
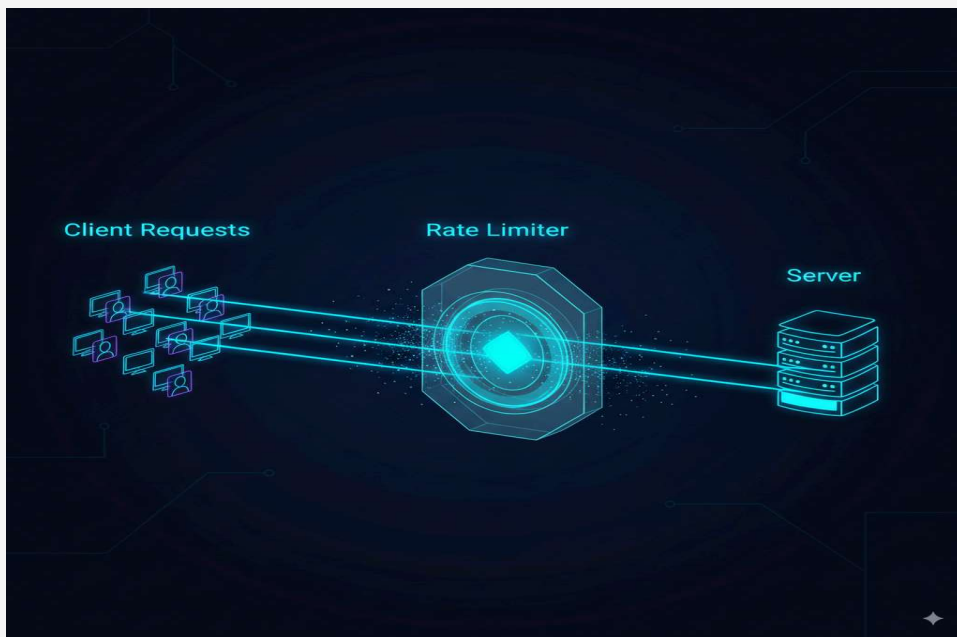


Lecture 5: API Rate Limiting & Create Your Own Rate Limiter

What is API Rate Limiting?

Rate limiting ek technique hai jo 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 hain – 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

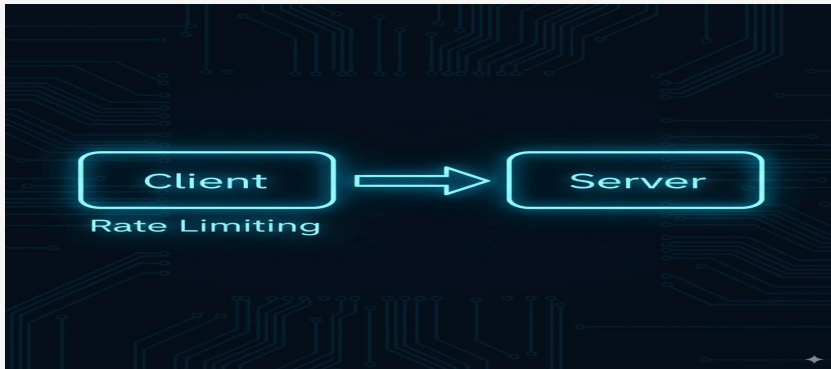
"Ek client ek time frame mein kitni requests bhej sakta hai?"

🧠 Is limit ko enforce karta hai **server** ya application — to ensure **stability and efficiency**.

