Chapter 4 Design a rate limiter

What is rate limiter?

- Rate limiter is used to control the rate of traffic sent by a client or a service.
- In HTTP world, a rate limiter limits the number of clients allowed to be sent over a specified period.
- If the API request count exceeds the threshold defined by the rate limiter, all the excess calls are blocked

Examples of rate limiter

- A user can write no more than 2 posts per second
- You can create a maximum of 10 accounts per day from the same IP address
- You can claim rewards no more than 5 times per week from the same devices.

Benefits of using an API rate limiter

- Prevent resource starvation caused by Denial of Service (DoS) attack
  - DoS attack is a malicious attempt to disrupt the normal functioning of a targeted server. The aim is to make the targeted resource unavailable to its intended users.
- Reduce cost.
  - Limiting excess requests => fewer servers, allocating more resources to high priority APIs. eg. per-call basis for external APIs.
- Prevent servers from being overloaded.

Design a rate limiter

Step 1 - Understand the problem and establish design scope

Questions to ask

- What kind of rate limiter? Client side or server side?
- Does rate limiter throttle API requests based on IP, the user ID, or other properties?
- Scale of the system? startup vs big company with a large user base?
- Will the system work in a distributed environment?
- separate service or be implemented in application code?
- Inform users who are throttled?

Requirements

- Accurately limit excessive requests
- Low latency => 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 requests are throttled
- High fault tolerance. => problem does not affect the entire system

Step 2 - Propose high-level design and get buy-in

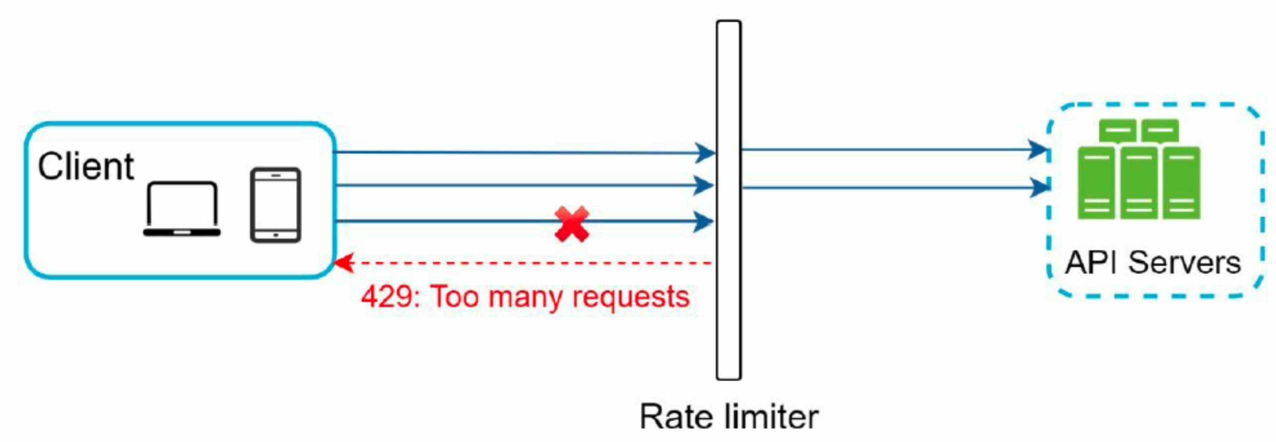
Where to put the rate limiter?

Client-side versus **Server-side** versus **Middleware**

- **Client-side**: unreliable => 1. client requests can easily be forged by malicious actors. 2. no control over the client implementation
- **Server-side**

- **Middleware/API gateway**
  - Rate limiting is usually implemented within a component called API gateway
  - API gateway: fully managed service that supports rate limiting, SSL termination, authentication, IP whitelisting, servicing static content.
  - eg. API allows 2 requests per second, a client sends 3 requests. => the first two requested are routed to API servers. the rate limiter middleware throttles the third requests

and returns a HTTP status code 429 (too many requests).



Benefits of choosing Middleware over Server-side

- **Centralized Management.** Middleware can manage rate limiting across multiple servers, easy to scale out.
- **Isolation**. Separate rate limiting from the application logic enhance security, changes can be applied without modifying application code.
- **Early request filtering**. Reduce malicious traffic before it reaches backend servers.
- **Enhanced monitoring**. Centralized logging, metrics

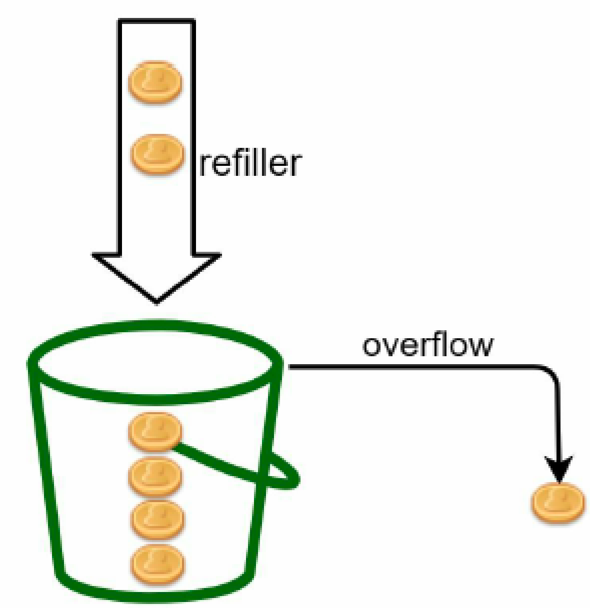
Things to consider when choose where to put rate limiter

