Lecture 5 : API Rate Limiting & Create Your Own Rate Limiter

#What is API Rate Limiting?

- Rate Limiting ek technique hai joh control karti hai ki ek client ek specific time frame mein kitni requests server ko bhej sakta hai.

- Jaise highway par limited cars hi pass ho sakti hain, waise hi server par bhi limited requests allow hoti hai — to avoid overload

#Why is Rate Limiting important ?

1. Protection from Attacks
   - DDoS/DoS attacks se server down ho sakta hai.
   - Rate Limiting acts as a shield, to protect the system

2. Managing Cost
   - Har request mein lagta hai CPU, Memory, Bandwidth
   - Example - Zomato, Swiggy, Google Maps = Millions of API Calls
   - Each request is expensive

#Definition of Rate Limiting

"Ek client ek time frome mein kitni requests bhej sakta hai ?"

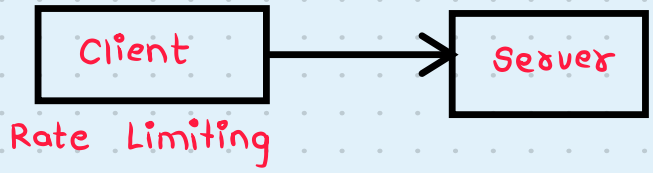
- iss limit ko enforce karta hai server yaa application - to ensure stability and efficiency

#Ways to implement Rate Limiting

- There are three ways to implement Rate Limiting
  1. Client-side Rate Limiting
  2. Server-side Rate Limiting
  3. Middleware -side Rate Limiting

1. Client - side Rate Limiting

- Client ko hi rate limits bataye jaate hain.
- Client app / browser requests ko khud regulate karta hai.

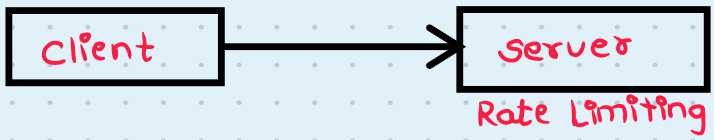


Problems

1. Yeh method unreliable hai
2. Can easily be bypass coz its added in client code & can be changed by inspect in browser.
3. Server phir bhi vulnerable ho sakta hai

2. Server - side Rate Limiting

- Server khud har client ka rate track karta hai
- Requests aane se pehle check hota hai :
  - Quota ke andaar → process
  - Exceed → Block

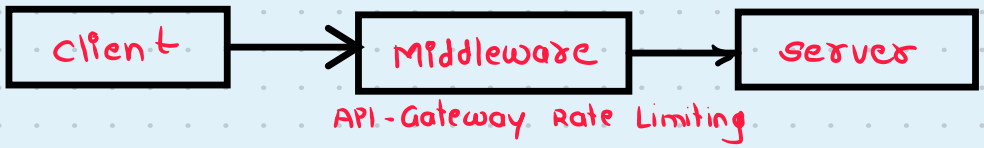


3. Middleware - based Rate Limiting

- Ek intermediary layer hoti hai

  client → middleware → Server

- Middleware har request ko :
  1. Intercept karta hai
  2. Rate limiting logic apply karta hai
  3. Valid requests ko forward karta hai



Benefits

1. Centralized Logic (server se alag)
2. Easily Scalable
3. Extra layer of security before server

Which one should we implement : server-side or middleware ?

Yeh aapke Application Use Case par depend karta hai -

1. Server-side implementation

- Aagar apki server-side language (Java/C++/JS/Python) hai → which is fast, toh its better to do server-side par rate limiting implement kare.

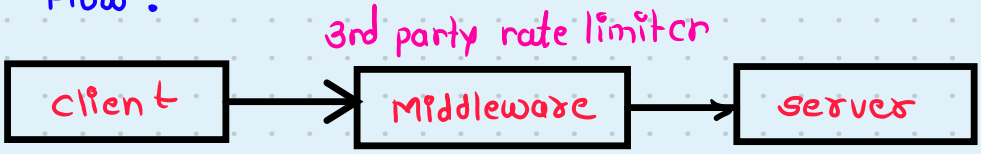
- Flow : Client ⟶ Server ⟶ Rate Limiting

2. Middleware Implementation

- Alternative way : 3rd party rate limiter providers ka use karenge like
  - ⊙ Amazon AWS API Gateway

- Yeh providers middleware layer ke upar rate limiting implement karte hai

- Flow :

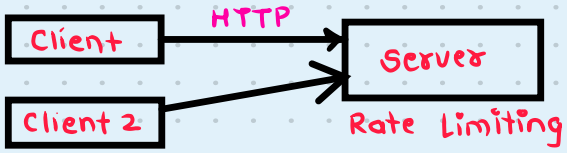


- We don't have to worry about logic - Amazon will manage itself.