Three Ways to Implement Rate Limiting

1. Client-side Rate Limiting


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




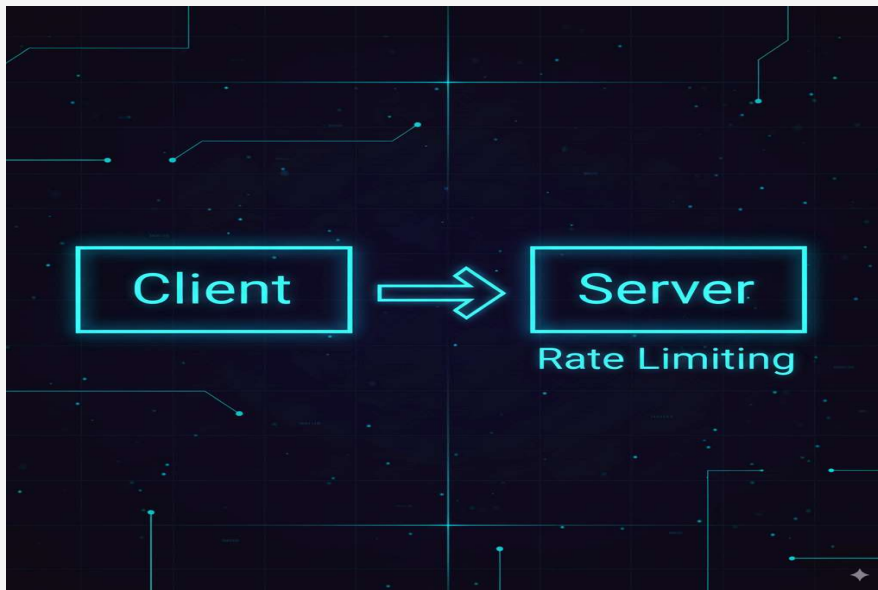
Problem:

- Yeh method **unreliable** hai ❌
- Easily **bypass** kiya ja sakta hai by attackers 🧑💻
- Server phir bhi vulnerable ho sakta hai.




2. Server-side Rate Limiting 👍

- Server khud har client ka **rate track** karta hai.
- Request aane se pehle check hota hai:
 -  Quota ke andar → Process

-  Exceed → Block



3. Middleware-based Rate Limiting

- Ek intermediary layer hoti hai:
Client  Middleware  Server
- Middleware har request ko:
 - **Intercept** karta hai
 - **Rate limiting logic apply** karta hai
 -  Valid requests ko forward karta hai



📌 Benefits:

- 🌿 **Centralized logic** (server se alag)
- ✅ **Easily scalable**
- 🛡️ **Extra layer of security** before server

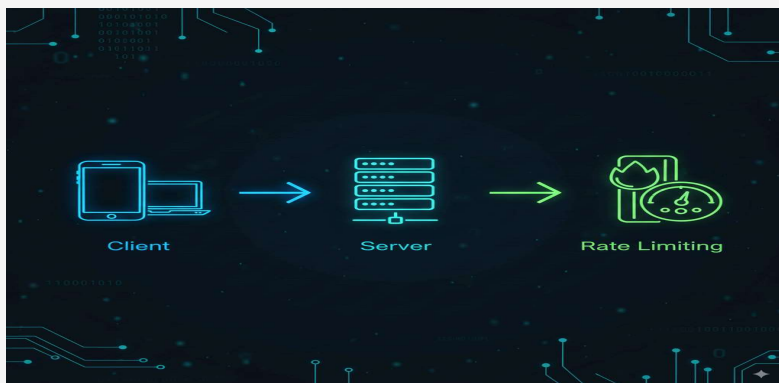
🤔 हम server-side और middleware में से कौनसा implement करें?

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

🔍 यह आपके **Application Use Case** पर depend करता है।

1 Server-side Implementation

- अगर आपकी server-side भाषा (Java, C++, JavaScript, Python) तेज़ है, तो बेहतर है कि आप **server-side** पर rate limiting implement करें।
- सरल Flow:
Client → Server → Rate Limiting



2 Middleware Implementation :

- Alternative तरीका है **3rd party Rate Limiter providers** का use करना, जैसे:
 - **Amazon AWS API Gateway** आदि
- ये providers **middleware layer** के ऊपर rate limiting implement करते हैं।

- Flow:
Client (Front End) → Middleware → Backend
- आपको इसकी चिंता करने की जरूरत नहीं होती,
Amazon खुद इसे manage करता है।



Rate limiting on what?

(किसके base पर करें?)

