Rate Limites

* Provides a limited no. trials in a particular amount of time. (requests).

* Used to improve Security

  ↳ Like else one can brute force our phone to get the password.
  ↳ or successfully execute a DDOS attack.
  ↳ or excessively increase the load on our cloud servers which may eventually lead to our servers getting crashed.

* Example of rate limiter on youtube

  ↳ Upload
    ↳ 20 videos / 24 hrs ⇒ to prevent excessive load on storage and network.
    ↳ 100 comments / day else the U.X. will hamper by the bots spamming over the popular videos.

* Majorly used to prevent Backend API's.

* If we were req. to implement rate limiter inside only one micro-service, then that would be the easiest task.

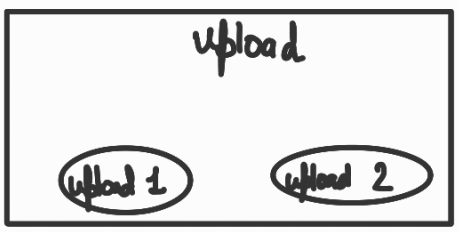
Basically, then we just have added more code to that.

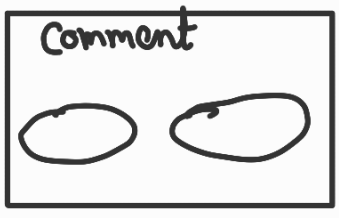
for now, assume that we are implementig this rate limiter for many microservices.

↳ i.e There are going to multiple services.

Rate limiter

Upload Rate limiter	Comment Rate limiter.





Now, the upload 1 needs to talk to the upload 2 as they will have a common total limit of 20 videos / 24 hr. Therefore we require the rate limiter to be global / common.

If (limit < rate · limit) ⇒ Hit

else reject with 429. error.

It is generally ideal to implement the rate limiter at the client-side.

↳ Problem

↳ User might reverse-engineer the API or the application and pass up the limit.

Therefore Rate limiter should be applied in both client and server side.

Question :- How do we going to identify the source?

↳ Identifying the origin of the request is really imp. as this will help to decide wether to block the request or not?

↳ Now, should we use

↳ UserID

↳ IP address

Non-functional req.nt

Storage

- An IP address has 4 bytes
- No. of req over a time period takes 128 bytes (Conservative)
- No. of users = 1 Billion

∴ Total Size = 1 Billion x 132 bytes

= 132 GB.

(can be easily stored in the memory.

Availability

- Ideally should be max. i.e. 5 9's.
- But what should happen if our rate limiter goes down?

Fail - open

↳ The system should work as if there was no rate limiter.

