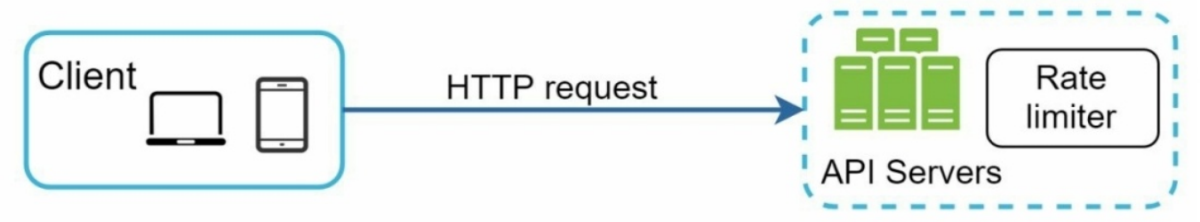
Design a Rate Limiter

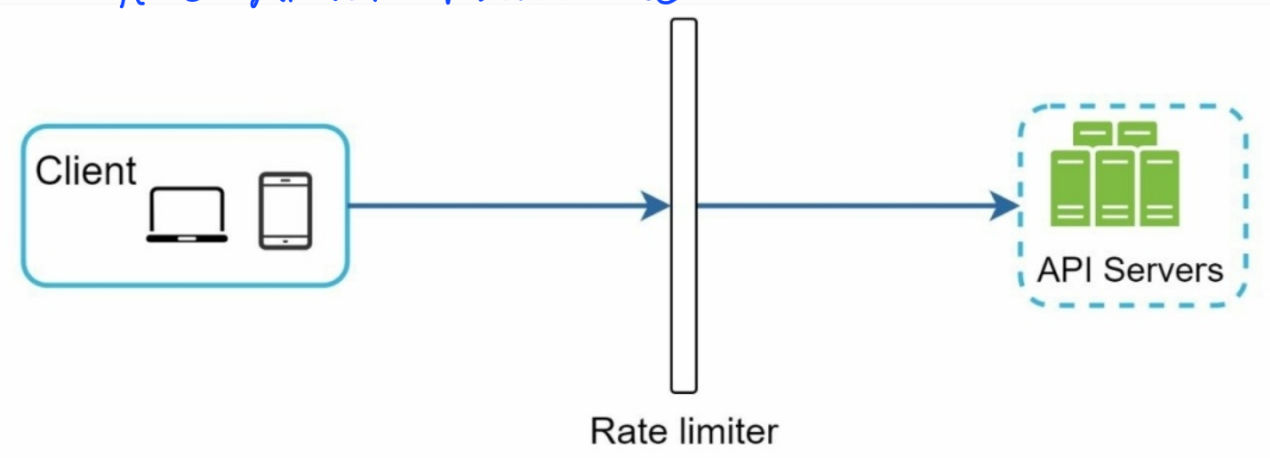
- Step 1 - Problem understanding and design scope
  - Requirements
    - Accurately limit excessive requests.
    - Low latency, should not slow down HTTP response time.
    - Use as little memory as possible.
    - Distributed rate limiting. Rate limiter can be shared across multiple servers or processes.
    - Exception Handling. Show clear exceptions to users when their requests are throttled.
    - High fault tolerance. If there are any problems with the rate limiter, 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, client is an unreliable place to enforce rate limiting because client requests can easily be forged.

    - Server side Implementation -



- Rate Limiter Middleware -

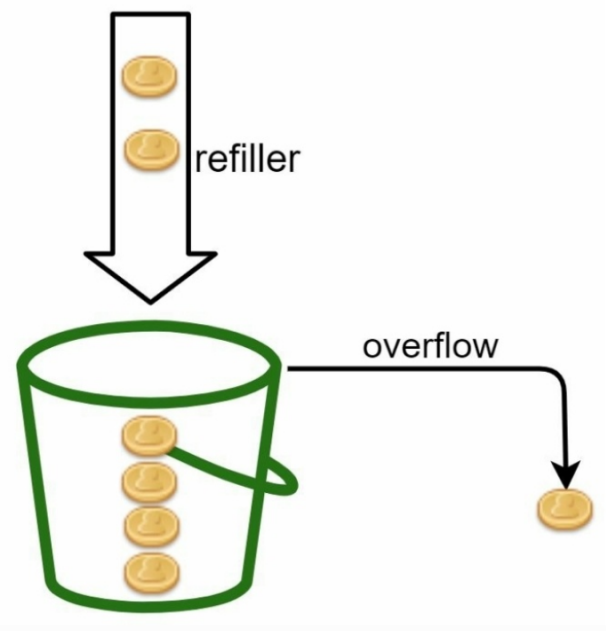


- While designing a rate limiter, an important question to ask is where the rate should be implemented, on server side or as a gateway.
- Here are a few general guidelines -
  - Evaluate your current technology stack, such as programming language, cache service etc. Make sure your current programming language is efficient to implement rate limiter on server side.
  - Identify rate limiting algorithm that fits your business needs. Implementing on server side gives you full control of the algorithm.
  - If you have already used microservice architecture and included an API gateway in the design to perform authentication, IP whitelisting etc., you may add a rate limiter to the API gateway.
  - Building your own rate limiting service takes time. If you do not have enough engineering resources to implement a rate limiter, a commercial API gateway is a better option.