- **IP based:** Client के IP address के आधार पर rate limiting।
- **User-based:** User ID के आधार पर rate limiting।
- **Other ways:** Primary key, Candidate key आदि के आधार पर भी rate limiting की जा सकती है।

Rate limiting flow diagram

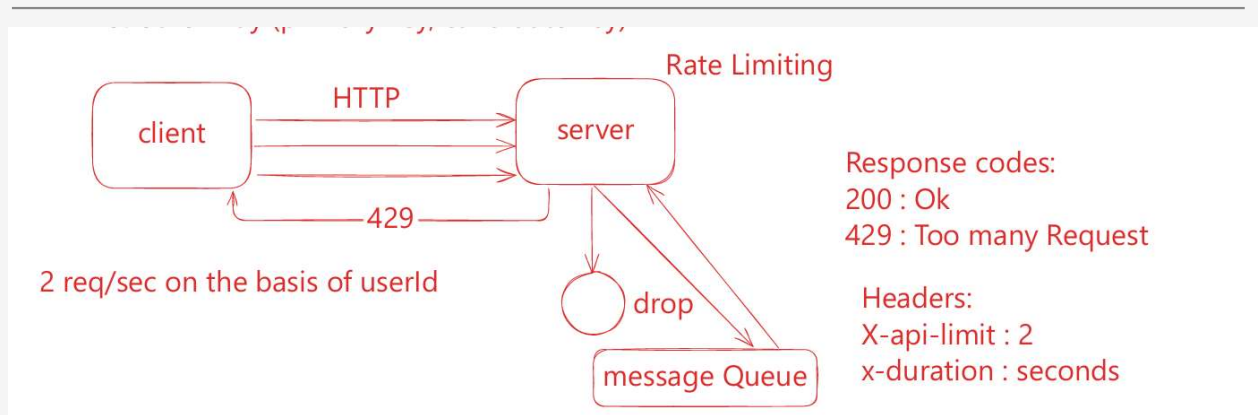
Client → Server (Rate Limiting applied here)

Rate limiting में होता क्या है?

(What happens in rate limiting?)

Flow Diagram

Client → (Rate limiting) → Server → Message Queue
 Server → Drop (429) when limit exceeds
Example: 2 requests/second on user ID basis



Important Points

- **Dropping requests silently (without notifying client) is not a good method!**
ऐसा करने से client को पता नहीं चलता, और ये धोखा जैसा लगता है।

Response Codes in Rate Limiting

Code	Meaning
200	OK (Request accepted)
429	Too Many Requests (Limit exceeded)

- Server provider info देता है जैसे:
 - **X-API-Limit:** 2 (allowed requests)
 - **X-Duration:** time frame in seconds