- **Technology stack** => programming language, cache service => language is **efficient** to implement rate limiting on the server-side
  - Lack of Concurrent support => Python's Global Interpreter Lock (GIL) can be a limitation in multi-threaded applications. => Difficulty in high volume of concurrent requests
  - Poor performance in handling high traffic => PHP => traditionally used in synchronous, request-per-process model, struggle under heavy load without significant architectural adjustment => Increased latency, higher resource consumption
  - Limited support for Asynchronous I/O=> Ruby => Inefficiencies in managing non-blocking I/O operations => slower request processing
  - Resource-intensive => Java with its JVM warm-up and memory usage => higher operational costs, potential scalability issues
  - Lack of Native support for rate limiting libraries => older or less popular languages => increased dev time and potential bugs
  - Good examples:
    - **Node.js**

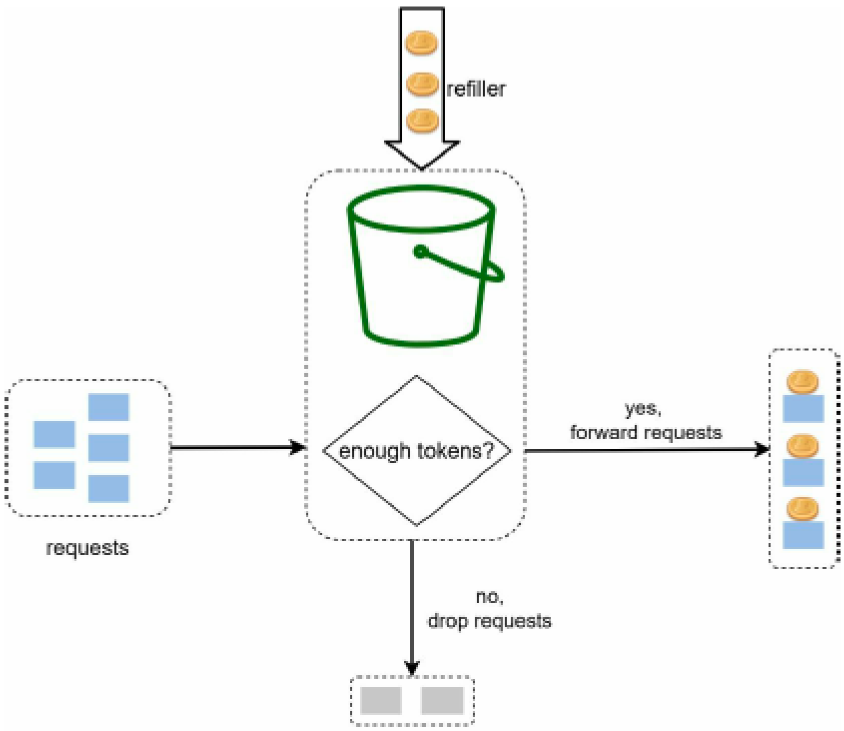
            - excellent support for asynchronous I/O, large ecosystem of libraries `express-rate-limit`, efficient of concurrent connection
            - Use Case => Real-time application: chat servers, streaming services
        - **Go (Golang)**
            - Strong support for concurrency with goroutines, low memory footprint, high performance
            - Use Case => High-performance microservices, APIs with heavy traffic
        - **Java**
            - mature ecosystem, robust concurrency support with tools like `RateLimiter` from Guava
            - Use Case => Enterprise-level applications, large-scale distributed systems
        - **Python (with certain frameworks)**
            - Asynchronous frameworks `aiohttp`, `FastAPI`, `Tornado` can handle rate limiting efficiency despite GIL
            - Use Case => Web applications, data-intensive services
- Rate limiting **Algorithm** that **fits your business needs**
    - Token bucket (Amazon, Stripe)
    - Leaking bucket
    - Fixed window counter
    - Sliding window log
    - Sliding window counter