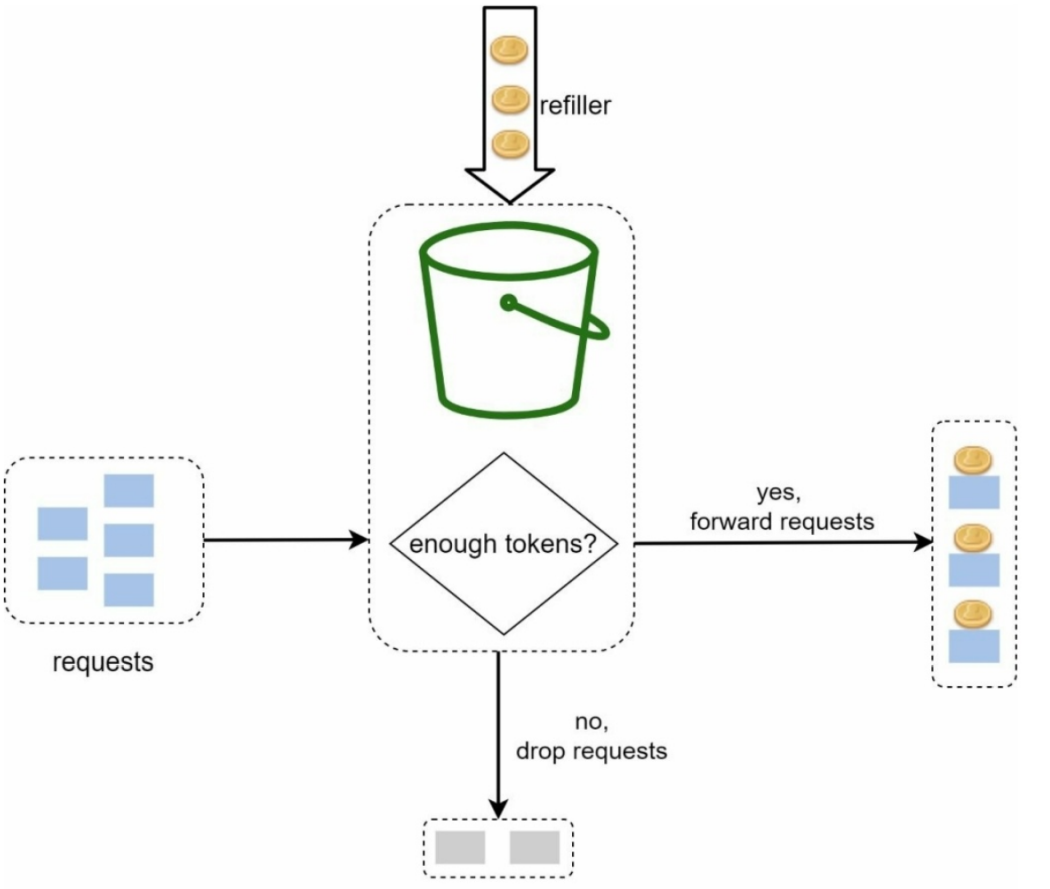
- Algorithms for Rate Limiting

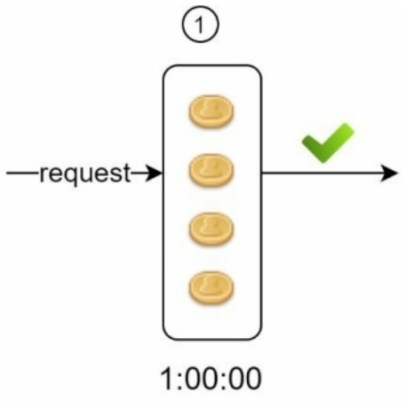
1. Token Bucket Algorithm

- It is widely used for rate limiting.
- The token bucket algorithm works as follows -
  - A token bucket is a container that has pre-defined capacity. Tokens are put in a bucket at preset rates periodically. Once the bucket is full, no more tokens are added. Once the bucket is full, extra tokens will overflow.



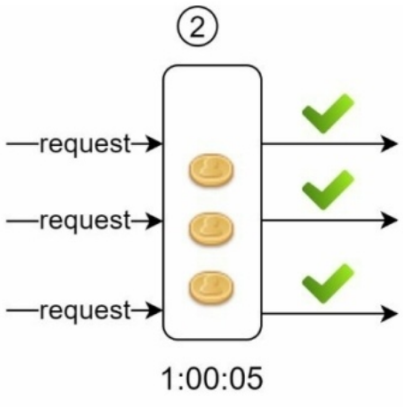
  - Each request consumes one token. When a request arrives, we check if there are enough tokens in the bucket.
    - If there are enough tokens, we take one token out for each request, and the request goes through.
    - If there are not enough tokens, the requests are dropped.





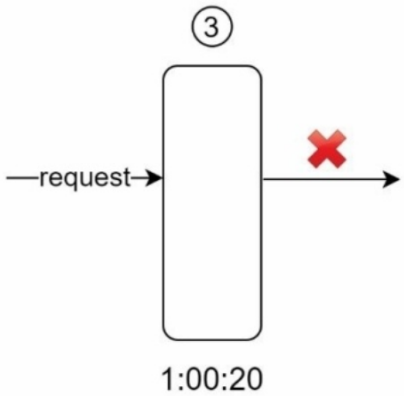
1:00:00

- Start with 4 tokens
- The request will go through
- 1 token is consumed



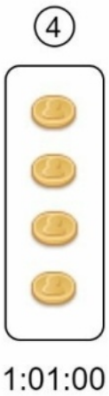
1:00:05

- Start with 3 tokens
- All three requests will go through
- 3 tokens are consumed



1:00:20

- Start with 0 token
- The request will be dropped.