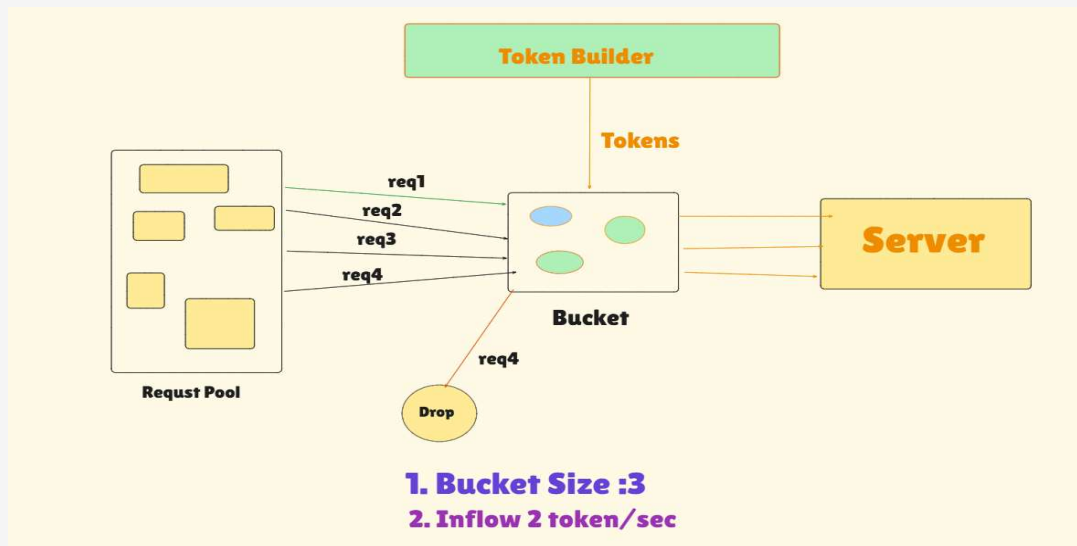
Algorithms of Rate Limiting

1 Token Bucket Algorithm

Flow Diagram

- Requests (req1, req2, req3, req4) → Bucket (capacity 3 tokens) → Server

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



👍 Pros

- Simple to implement
- Can handle short bursts of traffic

👎 Cons

- Requires accurate info to decide bucket size and token inflow rate

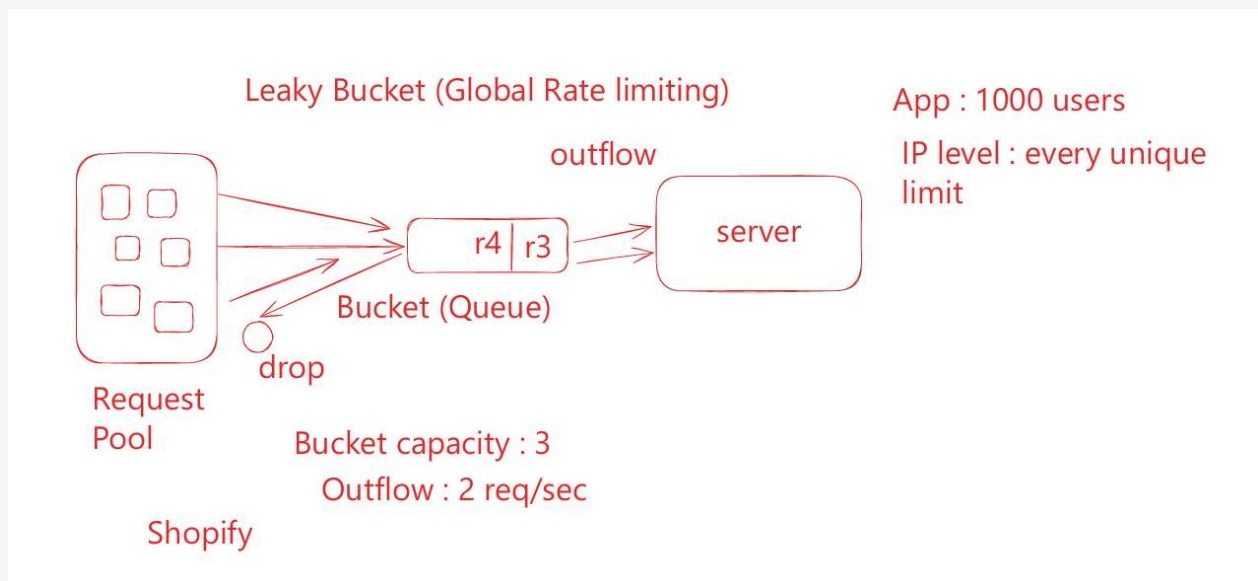
2 Leaky Bucket Algorithm (Global Rate Limiting)

🔄 Flow Diagram

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

🔑 Key Points

- **Bucket capacity:** 3 requests
- **Outflow rate:** 2 requests/sec
- Server processes requests at fixed outflow rate
- Middleware throttles requests so server not overwhelmed
- Example use: Shopify



👍 Pros

- Simple to implement
- Prevents server crash even during traffic bursts

👎 Cons

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

🔧 Application Example

- 100 APIs → API Gateway applies limiting → 2 bucket limit per API → handles overload