Rate Limiting on which basis?

- IP-based : Client ke IP address ke basis par
- User-based : User-id ke basis par
- Other ways : Primary key, Candidate key, ke basis par

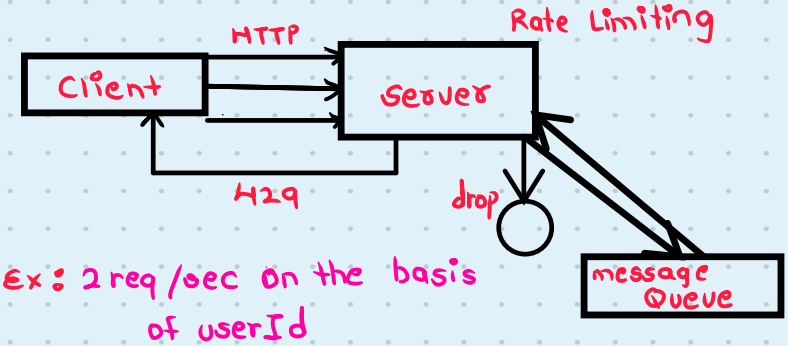
Rate Limiting Diagram



⊙ Flow

- Client → Rate Limiting → Server → Message Queue
- Server → drop (429) when limit exceeds
- Ex : 2 req/second on user ID exists

What happens during Rate Limiting?



Ex : 2 req/sec on the basis of userId

Response Code

200 : OK
429 : Too many requests

Headers

x-api-limit ⇒ 2
x-duration ⇒ seconds

Important Points

- dropping requests silently (without notifying clients) is not a good method : Not a good idea as client is unaware

Response Code in Rate Limiting

Code	Meaning
200	OK (Req. accepted)
429	Too many requests (Limit exceeded)

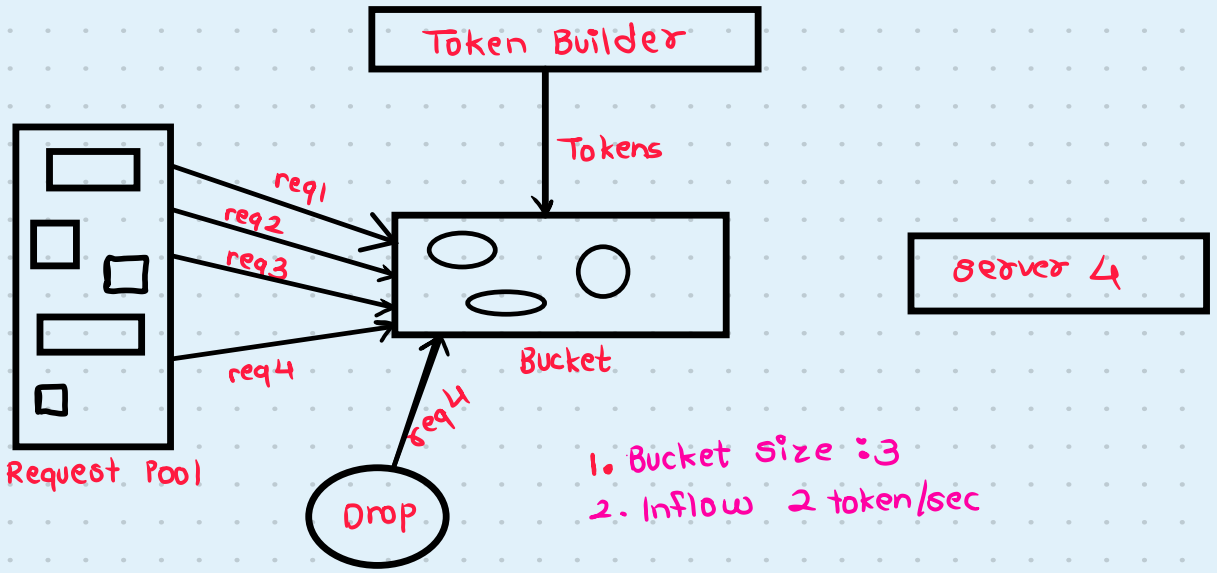
⊙ Server provider info deta hai like
↳ x-api-limit : 2 (allowed requests)
↳ x-duration : time frame in seconds