1:01:00

- 4 tokens are refilled at 1 minute interval

- The token bucket algorithm takes 2 parameters:
  - Bucket Size - Maximum number of tokens allowed in the bucket.
  - Refill Rate - Number of tokens put into the bucket every second.

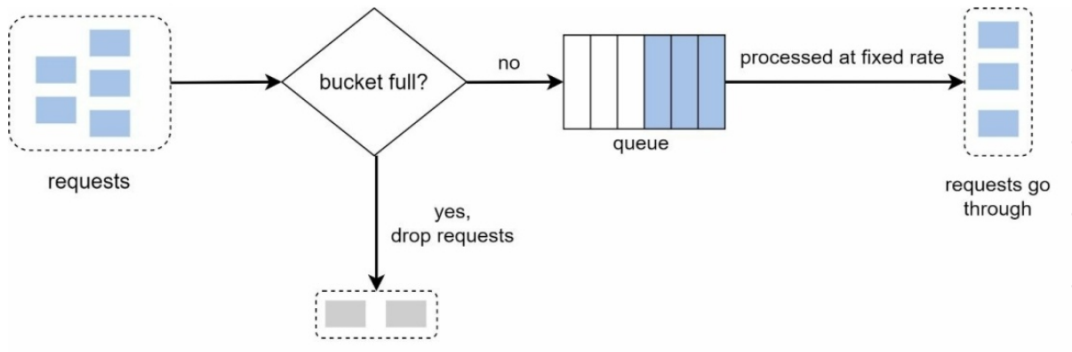
- How many buckets are needed? This depends on the rate limiting rules.
  - It is usually necessary to have different buckets for different API endpoints. For Eg.- If a user is allowed to make 1 post per second, add 150 friends per day, like 5 posts per second, 3 buckets are required for each user.
  - If throttle needs to be done on IP addresses, each IP address requires a bucket.
  - If the system allows a maximum of 10,000 requests per second, it makes sense to have a global bucket shared by all requests.

- Pros:
  - Algorithm is easy to implement.
  - Memory efficient.
  - Token bucket allows a burst of traffic for short periods. A request has to go through as long as there are tokens left.

- Cons:
  - It is challenging to tune the parameters of the algorithm.

2. Leaking Bucket Algorithm

- Leaking Bucket Algorithm is similar to the token bucket except that requests are processed at a fixed rate. It is usually implemented with a FIFO queue. The algorithm works as follows:
  - When a request arrives, the system checks if the queue is full. If it is not full, the request is added to the queue.
  - Otherwise, the request is dropped.
  - Requests are pulled from the queue and processed at regular intervals.



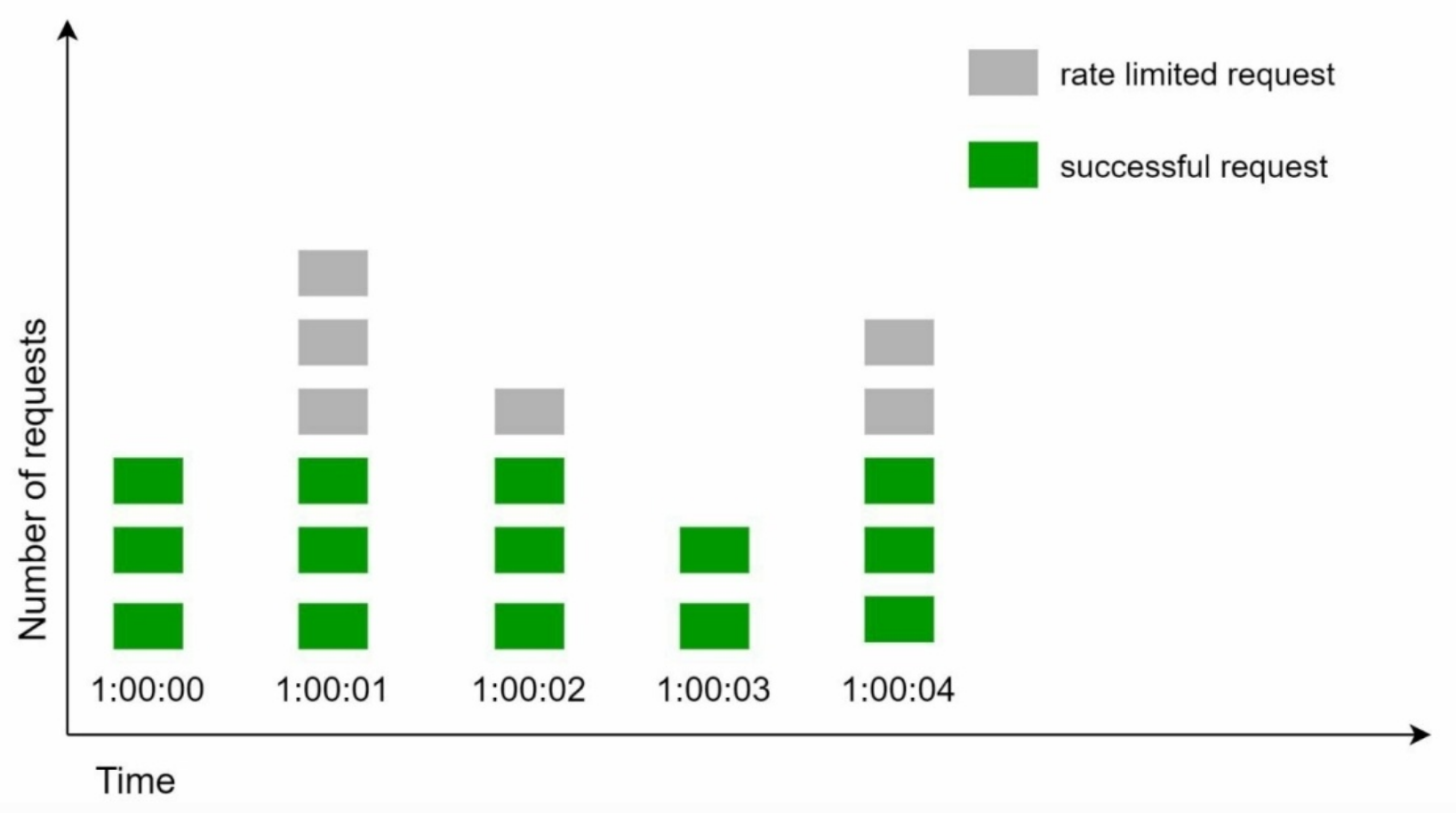
- Leaking Bucket algorithm takes the following 2 parameters:
  - Bucket size: it is equal to queue size. The queue holds the requests to be processed at a fixed rate.
  - Outflow rate: It defines how many requests can be processed at a fixed rate, usually in seconds.