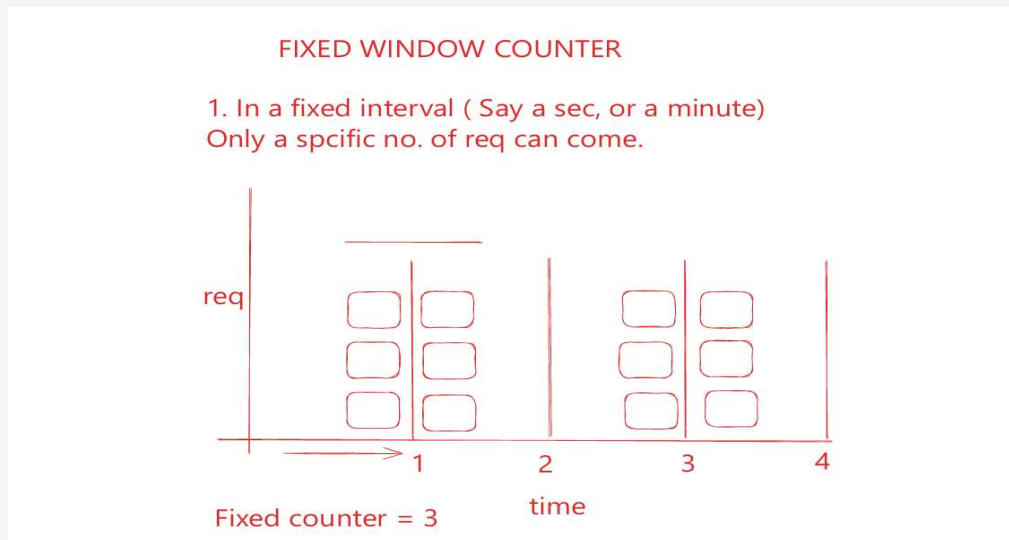
3 Fixed Window Counter ⌚

Concept

- In a fixed time interval (e.g., 1 second or 1 minute), only a specific number of requests are allowed. 🚦

Diagram

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



Details

- Fixed Counter limit: 3 requests per interval 🎯
- Requests exceeding the limit in the window are dropped ❌
- Can be applied based on IP, User ID, or Client ID 🌐

Cons

- If burst traffic hits at the end of the window, it can cause server overload and high latency ⚠️
- Example: 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 📋

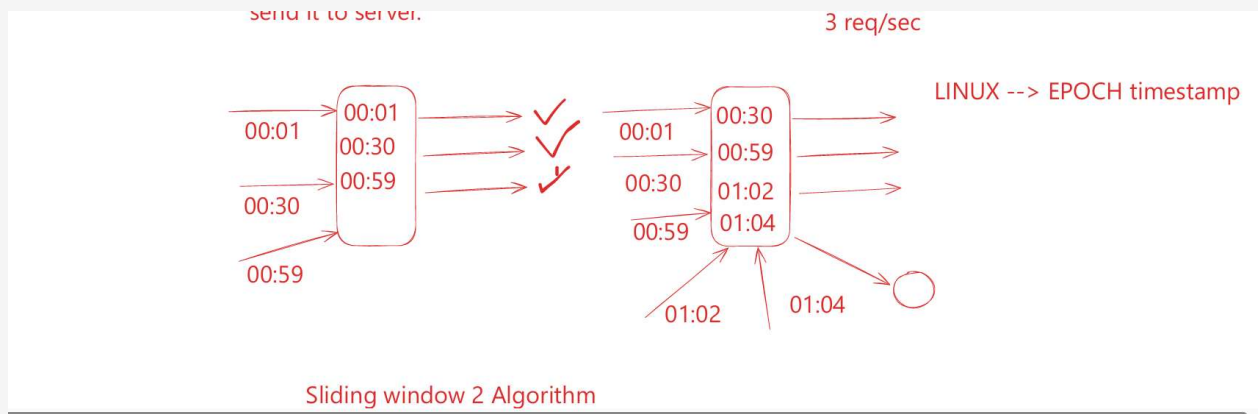
Overview

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

How it works

- Stores timestamp of each request in a log 🕒
- On new request:
 - Removes outdated requests from the log (older than window duration) 🗑️
 - Adds new request to the log ➕
 - Checks if request count exceeds the limit 🔍
 - If limit exceeded → request dropped ❌, else processed ✅

