Functional Requirements:

- Limit the number of requests to an API within a time window, e.g., 20 requests per second.
- The APIs are accessible through a cluster, so the rate limit should be set up across different servers. The user should get an error message whenever the defined threshold is crossed within a single server or across a combination of servers.

Non-Functional Requirements:

- The system should be highly available.

- There should have miniscule or no delay in latency in Rate Limit since it might expose it to DOD attacks

Capacity Estimation and Constraints

Let calculate estimation for 1 API Rate Limiter

Total Request per seconds = 20

Total Request in 1 hour = 20 * 60 = 1200

Total Requests inn 1 day = 1200 * 24 = 28800

Total Request in 1 year = 28800 * 365 = 10512000

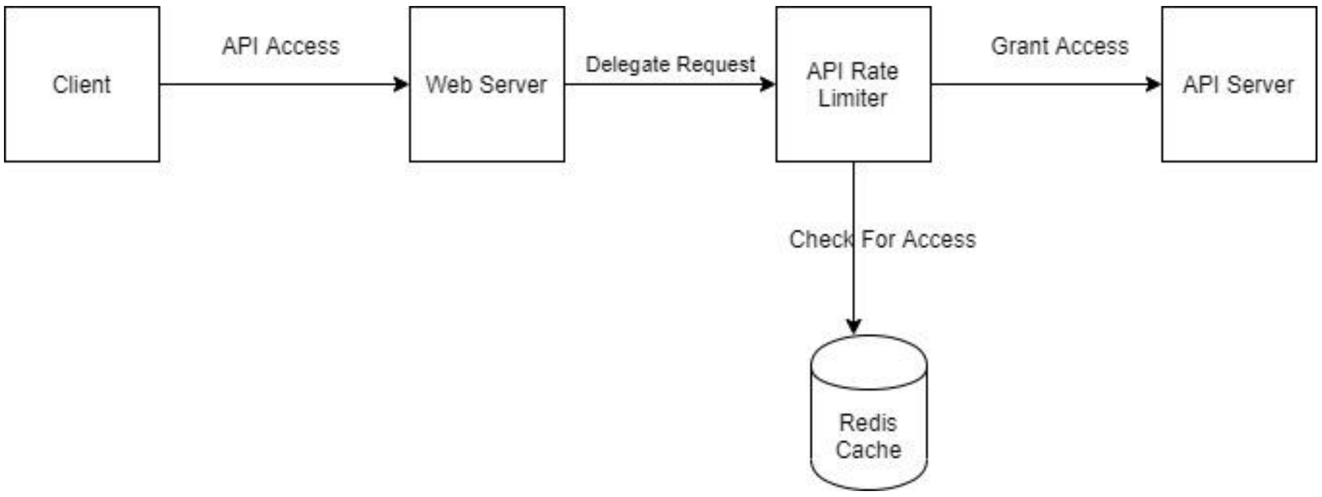
Total Request in 5 years = 10512000 * 5 = 52560000

Assuming we have 20 API Limit Servers = 52560000 * 20 = 1051200000 requests will be servers

Assuming 1 Request is 2 KiB in size

Total Storage Required = 1051200000 * 2000 = 2.12 TB

High Level Design



System APIs

Rate limiter (API Dev Key , IP address ,User id, API Service Endpoint)

Database Design

Rate Limiter

Field	Type
User_id	Integer
Ip address	Integer
Total count	Integer
API last access time	Integer

Since our system requires minimum latency, we propose to utilize Redis/Mongo Combo Redis for In-memory database and Mongo DB for permanent storage for our API rate limiter

Basic System Design and Algorithm

Throttling is a process that is used to control the usage of **APIs** by consumers during a given period. When a throttle limit is crossed, the server returns HTTP status “429 Too many requests".

Hard Throttling: The number of API requests cannot exceed the throttle limit.

Soft Throttling: In this type, we can set the API request limit to exceed a certain percentage. Let say we are serving 1000 request per seconds , an extra 20% of 1000 i.e.. 200 + 1000 =1200 requests can be served

Elastic or Dynamic Throttling: Under Elastic throttling, the number of requests can go beyond the threshold if the system has some resources

Rate Limiting Algorithms

Leaky Bucket

Leaky bucket (closely related to token bucket) is an algorithm that provides a simple, intuitive approach to rate limiting via a queue which you can think of as a bucket holding the requests. When a request is registered, it is appended to the end of the queue. At a regular interval, the first item on the queue is processed. This is also known as a first in first out (FIFO) queue. If the queue is full, then additional requests are discarded (or leaked). The advantage of this algorithm is that it smooths out bursts of requests and processes them at an approximately average rate. It is also easy to implement on a single server or load balancer and is memory efficient for each user given the limited queue size.
However, a burst of traffic can fill up the queue with old requests and starve more recent requests from being processed. It also provides no guarantee that requests get processed in a fixed amount of time. Additionally, if you load balance servers for fault tolerance or increased throughput, you must use a policy to coordinate and enforce the limit between them. We will come back to challenges of distributed environments later.