Algorithms of Rate Limiting

1. Token Bucket Algorithm

⊙ Requests (req1, req2, req3, req4) ⟶ Bucket (capacity 3 tokens) ⟶ Server

- Token Builder adds tokens to bucket at 2tokens/sec
- If Bucket full ⟶ new tokens overflow (lost)
- Requests uses 1 token to proceed
- If no token ⟶ request dropped



Pros

1. Simple to implement
2. Can handle short bursts of traffic

Cons

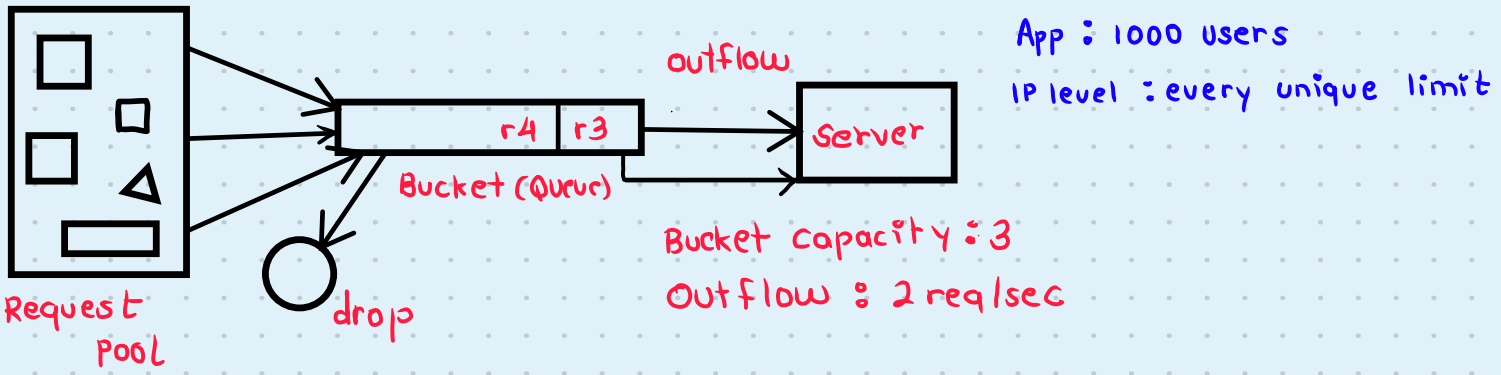
1. Requires accurate info to decide bucket size & token inflow rate

2. Leaky Bucket Algorithm (Global Rate Limiting)

- Requests (r1, r2, r3, r4) ⟶ Bucket (queue capacity 3) ⟶ Server (handles outflow 2 req/sec)
- Overflow requests dropped

Key Points

1. Bucket Capacity : 3 requests
2. Outflow rate : 2 req/sec
3. Server processes requests at fixed outflow rate
4. Middleware throttles requests so server not overwhelmed.
5. Example Use : Shopify



⊙ Pros

1. Simple to implement
2. Prevents server crash even during traffic bursts

⊙ Cons

1. During DDoS / DoS, server may still miss legitimate valuable requests
2. Requests exceeding outflow get dropped

Application Example

⊙ 100 APIs ⟶ API Gateway applies limiting ⟶ 2 bucket limit per API handles overload

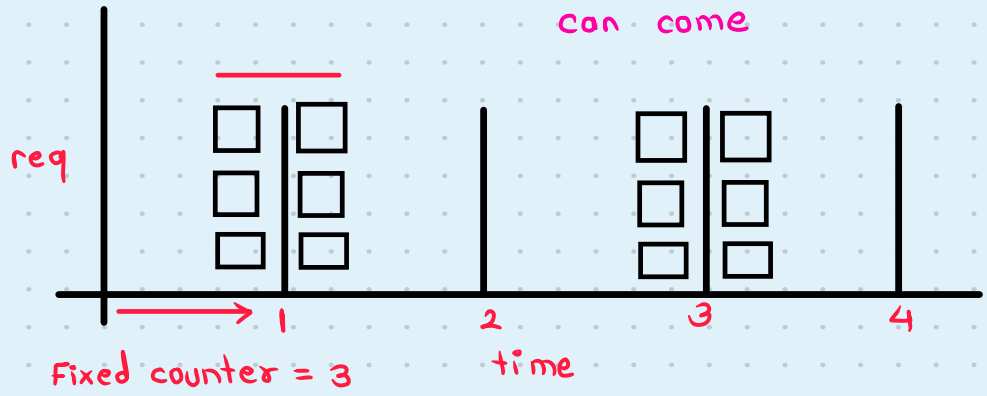
3. Fixed Window Counter

- In a fixed time interval (eg 1 second or 1 minute), only a specific number are allowed

