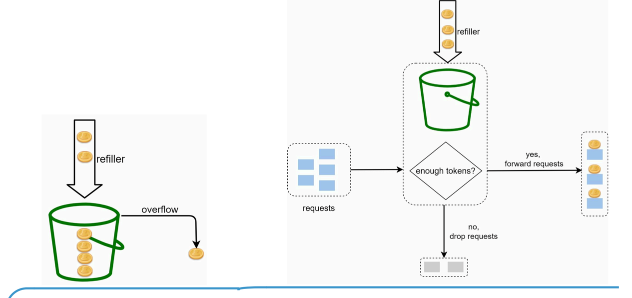
4. Design a Rate Limiter

- Benefits of using an API rate limiter
  - Prevent resource starvation caused by Denial of Service (DoS) attack
  - Reduce cost
  - Prevent servers from being overloaded
- Step 1. Understand the problem and establish design scope
  - Requirements
    - Accurately limit excessive requests
    - Low latency. The rate limiter should not slow down HTTP response time
    - Use as little memory as possible
    - Distributed rate limiting. The rate limiter can be shared across multiple servers or processes
    - Exception handling. Show clear exceptions to users when their requests are throttled
    - High fault tolerance. If there are any problems with the rate limiter (for example, a cache server goes offline), it does not affect the entire system.
- Step 2. Propose high-level design and get buy-in
  - Where to put the rate limiter?
    - Client-side implementation
      - can easily be forged by malicious actors
    - Server-side implementation
      - Rate limiter at the API servers (on the server-side)

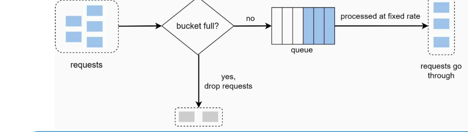
      - Rate limiter middleware (in a gateway)

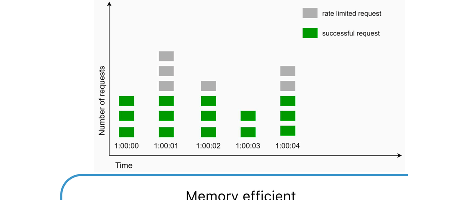
        - API gateway usually suports
          - rate limiting
          - SSL termination
          - authentication
          - IP whitelisting
          - servicing static content
  - Algorithms for rate limiting
    - Token bucket

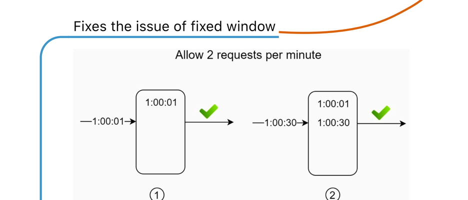
      - Params
        - 1. Bucket size: the maximum number of tokens allowed in the bucket
        - 2. Refill rate: number of tokens put into the bucket every second
          - ⚠️ Actually there is a third parameter - internal
      - Pros
        - The algorithm is easy to implement.
        - Memory efficient
        - Token bucket allows a burst of traffic for short periods
          - A request can go through as long as there are tokens left
      - Cons
        - It might be challenging to tune two parameters properly
    - Leaking bucket

      - Params
        - 1. Bucket size: it is equal to the queue size. The queue holds the requests to be processed at a fixed rate
        - 2. Outflow rate: it defines how many requests can be processed at a fixed rate, usually in seconds
      - Pros
        - Memory efficient given the limited queue size
        - Requests are processed at a fixed rate therefore it is suitable for use cases that a stable outflow rate is needed
      - Cons
        - A burst of traffic fills up the queue with old requests, and if they are not processed in time, recent requests will be rate limited
        - It might not be easy to tune two parameters properly
    - Fixed window counter

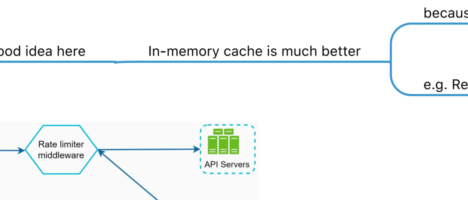
      

      - Pros
        - Memory efficient
        - Easy to understand
        - Resetting available quota at the end of a unit time window fits certain use cases.
      - Cons
        - Spike in traffic at the edges of a window could cause more requests than the allowed quota to go through
          - ❓ Q: the same problem exists with Token bucket and Leaking buckets!?
    - Sliding window log
      - Fixes the issue of fixed window

      - ❓ Q: why do we keep 1:00:50 in log if it was rejected? We would not serve promised number of requests because of this!
        - I would not log rejected requests!
      - Pros
        - Rate limiting implemented by this algorithm is very accurate
          - In any rolling window, requests will not exceed the rate limit
      - Cons
        - The algorithm consumes a lot of memory because even if a request is rejected, its timestamp might still be stored in memory
    - Sliding window counter