Fixed Window

In a **fixed window** algorithm, a window size of n seconds (typically using human-friendly values, such as 60 or 3600 seconds) is used to track the rate. Each incoming request increments the counter for the window. If the counter exceeds a threshold, the request is discarded. The windows are typically defined by the floor of the current timestamp, so 12:00:03 with a 60 second window length, would be in the 12:00:00 window. The advantage of this algorithm is that it ensures more recent requests gets processed without being starved by old requests. However, a single burst of traffic that occurs near the boundary of a window can result in twice the rate of requests being processed, because it will allow requests for both the current and next windows within a short time. Additionally, if many consumers wait for a reset window, for example at the top of the hour, then they may stampede your API at the same time.

Sliding Log

Sliding Log rate limiting involves tracking a time stamped log for each consumer's request. These logs are usually stored in a hash set or table that is sorted by time. Logs with timestamps beyond a threshold are discarded. When a new request comes in, we calculate the sum of logs to determine the request rate. If the request would exceed the threshold rate, then it is held. The advantage of this algorithm is that it does not suffer from the boundary conditions of fixed windows. The rate limit will be enforced precisely. Also, because the sliding log is tracked for each consumer, you do not have the stampede effect that challenges fixed windows. However, it can be very expensive to store an unlimited number of logs for every request. It is also expensive to compute because each request requires calculating a summation over the consumer's prior requests, potentially across a cluster of servers. As a result, it does not scale well to handle large bursts of traffic or denial of service attacks.

Sliding Window

This is a hybrid approach that combines the low processing cost of the fixed window algorithm, and the improved boundary conditions of the sliding log. Like the fixed window algorithm, we track a counter for each fixed window. Next, we account for a weighted value of the previous window's request rate based on the current timestamp to smooth out bursts of traffic. For example, if the current window is 25% through, then we weight the previous window's count by 75%. The relatively small number of data points needed to track per key allows us to scale and distribute across large clusters. We recommend the **sliding window** approach because it gives the flexibility to scale rate limiting with good performance. The rate windows are an intuitive way she to present rate limit data to API consumers. It also avoids the starvation problem of leaky bucket, and the bursting problems of fixed window implementations.

Data Partitioning and Replication

We can shard based on the 'UserID' to distribute the user's data.

If our APIs are partitioned, a practical consideration could be to have a separate smaller rate limiter for each API shard as well.

Caching

Since we need to frequently access the information for API Rate Limiter. We can use in memory cache such as Redis to store USER/IP address information .This information will be of immense importance for limiting the API access by users. Our rate limiter can significantly benefit from the Write-back cache by updating all counters and timestamps in cache only. The write to the permanent storage can be done at fixed intervals. This way we can ensure minimum latency added to the user's requests by the rate limiter.

Load Balancing

We can put load balancers between web servers and rate limiter server to distribute the network traffic. We can use different load balancing algos to distribute the traffic. Some of techniques are explained below :

Weighted Round Robin

Weighted Round Robin builds on the simple Round Robin load balancing method. In the weighted version, each server in the pool is given a static numerical weighting. Servers with higher ratings get more requests sent to them.

Agent-Based Adaptive Load Balancing

Each server in the pool has an agent that reports on its current load to the load balancer. This real-time information is used when deciding which server is best placed to handle a request. This is used in conjunction with other techniques such as Weighted Round Robin and Weighted Least Connection

Handling Edge cases in given system

Rate Limiting

IP: In this scheme, we throttle requests per-IP, although it is not optimal in terms of

differentiating between authentic and malicious users.

Following are problem associated with IP throttling

- No differentiation of IP Addresses. Also, Single shareable public IP can be used for malicious users to flood the Rate Limiter.
- During caching IP-based limits, as there are a huge number of IPv6 addresses available to a hacker from even one computer, it is trivial to make a server run out of memory tracking IPv6 addresses!

User: Rate limiting can be done on APIs after user authentication. Once authenticated, the user will be provided with a token which the user will pass with each request. This will ensure that we will rate limit of API with a valid authentication token. However, utilizing single token for every request will expose API to security threats by malicious users. So periodic exchange of auth token between sever and user can help overcome security threats. These auth token should be ephemeral in nature. Let say 100-200 second be validity of one auth token

Hybrid: A right approach could be to do both per-IP and per-user rate limiting, as they both have weaknesses when implemented alone, though, this will result in more cache entries with more details per entry, hence requiring more memory and storage.