Diagram:



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** (e.g., last 3 minutes) 🔄

Pros

- Very strict enforcement of limits ✅

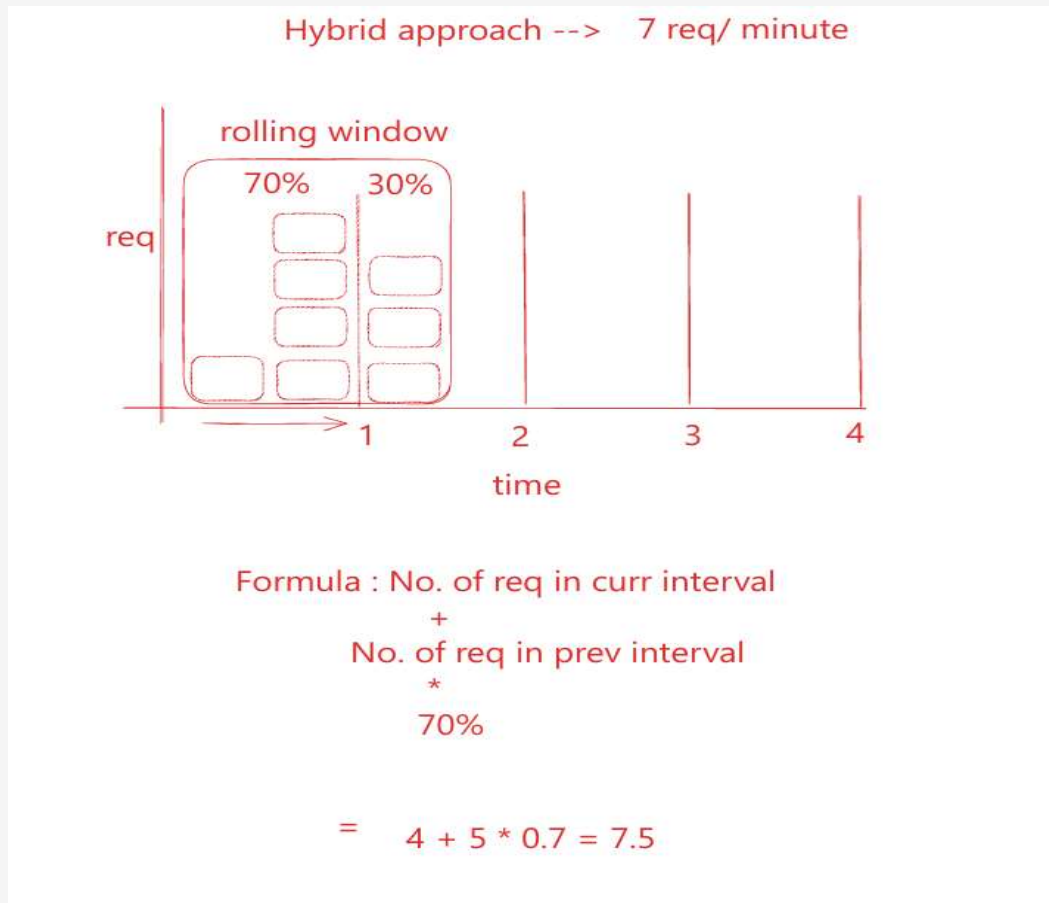
Cons

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

5 Sliding Window Counter Algorithm 📋

Overview

- Hybrid approach combining **Fixed Window** and **Sliding Window Log** algorithms 🕒
- Counts requests in the current interval **plus** a weighted fraction of requests from the previous interval 📊



Purpose

- Provides **smoother rate limiting** by considering partial previous interval traffic 🕒
- Avoids sudden spikes in request count at window boundaries 🚫