↳ Genuine users :) ✓

Malicious users :- 

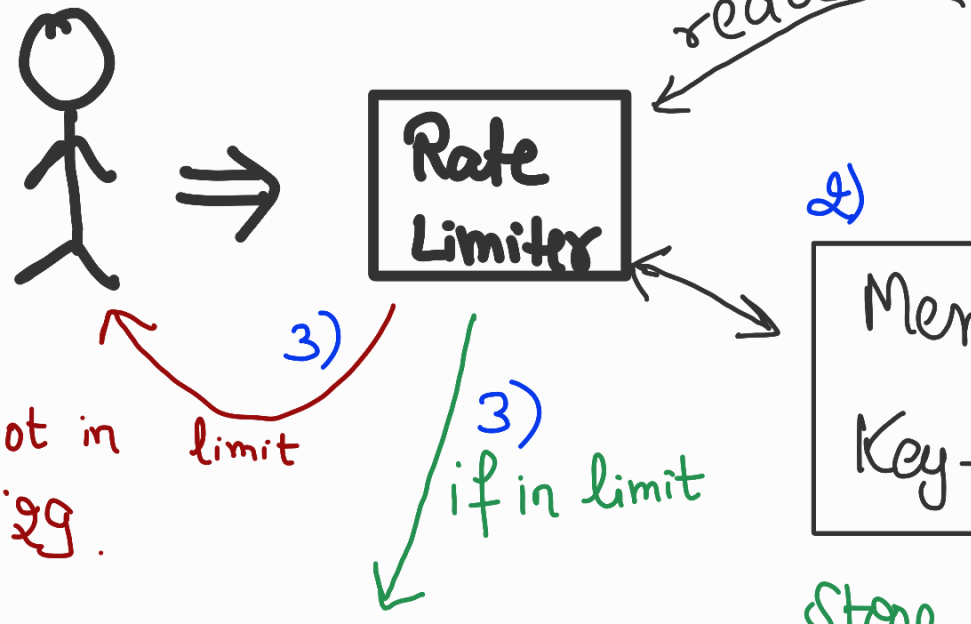
Fail - closed

↳ The system should collapse.

↳ Genuine users ⇒ ✗

Malicious users ⇒ 

Proposing the high level design.



→ our rules

→ should be persistent

Store UID vs no. of requests.

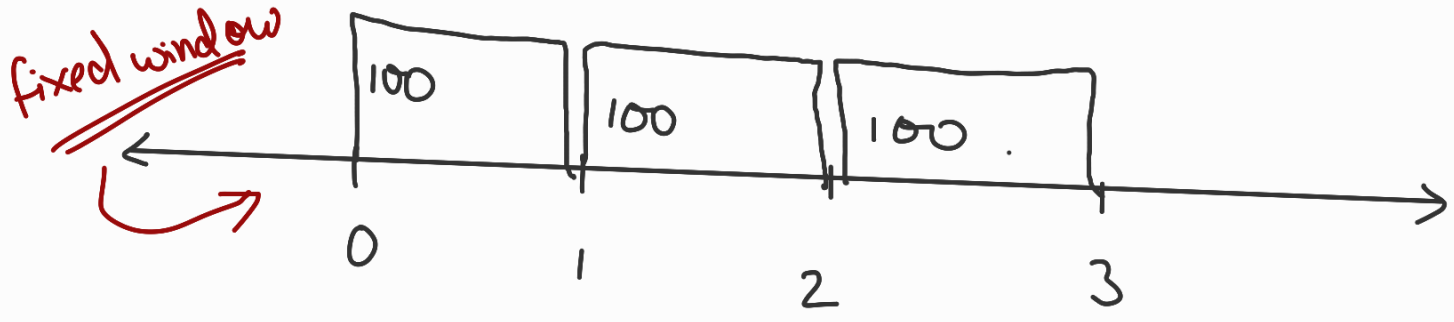
Redis or Mem Cache.

Choosen memory

↳ Because writes to be done & memory is much faster than storage

But which algorithm we might be going to use?

↳ Suppose we allow 100 req./min. like



Now, suppose the client send

100 req at 12:00:59

& 100 req at 12:01:01

So basically he sent 200 req in an interval of 2 sec. which is

Okay since he have to wait 59 sec

but not okay as he got 200 req./sec.

* Benifit
  - ↳ Simple
  - ↳ Do not have to track when each req is

made.

* Using Sliding window:

↳ More accurate

↳ Have to store every req. time stamp i.e. 32 bits per requests.

↳ Can delete data older than 60 seconds.

↳ Can use

↳ Redis's Sorted Sets.

↳ Token Bucket

↳ Sliding window Counter.

* Data Schemas for the in-memory store.

Key → count.

& Rate Limiter will enforce the rules.

on every request counter will ++.

& set expiry date of the data in redis using expire.

* For rules.

id : string

API : string

Operation : upload / Download / comment

time-unit : string

requests : Integers.

* In the memory, we will use

Consistent hashing & sharding.