Diagram

Time blocks 1, 2, 3, 4 with requests shown in each, block 2 has 3 requests

1. In a fixed interval (say a sec, or a minute). Only a specific no. of req can come



Details

1. Fixed Counter limit : 3 requests per interval
2. Requests exceeding the limit in the window are dropped
3. can be applied based on IP, User Id, or Client id

Cons

1. If burst traffic hits at the end of the window, it can cause server overload and high latency
2. Examples : In a 3-minute window, if 6 requests come at the last second, all will be processed together, causing potential issues

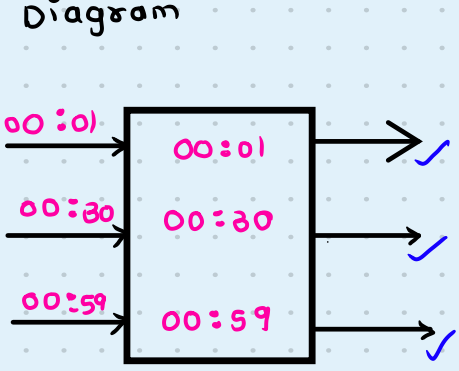
4. Sliding Window Log Algorithm

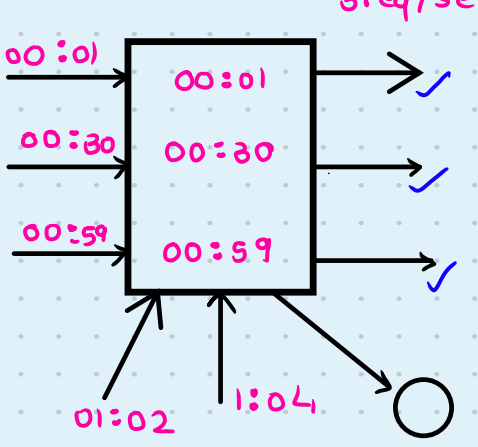
- Considered the best and strictest rate limiting algorithm.
- Tracks every request individually in a log file.

How it works?

- Stores timestamp of each request in a log
- On new request :
  1. Removes outdated requests from the log (older than window duration)
  2. Adds new request to the log
  3. Checks if request count exceeds the limit
  4. If limit exceeded ⟶ request dropped, else processed.

Diagram





Linux ⟶ EPOCH timestamp

Example

- A request arriving at 01:01 (within 1-minute window) will be dropped if limit reached.
- Sliding window counts requests in the last N minutes (eg last 3 minutes)

Pros

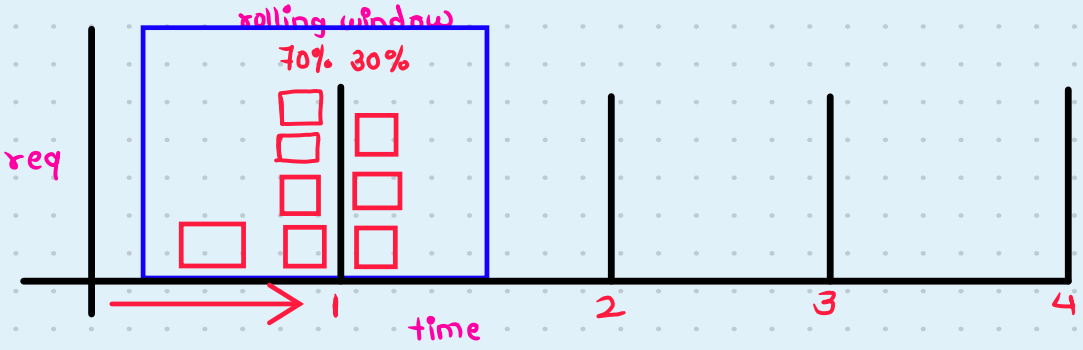
- Very strict enforcement of limits

Cons

- Slow due to frequent log updates
- Memory-intensive to store request logs

5. Sliding Window Counter Algorithm

- Hybrid approach combining Fixed Window & Sliding Window Log algorithms.
- Counts requests in the current interval plus a weighted fraction of requests from the previous interval



Formula: No. of req in curr interval + No. of req in prev interval * 70%.