- 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.

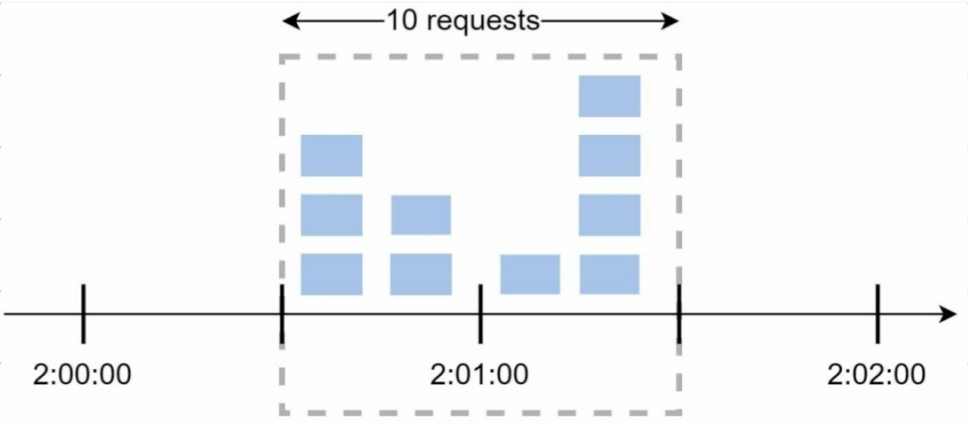
- Cons
  - A burst of traffic fills up the queue with old requests, and if they are not processed in time, recent requests will be rate limited.
  - The 2 parameters might not be easy to tune properly.

3. Fixed Window Counter Algorithm

- Fixed Window Counter Algorithm works a follows:
  - The algorithm divides the timeline into fixed size time windows and assign a counter for each window.
  - Each request increment the counter by one.
  - Once the counter reaches the pre-defined threshold, new requests are dropped until a new time window starts.



- A major problem with this algorithm is that a burst of traffic at the edges of time windows could cause more requests than allowed quota to go through.



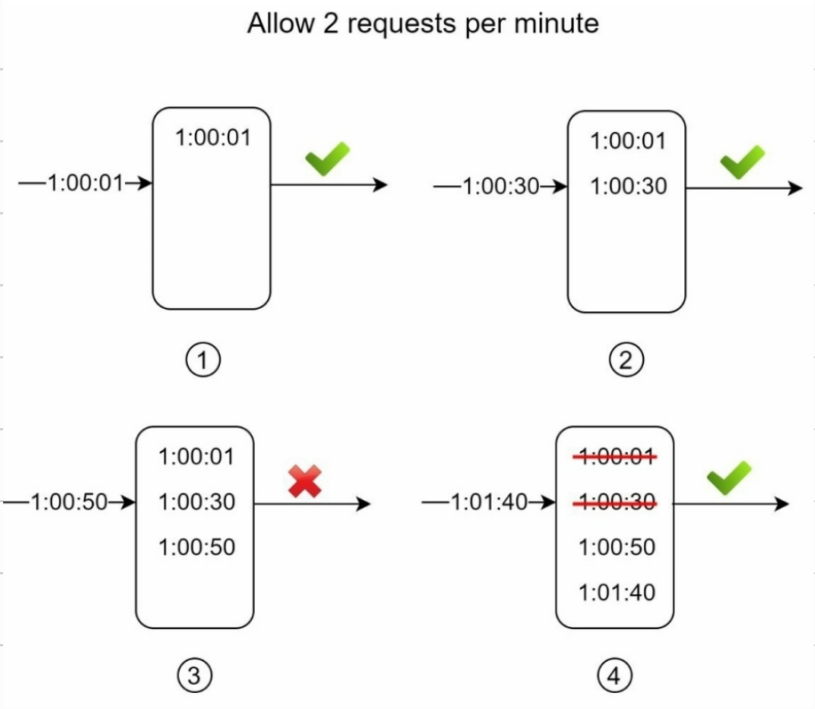
- The system allows a maximum of 5 requests per minute, and the quota resets in a minute.
  - 5 requests between 2:00:00 and 2:01:00.
  - 5 requests between 2:01:00 and 2:02:00
  - Between 2:00:30 and 2:01:30, 10 requests go through, twice the allowed requests.