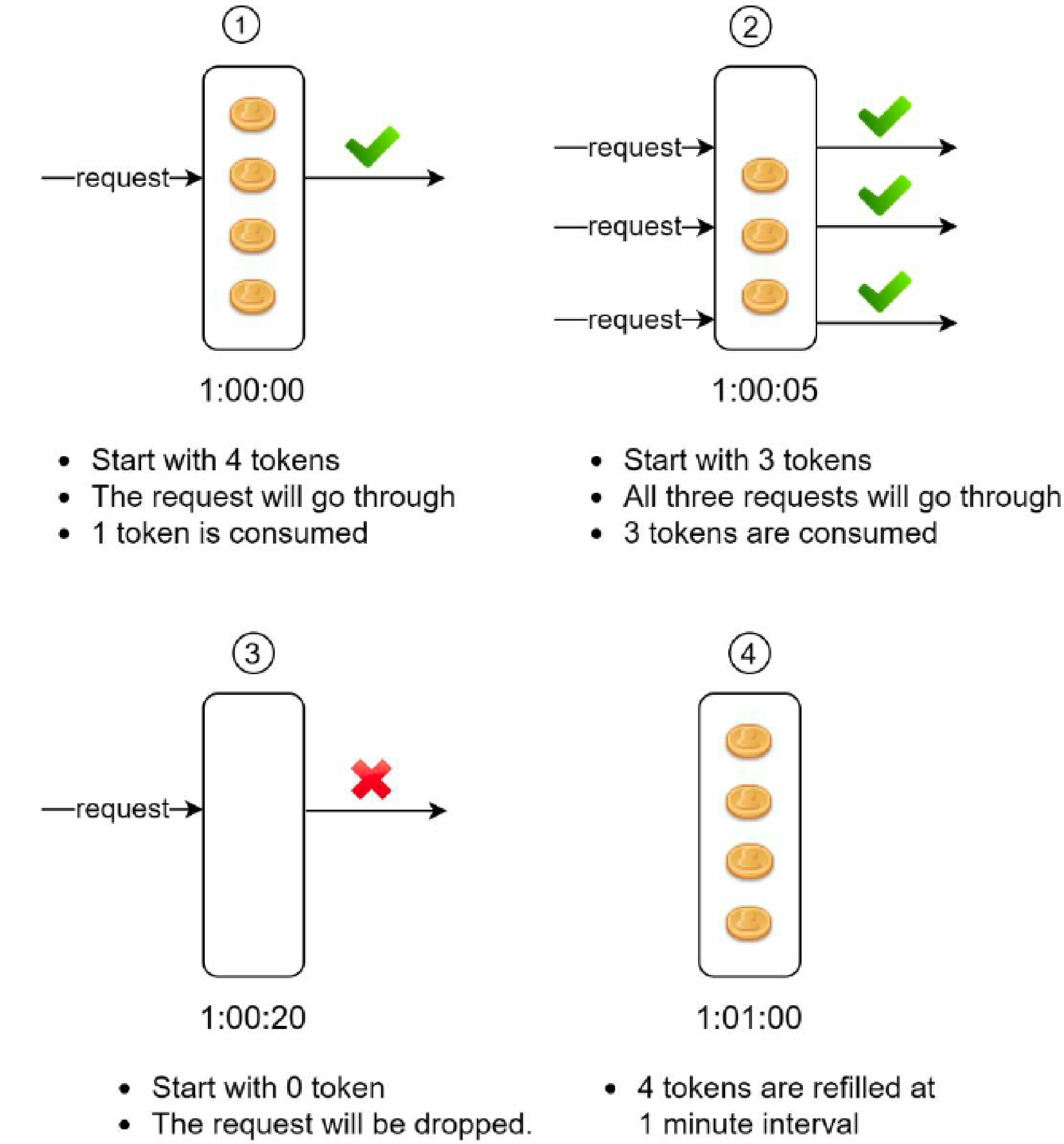
Token bucket (Amazon, Stripe)

- A container that has pre-defined capacity. Tokens are put in the bucket at preset rates periodically. Once bucket is full -> no more tokens are added -> extra tokens will overflow



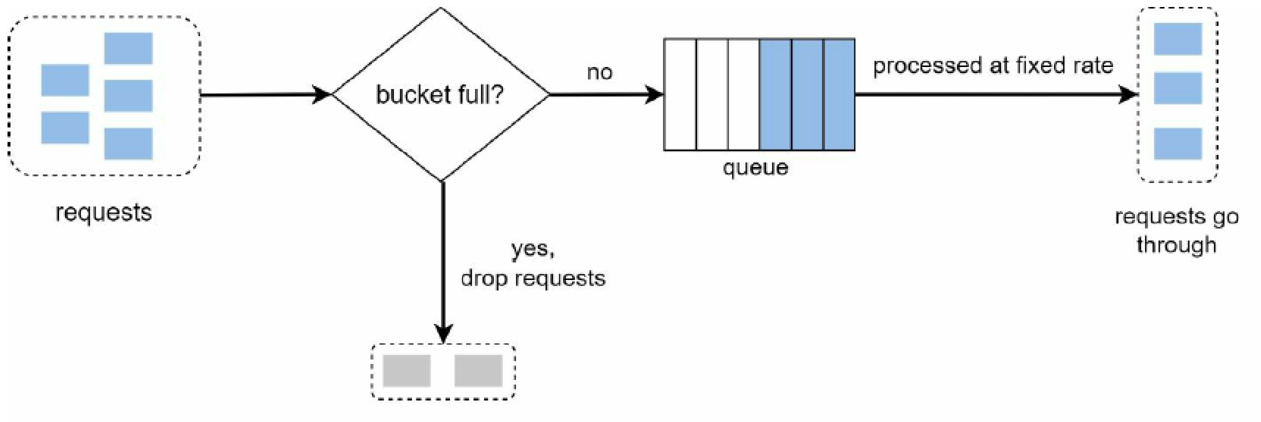
- eg. Capacity is 4, the refiller puts 2 tokens into the bucket every second.
- Each request consumes one token. When a request arrives, we check if there are enough tokens in the buckets
- If there are enough tokens -> we take one token out for each request, and the request goes through
- If there are not enough tokens, the request is dropped





- Parameters:
  - **Bucket size**: the maximum number of tokens allowed in the bucket
  - **Refill rate**: number of tokens put into the bucket every second
- Pros
  - 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
  - Two parameters required. Challenging to tune them properly