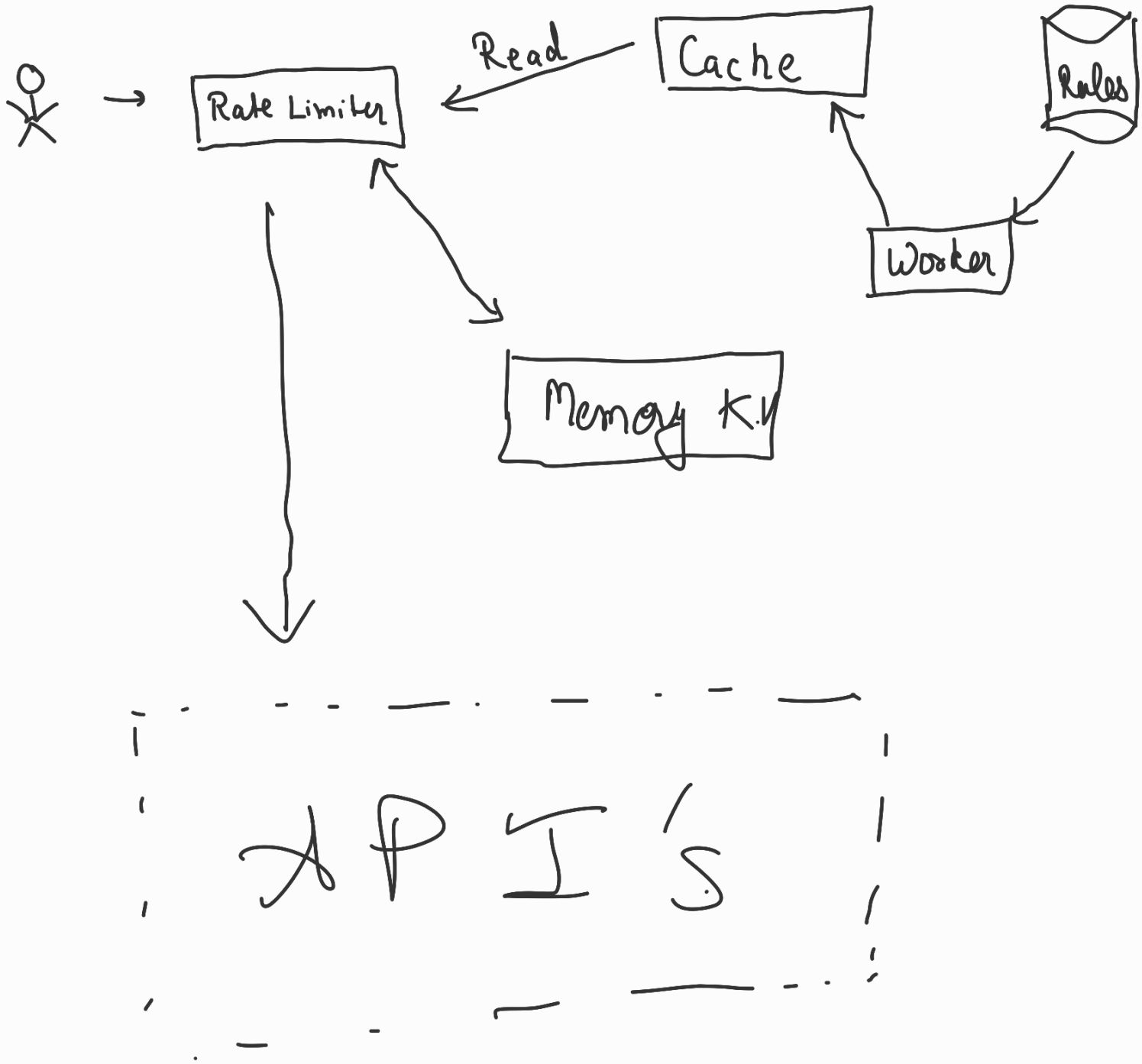
* To improve the latency :-

  ↳ Can't do much about read/writes from the storage.

  ↳ But the D.B.'s read are quite slow.

  So, we will add cache in b/w the Rules Database and Rate limiter. and periodically update cache from the rules.

  ↳ This will make the rules persistent and fast to access.


Read
Cache
Rules
Rate Limiter
Worker
Memory K.V
APIs