- 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.

4. Sliding Window Log Algorithm

- Sliding Window Log Algorithm works as follows:
  - The algorithm keeps track of requests timestamps. Timestamp data is usually kept in cache, such as sorted sets of Redis.
  - When a new request comes in, remove all the outdated timestamps. Outdated timestamps are defined as those older than the start of current time window.
  - Add timestamp of the new request to the log.
  - If the log size is the same or lower than the allowed count, a request is accepted. Otherwise, it is rejected.



- Pros
  - Rate limiting implemented by this algorithm is very accurate. In any rolling window, request will not exceed rate limit.
- Cons
  - Consumes a lot of memory even if request is rejected,

its timestamp might still be stored in memory.

5. Sliding Window Counter Algorithm

- Sliding window counter algorithm is a hybrid approach that combines the fixed window counter and sliding window log.

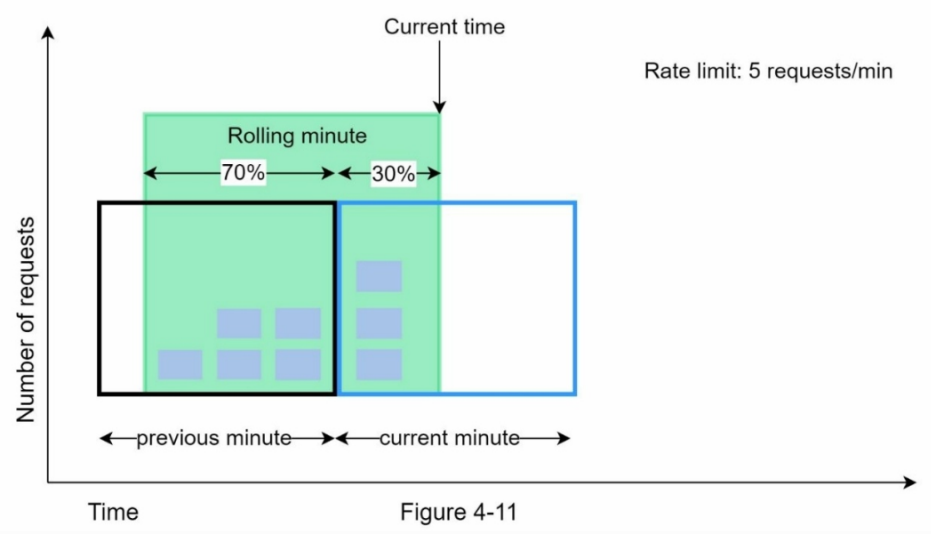


Figure 4-11

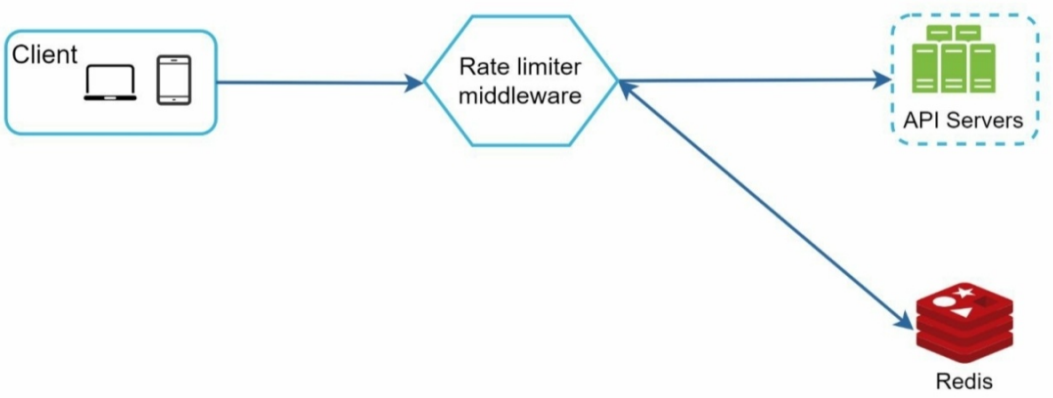
- Assume rate limiter allows a maximum of 7 requests per minute, and there are 5 requests in the previous minute and 3 in the current minute. For a new request that arrive at a 30% position in the current minute, the number of requests in the rolling window is calculated using following formula:
  - 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. Here it is rounded down to 6.

- Since the rate limiter allows a maximum of 7 requests per minute, the current request can go through. However, the limit will be reached after receiving 1 more request.