🔧 Create Our Own Rate Limiter

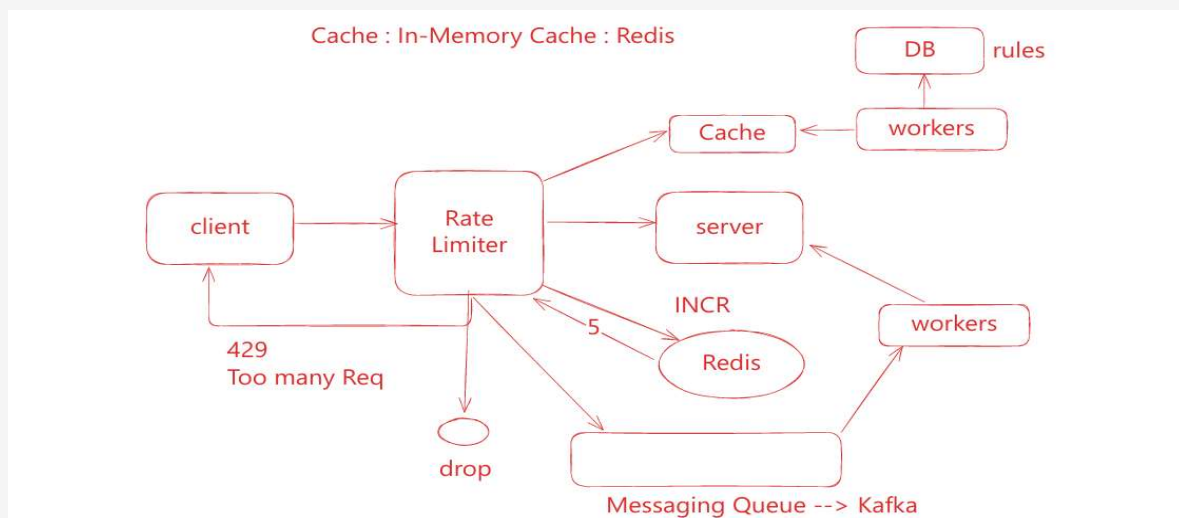
Which Algorithm to Implement? 🤔

- **Fixed Window Counter** is a simple and common choice.
 - Example:
Time blocks: 1 | 2 | 3 | 4
Expected count = 3
Current count = 3
-

Storage for Counters

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

Rate Limiting Implementation



- Usually done in **middleware** layer
 - Middleware intercepts requests, checks limits, and allows or blocks accordingly
-

System Architecture (Image 2 Description)

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
 - **Message Queue (Kafka):** Handles high priority requests and async processing
 - **Client** → **Rate Limiter** → **Server** flow
-

Redis Logic Explained

- 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 and counters
 - Workers regularly fetch rules from DB and update cache
 - High priority requests pushed to Kafka for async handling
-

API Rate Limiting — Quick Notes

What & Why?

- Limits requests per client/time to protect server and reduce cost.
- Prevents DDoS, controls CPU & bandwidth usage.

Implementation

- **Client-side:** Unreliable, easy to bypass.
- **Server-side:** Tracks & blocks excess requests.
- **Middleware:** Intercepts requests before server, scalable & secure.

Rate Limit Based On

- IP, User ID, or keys.
-

Algorithms Overview

Algorithm	Pros	Cons
Token Bucket	Handles bursts, simple	Needs tuning
Leaky Bucket	Prevents overload	Drops legit requests
Fixed Window Counter	Simple	Burst spikes
Sliding Window Log	Strict	Slow, memory-heavy
Sliding Window Counter	Smooth limiting	More complex

Build Your Rate Limiter

- Use **Fixed Window Counter** + **Redis cache** (fast counters with expiry).
 - Workers sync rules DB → Redis.
 - Kafka handles priority tasks asynchronously.
 - Return **429 Too Many Requests** on limit exceed.
-

Key Takeaway

Rate limiting = protect + stabilize server by controlling request flow, choosing algorithm & infrastructure wisely.