$= 4 + 5 \times 0.7$

$= 7.5$

Purpose

- Provides smoother rate limiting by considering partial previous internal traffic
- Avoids sudden spikes in request count at window boundaries

Create Your Own Rate Limiter

1] Which Algorithm to implement?

- Fixed Window Counter is a simple & common choice
- Examples:

  Time Blocks: 1|2|3|4

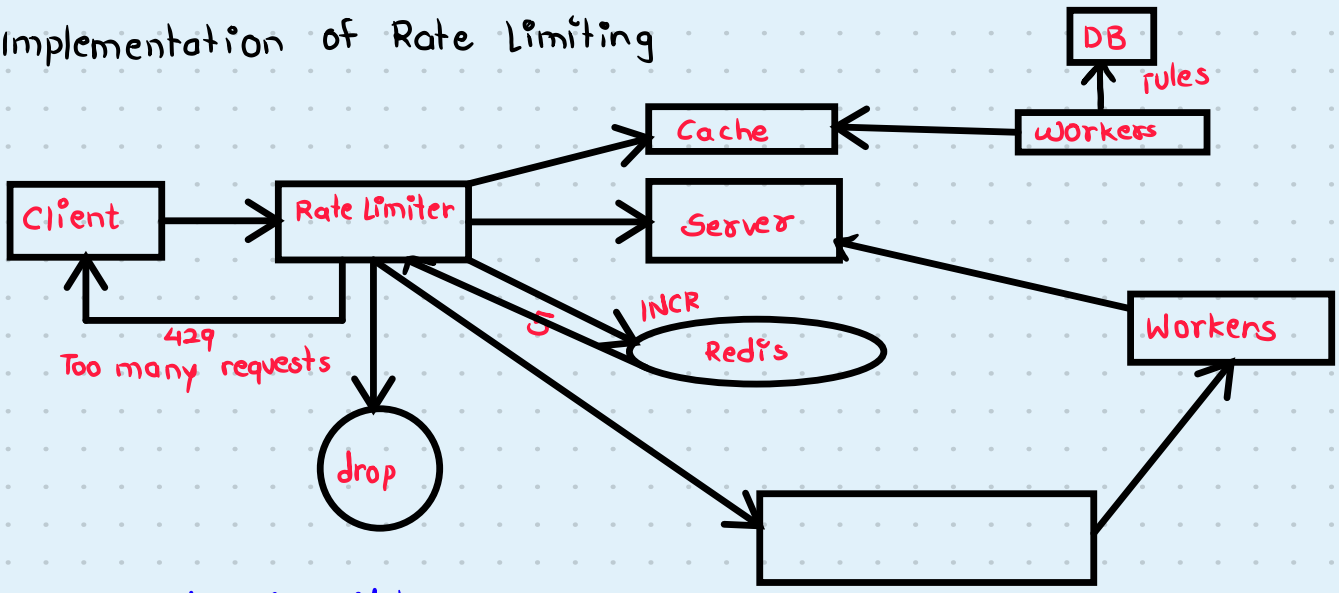
  Expected count: 3

  Current count: 3

8] Storage for Counters

- Counter Storage Options
  - Database (db) : SQL / NoSQL (slow)
  - Cache : In-memory cache like Redis (fast)

Implementation of Rate Limiting



- usually done in middleware
- middleware intercepts requests, check limits and allows or blocks accordingly

◉ System Architecture

Components

- DB - Stores rules and data (slow)
- Cache (Redis) - stores counters and rules for quick access
- Worker - Periodically syncs rules from DB to cache
- Rate Limiter - uses cached counters to enforce limits
- Messaging Queue (kafka) - Handles high priority requests & async processing

◉ Client ⟶ Rate Limiter ⟶ Server flow

◉ Redis Logic

- Counter example : If counter = 4 and limit exceeded, requests denied
- Redis key expiry : 1 minute - after which counter resets to 0
- Cache reduces DB load by storing rules & counters
- Workers regularly fetch rules from DB & update cache
- High priority requests pushed to kafka for async handling