- Pros
  - It smooths out spikes in traffic because the rate is based on the average rate of the previous window.
  - 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 previous window are evenly distributed.

- High Level Architecture
  - Where shall we store the counters? Using the database is not a good idea due to slowness of disk access. In-memory cache is chosen because it is fast and supports time-based expiration strategy. For instance, Redis is a popular option to implement rate limiting. It is an in-memory store that offers 2 commands: INCR and EXPIRE.
    - INCR: It increases the stored counter by 1.
    - EXPIRE: It sets a timeout for the counter. If the timeout expires, the counter is automatically deleted.



- The client sends a request to rate limiting middleware.
- Rate limiting middleware fetches the counter from the corresponding bucket in Redis and checks if the limit is reached or not.
  - If the limit is reached, the request is rejected.
  - If the limit is not reached, the request is sent to API servers. Meanwhile, the system increments the counter and saves it back to Redis.

- Step 3 - Design Deep Dive
  - The high level design doesn't answer the following questions:
    - How are rate limiting rules created? Where are the rules stored?
    - How to handle requests that are rate limited?

  - Rate Limiting Rules
    - Lyft open-sourced their rate-limiting component. Following is an example:

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

- This rule shows that clients are not allowed to login more than 5 times in 1 minute. Rules are generally written in configuration files and saved on disk.

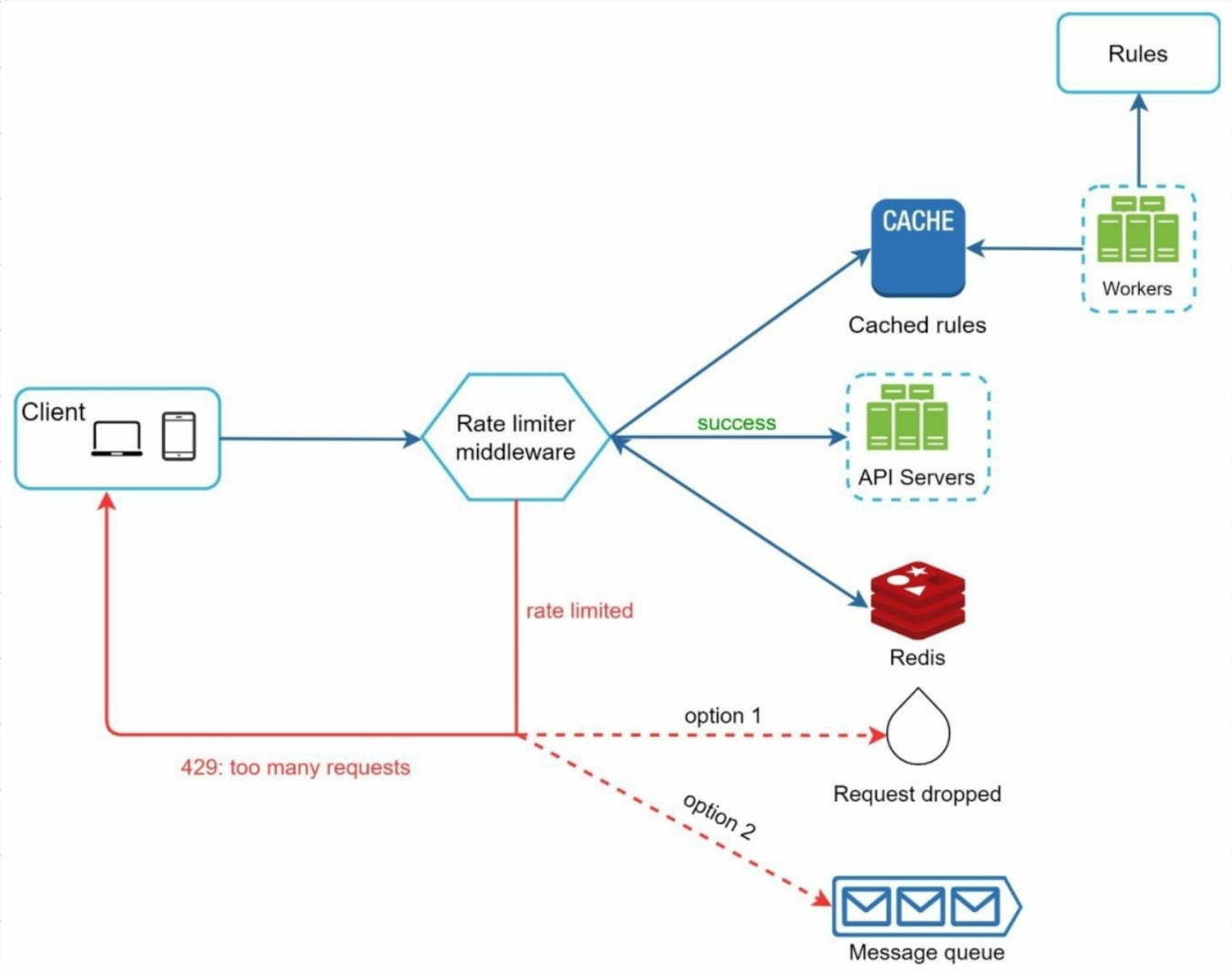
- Exceeding the rate limit
  - In case a request is rate limited, APIs return a HTTP response code 429 (too many requests) to the client.
  - Depending on the use cases, we may enqueue the rate-limited requests to be processed later.