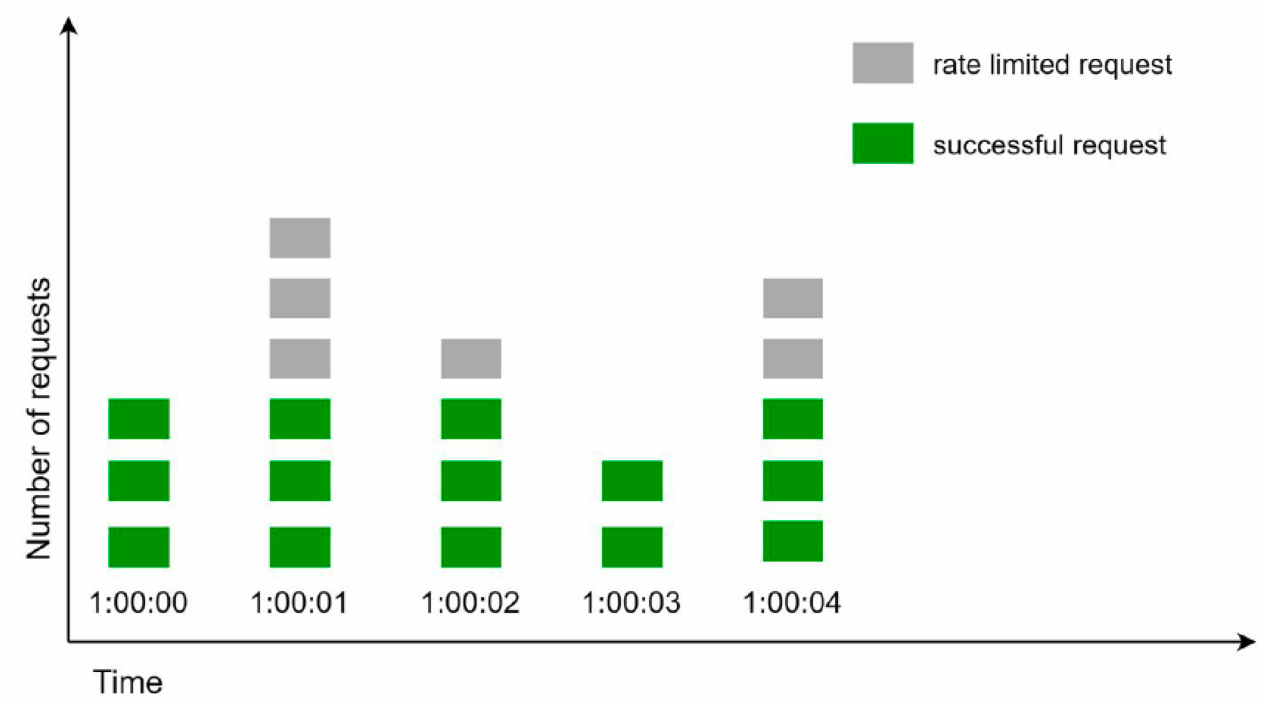
Leaking bucket (Shopify)

- Similar to token bucket except that requests are processed at a fixed rate.
- Usually implemented with FIFO queue
- System checks if the queue is full. If not, the request is added to the queue. Otherwise, the request is dropped
- Requests are pulled from the queue and processed at regular intervals
- 

- 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 -> 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
  - Two parameters required. Challenging to tune them properly

Fixed window counter

- Divides timeline into fix-sized time window and assign a counter for each window.
- Each request increments the counter by one
- Once the counter reaches the pre-defined threshold, new requests are dropped until a new time window starts

- 

- eg. time unit is 1 second, system allows a maximum of 3 requests per second. If more than 3 requests are received, extra requests are dropped.

  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. Consider the following case:

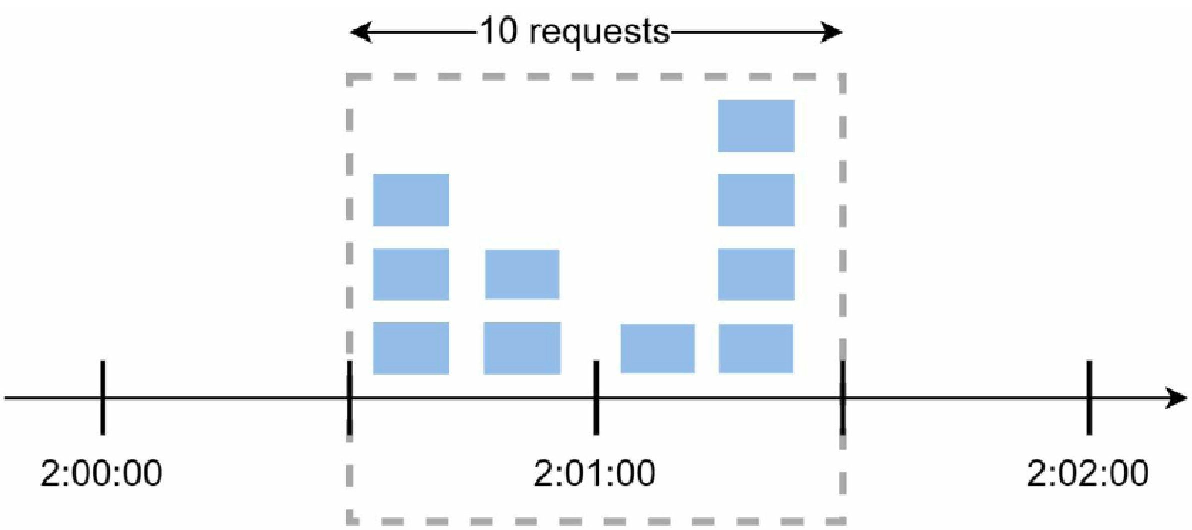
  

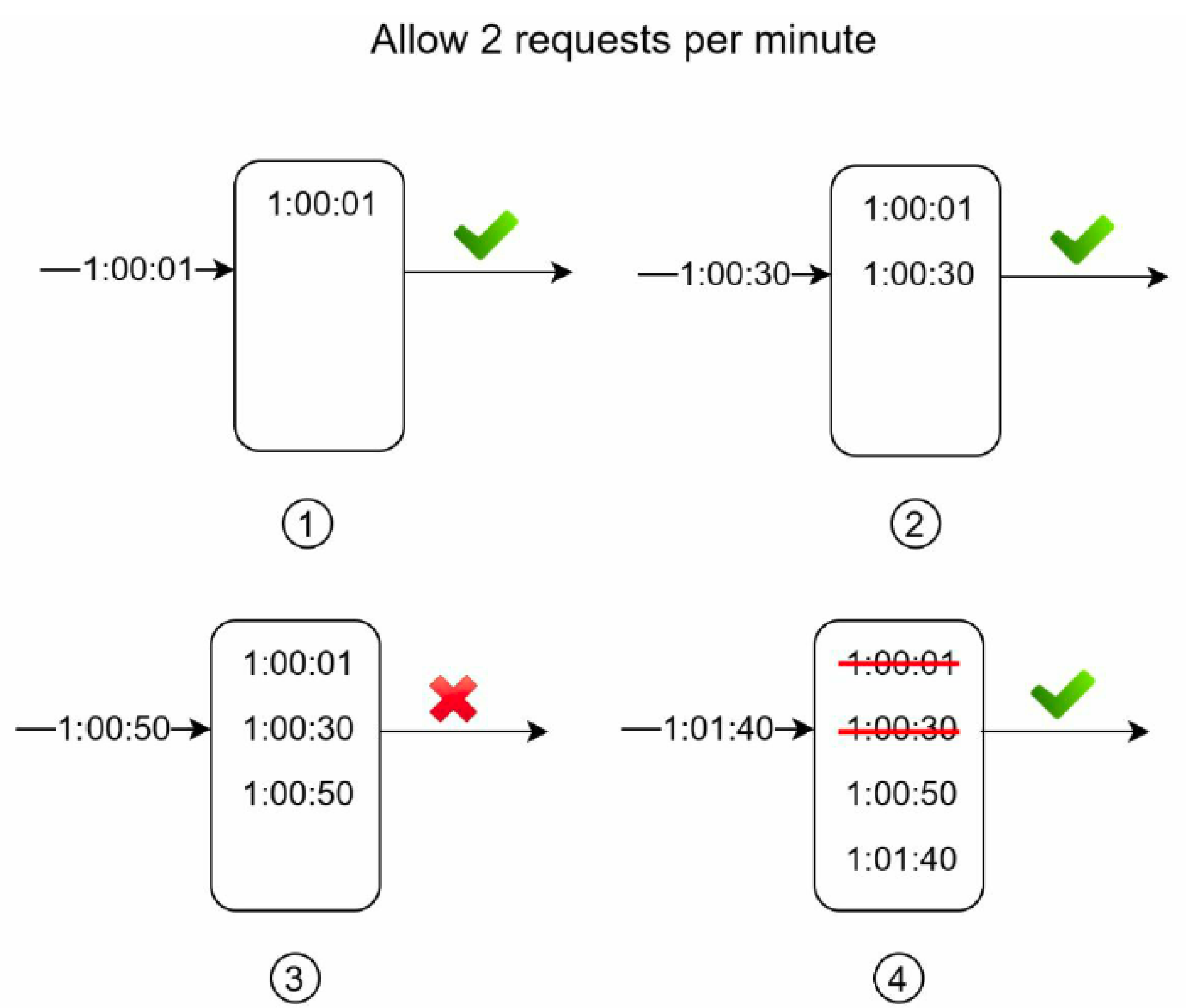
  Figure 4-9

  In Figure 4-9, the system allows a maximum of 5 requests per minute, and the available quota resets at the human-friendly round minute. As seen, there are five requests between 2:00:00 and 2:01:00 and five more requests between 2:01:00 and 2:02:00. For the one-minute window between 2:00:30 and 2:01:30, 10 requests go through. That is twice as many as 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

Sliding window log

- Fix the issue of `Fixed window counter` of allowing more requests to go through at the edges of a window.
- Keeps track of request timestamps. Timestamp data is usually kept in cache, such as sorted sets of Redis
- When a new request comes in -> remove all the outdated timestamp. Outdated timestamp -> those older than the start of the 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, rejected
- 

- 1:01:50 ->log size becomes 3 -> reject 1:01:50

- 1:02:10 [1:01:10, 1:02:10) -> log: 1:01:40, 1:01:50, 1:02:10, log size becomes 3 -> reject 1:02:10
- Pros
  - accurate. In any rolling window, requests will not exceed the rate limit
- Cons
  - Consumes a lot of memory because even if a request is rejected, its timestamp might still be stored in memory