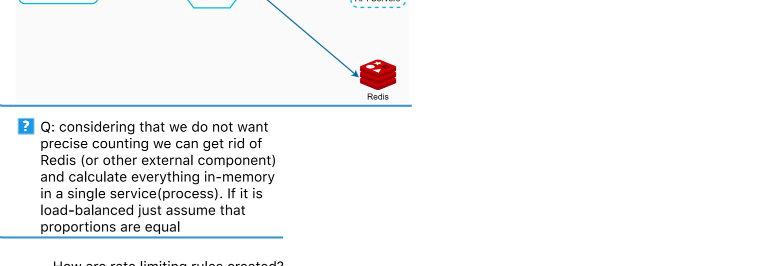
      - Approach #1 (other approaches exist)
        - Number of requests =
          - Requests in current window + requests in the previous window * overlap percentage of the rolling window and previous window
            - Using this formula, we get 3 + 5 * 0.7% = 6.5 request. Depending on the use case, the number can either be rounded up or down
      - Pros
        - It smooths out spikes in traffic because the rate is based on the average rate of the previous window
          - ❓ Q: Is it good? I thought we wanted precise limitation
        - Memory efficient
      - Cons
        - It only works for not-so-strict look back window
          - It is an approximation of the actual rate because it assumes requests in the previous window are evenly distributed
            - However, this problem may not be as bad as it seems: At Cloudflare only 0.003% requests are wrongly accepted
  - High-level architecture
    - Databases are not good idea here
      - In-memory cache is much better
        - because
          - They are fast
          - They provide time-based expiration strategy
        - e.g. Redis is often chosen
          - INCR
          - EXPIRE

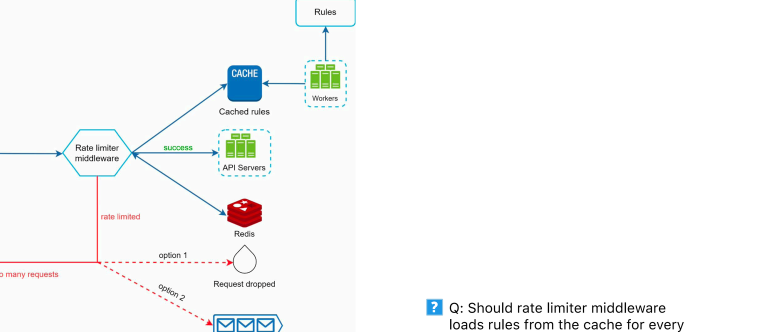
    - ❓ Q: considering that we do not want precise counting we can get rid of Redis (or other external component) and calculate everything in-memory in a single service(process). If it is load-balanced just assume that proportions are equal
- Step 3. Design deep dive
  - Unanswered questions during step #2
    - How are rate limiting rules created? Where are the rules stored?
    - How to handle requests that are rate limited?
  - Rate limiting rules
    - Examples from Lyft's open sourced Rate Limiter

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

  - Exceeding the rate limit
    - We can decide to return HTTP 429 response code or can postpone the request to process it later
    - Rate limiter headers
      - The rate limiter returns the following HTTP headers to clients
        - X-Ratelimit-Remaining
          - The remaining number of allowed requests within the window
        - X-Ratelimit-Limit
          - It indicates how many calls the client can make per time window
        - X-Ratelimit-Retry-After
          - The number of seconds to wait until you can make a request again without being throttled
            - ❓ Q: we simply return a window here or a real wait time for one next request?
  - Detailed design

    - ❓ Q: Should rate limiter middleware loads rules from the cache for every request??? This is very inefficient
  - Rate limiter in a distributed environment
    - Two challenges
      - Race condition

        - Two strategies
          - Lua script
          - Sorted sets data structure in Redis
            - ❓ Q: How would this help?
          - ❓ Q: I believe Redis has atomic operation for this already (kind of "build-in pre-compiled Lua script")
      - Synchronization issue
        - ⚠️ When multiple rate limiter servers are used, synchronization is required
        - Solutions
          - Use sticky sessions
          - Use centralised Redis
  - Performance optimization
    - 1. A multi-data center setup
      - It is crucial for a rate limiter
        - because latency is high for users located far away from the data center
    - 2. Synchronize data with an eventual consistency model
  - Monitoring
    - ⚠️ It is important to gather analytics data to check whether the rate limiter is effective
    - We want to make sure
      - The rate limiting algorithm is effective
      - The rate limiting rules are effective
- Step 4. Wrap up
  - Extra points to discuss
    - Hard vs soft rate limiting
      - Hard
        - The number of requests cannot exceed the threshold
      - Soft
        - Requests can exceed the threshold for a short period
    - Rate limiting at different levels
      - Layer 1: Physical layer
      - Layer 2: Data link layer
      - Layer 3: Network layer
        - e.g. can apply rate limiting by IP addresses using Iptables
      - Layer 4: Transport layer
      - Layer 5: Session layer
      - Layer 6: Presentation layer
      - Layer 7: Application layer
        - ⚠️ We covered only this level this this chapter!
    - Avoid being rate limited
      - 👍 Design your client with best practices
        - Use client cache to avoid making frequent API calls
        - Understand the limit and do not send too many requests in a short time frame
        - Include code to catch exceptions or errors so your client can gracefully recover from exceptions
        - Add sufficient back off time to retry logic