- Rate Limiter Headers
  - How does a client know whether it is being throttled? 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.

- When a user has sent too many requests, a 429 too many requests error and X-Ratelimit-Retry-After header are returned to the client.

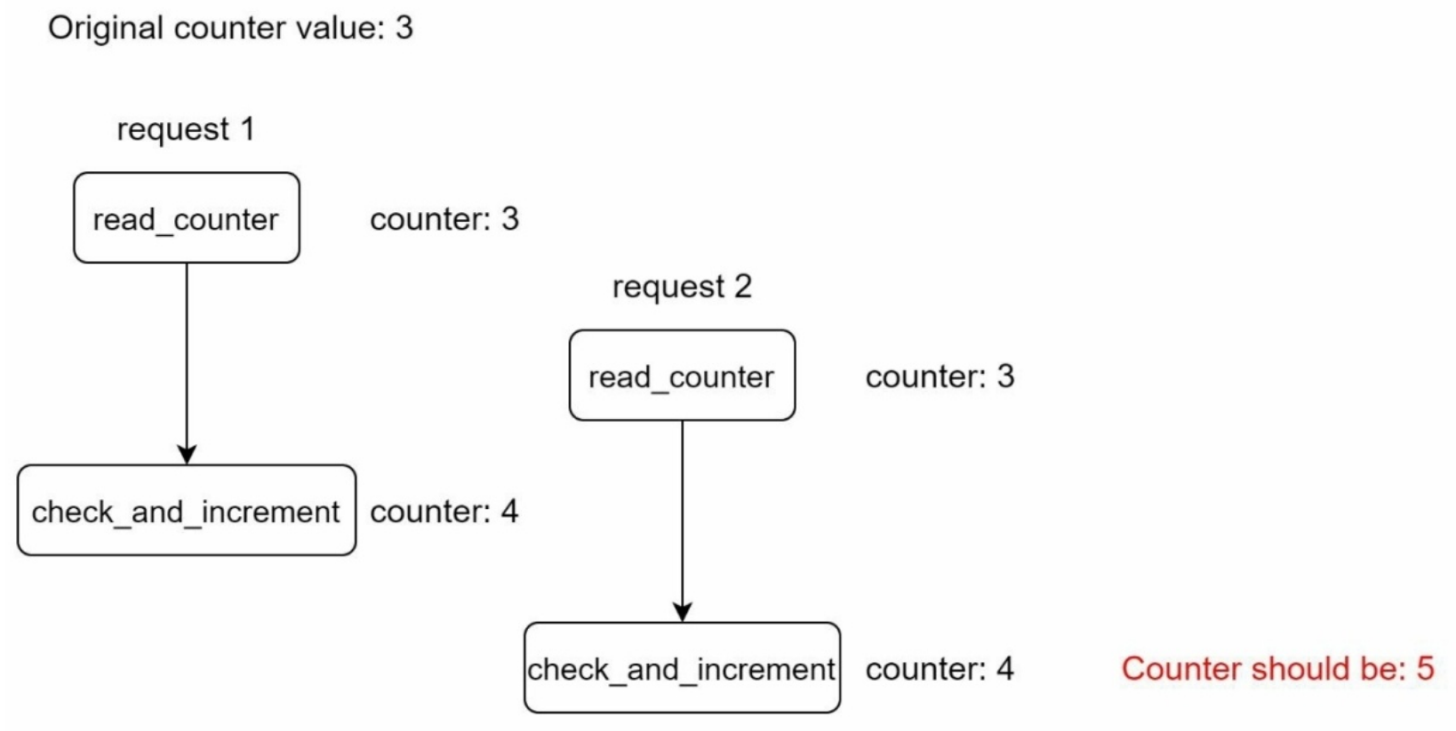
- Detailed Design



- Rules are stored on the disk. Workers frequently pull rules from the disk and store them in cache.
- When a client sends a request to the server, the request is sent to the rate limiter middleware first.
- Rate limiter middleware loads rules from the cache. It fetches counters and last request timestamp from Redis cache. Based on the response, the ratelimiter decides:
  - If the request is not rate limited, it is forwarded to API servers.
  - If the request is rate limited, the rate limiter returns 429 too many requests error to the client. In the meantime, In the meantime, the request is either dropped or added to the queue.