Sliding window counter

- hybrid approach that combines the fixed window counter and sliding window log

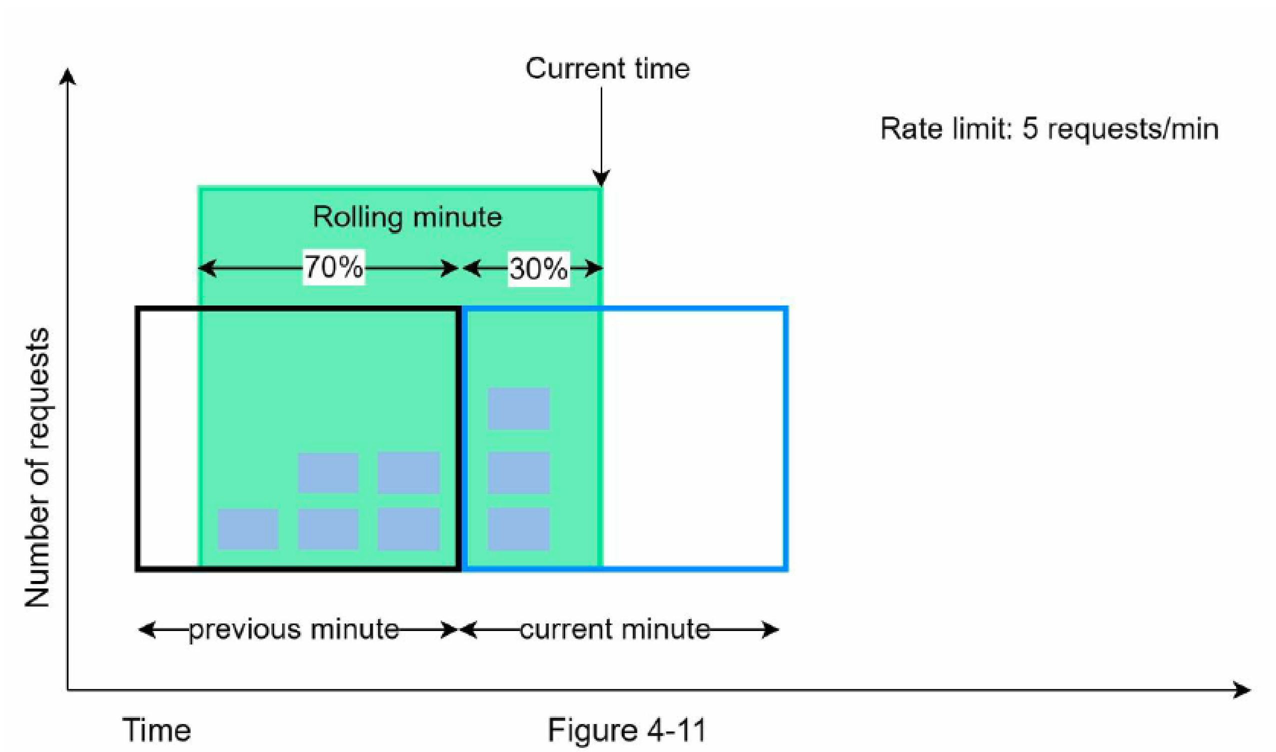


Figure 4-11

- Assume the 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 arrives at a 30% position in the current minute, the number of requests in the rolling window is calculated using the following formula:

  • Requests in current window + requests in the previous window * overlap percentage of the rolling window and previous window

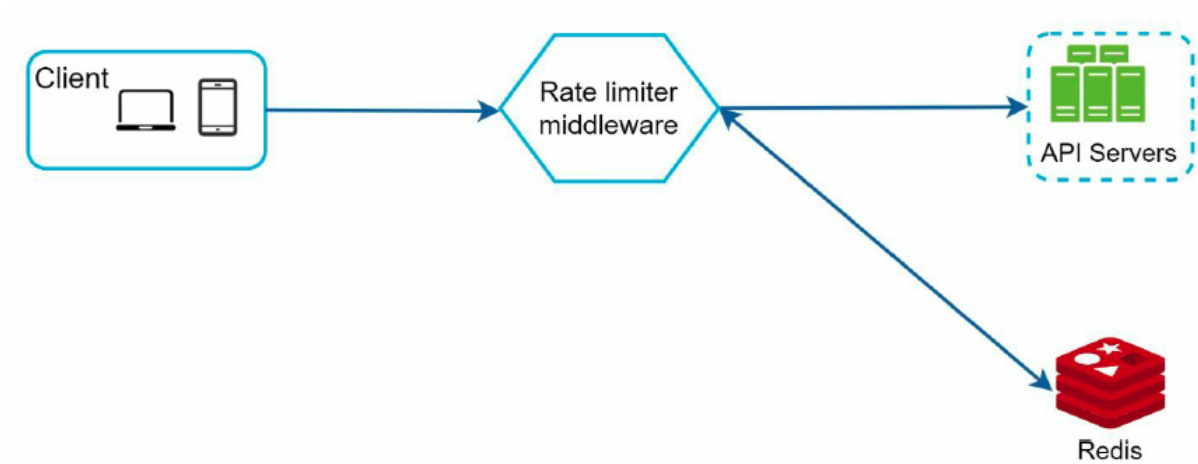
- Using this formula, we get 3 + 5 * 0.7% = 6.5 request. Depending on the use case, the number can either be rounded up or down. In our example, it is rounded down to 6.
- Pros
  - Smooths out spikes in traffic because the rate is based on the average rate of the previous window
  - Memory efficient

- Cons
    - 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

High-level architecture

- a counter to keep track of no. of requests are sent from the same user, IP address, etc.

Where shall we store counters?

- In-memory cache
    - Fast
    - Supports time-based expiration strategy
    - eg. Redis -> in-memory store that offers two commands: INCR, ENPIRE
        - INCR: 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 -> Reject
        - If the limit is not reached, the request is sent to API service. Meanwhile, the system increments the counter and saves it back to Redis.

Step 3 - Design deep dive

Questions to be answered

- How are rate limiting rules created? Where are the rules stored?
- How to handle request that are rate limited?

Rate limiting rules

Lyft open-sourced their rate-limiting component [12]. We will peek inside of the component and look at some examples of rate limiting rules:

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

In the above example, the system is configured to allow a maximum of 5 marketing messages per day. Here is another 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

- HTTP response code 429 (too many request)
- Rate limiter headers
  - 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.
  - eg. When a user has sent too many requests, a 429 too many requests error and X-RatelimitRetry-After header are returned to the client.