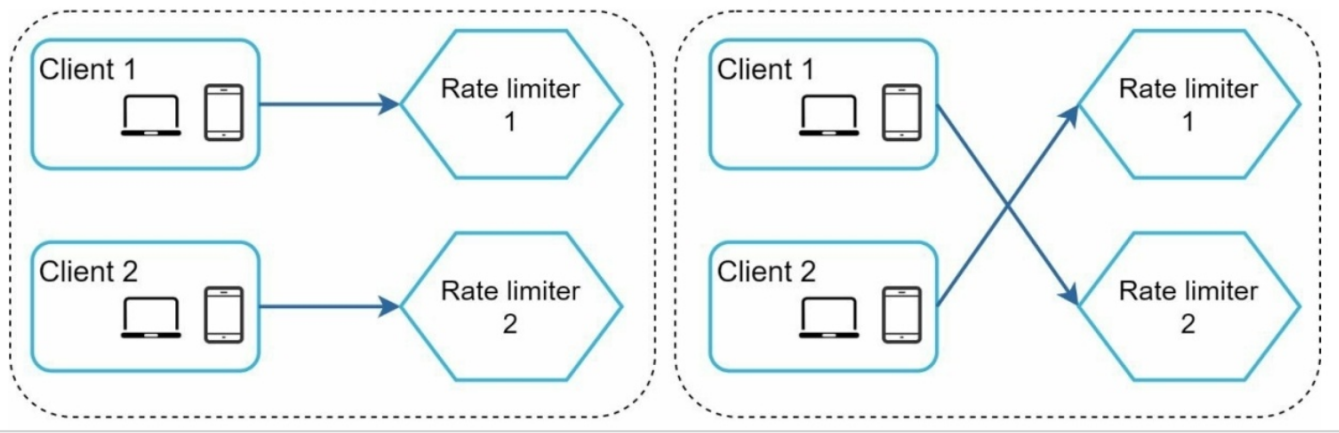
- Rate Limiter in a distributed environment
  - Building a rate limiter that works in a single server environment is not difficult. However, scaling the system to support multiple servers and concurrent threads is a different story. There are 2 challenges:
    - Race Condition
    - Synchronization Issue

- Race Condition
  - Race conditions can happen in a highly concurrent environment as shown below:

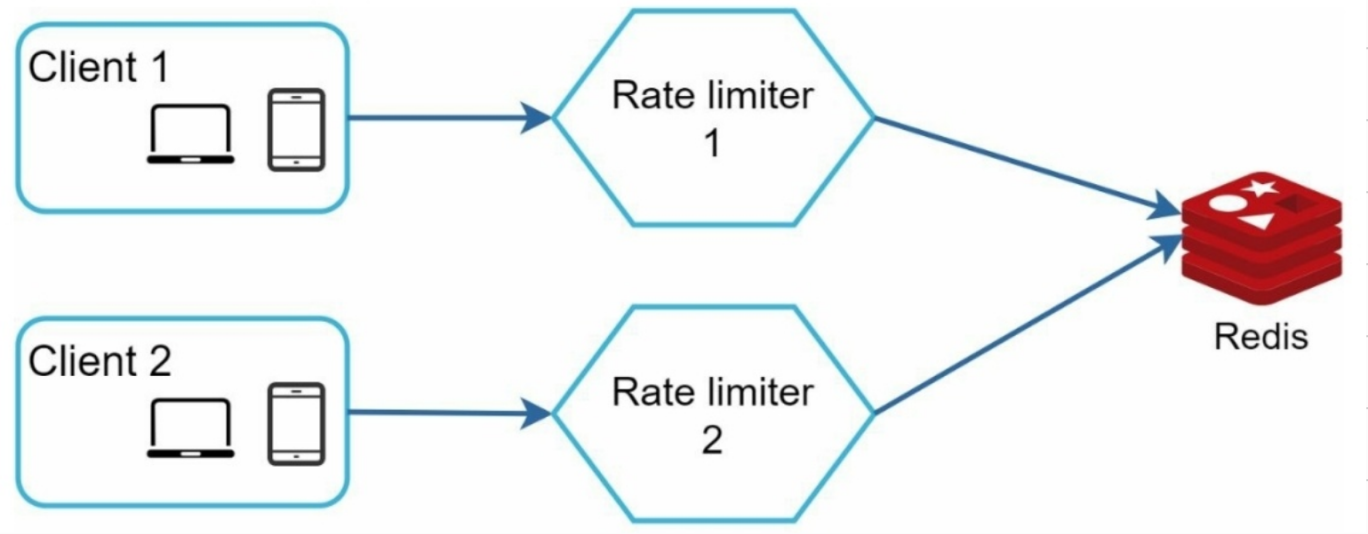


  - Locks are most obvious solution for solving race condition. However, locks will significantly slow down the system.
  - 2 strategies are commonly used to solve the problem: Lua Script and sorted set data structure in Redis.

- Synchronization Issue
  - To support million of users, one rate limiter server might not be enough to handle the traffic. When multiple rate limiter servers are used, synchronization is required.
  - On the left side, client 1 sends requests to rate limiter 1 and client 2 sends to rate limiter 2. As the web tier is stateless, clients can send requests to a different rate limiter. If no synchronization happens rate limiter cannot work properly.



  - 1 possible solution is to use sticky sessions that allows a client to send traffic to same rate limiter. A better approach is to use centralized data stores like Redis.



- Performance Optimization
  - Multi-data center setup is crucial for a rate limiter because latency is high for users located far away from data center.
  - Synchronize data with an eventual consistency model.

- Monitoring
  - Monitoring is needed for following reasons:
    - The rate limiting algorithm is effective.
    - The rate limiting rules are effective.

- Step 4 - Wrap Up

  - Algorithms
    - Token Bucket
    - Leaking Bucket
    - Fixed Window
    - Sliding Window Log
    - Sliding Window Counter

  - Additional talking points:
    - Hard vs Soft rate limiting
      - Hard: The number of requests cannot exceed the threshold.
      - Soft: Request can exceed a threshold for a short period.
    - It is possible to apply rate limiting at other layers. For eg.- Rate limiting can be applied on IP address using IP Tables.
    - Avoid being rate limited. 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 period.
- Include code to catch exceptions or errors so your client can gracefully recover from exceptions.
- Add sufficient back off time to retry logic.