Detailed design

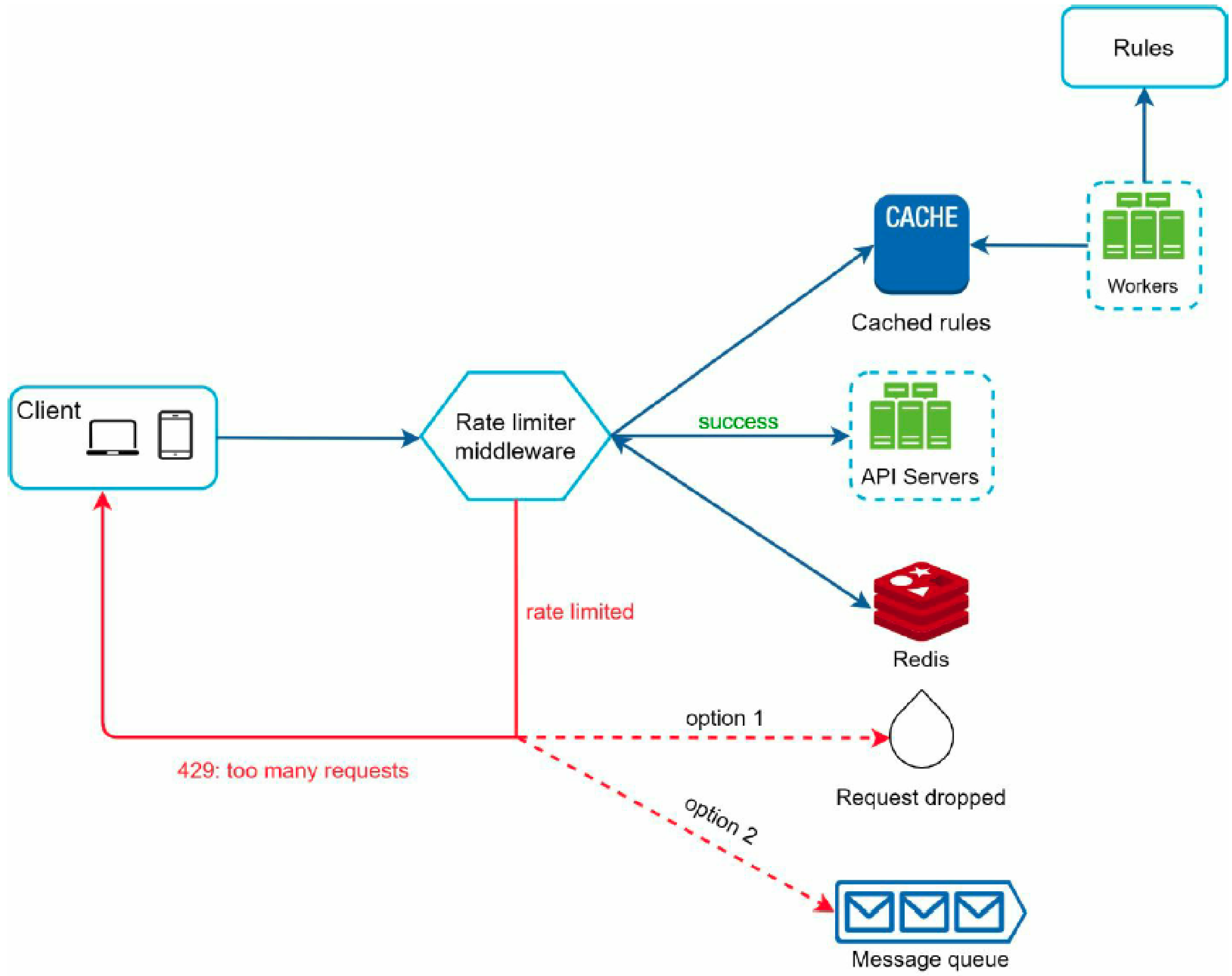


Figure 4-13

• Rules are stored on the disk. Workers frequently pull rules from the disk and store them in the 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 rate limiter 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, the request is either dropped or forwarded to the queue.

Rate limiter in a distributed environment

Challenges

- Race condition
- Synchronization issue

Race condition

- Read the counter value from Redis
- Check if (counter + 1) exceeds the threshold
- If not, increment the counter value by 1 in Redis

  Race conditions can happen in a highly concurrent environment as shown in Figure 4-14.

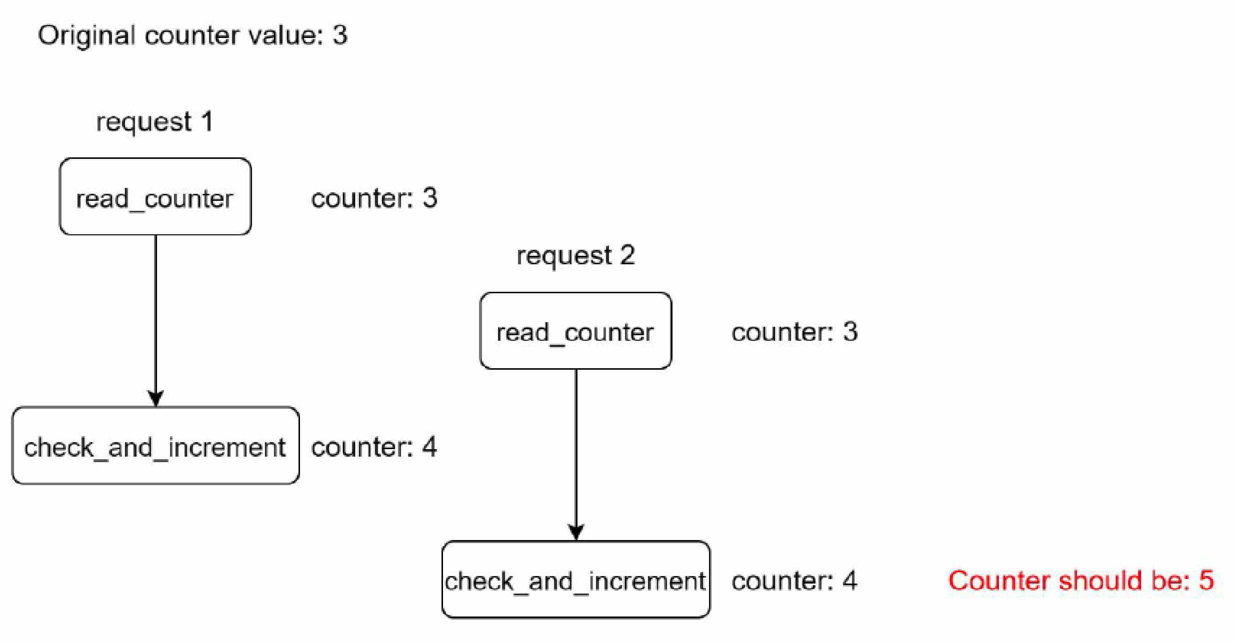
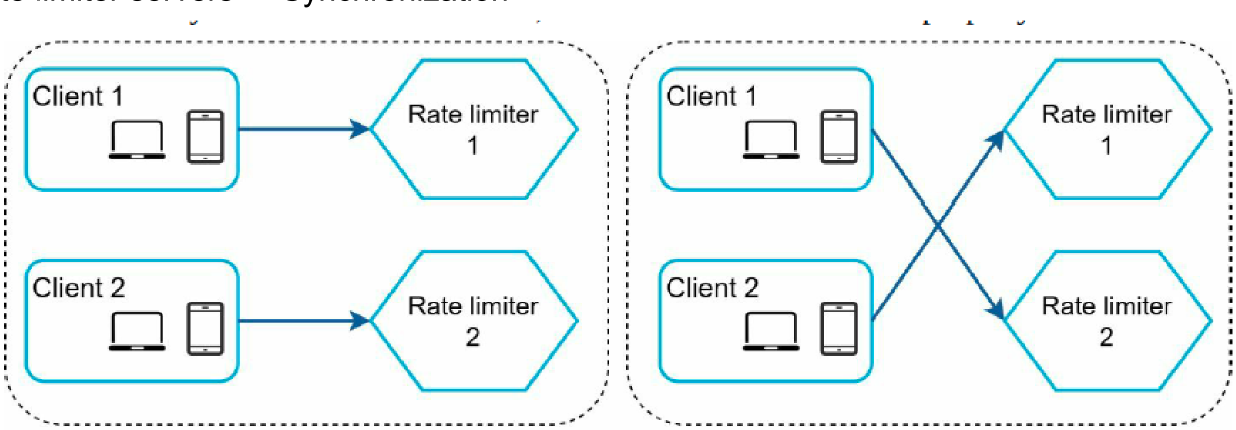


Figure 4-14

-
- Solution for solving race condition
  - Locks -> slow down system
  - Lua script
  - Sorted sets data structure in Redis

Synchronization issue

- To support millions of users, one rate limiter is not enough to handle the traffic -> multiple rate limiter servers -> Synchronization



-

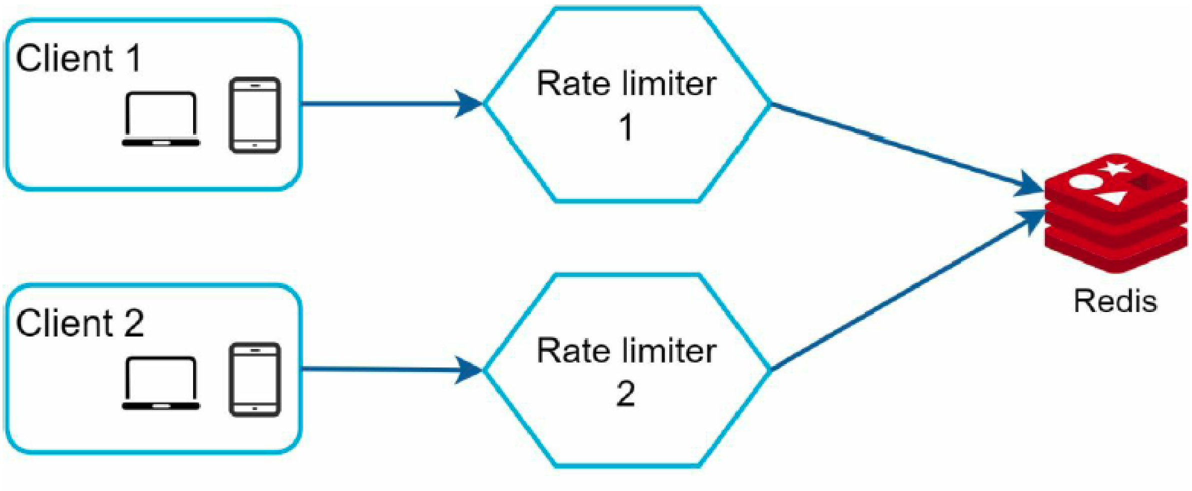


Figure 4-16

Performance optimization

- multi-data center -> Traffic is automatically routed to the closest edge server to reduce latency
- Synchronize data with an `eventual consistency model` (refer to Chapter 6: Design a Key-value store)
  - It allows for **temporary inconsistencies** across distributed data stores with the guarantee that, if no new updates are made to the system, eventually all accesses to a given data item will return the last updated value. This model is often used in distributed systems where **high availability** and partition tolerance are prioritized over strict consistency.

Monitoring

- The rate limiting algorithm is effective
- The rate limiting rules are effective

Step 4 - Wrap up

- Algorithms and pros/cons
- system architecture
- rate limiter in a distributed system
- performance optimization
- monitoring
- Optional for interview
  - hard vs soft rate limiting
    - Hard: The number of requests cannot exceed the threshold

        - Soft: Requests can exceed the threshold for a short period
    - Best practice
        - Use client cache to avoid 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 -> allow client to recover from exceptions
        - Add sufficient back off time to retry logic

Open System interconnection model (OSI model) has 7 layers

- Layer 1: Physical layer
- Layer 2: Data link layer
- Layer 3: Network layer
- Layer 4: Transport layer
- Layer 5: Session layer
- Layer 6: Presentation layer
- **Layer 7: Application layer** (we talked about above)