Design a Rate Limiter

Disclaimer: The design is far from perfect but I have tried to capture things that you can realistically cover in a 25–35 mins interview.

General Overview on Rate limiter for our understanding:

Rate Limiter: A rate limiter in HTTP world is used to limit the number of client requests allowed to be processed over a specified period.

Some real-world use-cases of the rate limiter:

1. For the client applications of your trading system: allow each client app to only send 5 orders per second
2. Throttle number of rewards on payment platform per user per month, or/and a global limit on rewards per user
3. For platforms like Medium: allow non-premium users to access only 3 premium stories per month
4. Early Fraud detection if possible can be dropped from middleware before reaching the web servers.
5. In Leetcode or some website where we can execute our code, rate limiting may be needed so that users don't spam the code execution service needlessly.

In general, a system design interview goes as below

1. **Gather scope requirements** — High-level idea on what is the intent, scope, and scale of the system. Standalone vs BeSpoke service etc. Distributed vs non-distributed. This is important because it helps us design keeping the scalability aspect in mind
2. **Functional and Nonfunctional requirements:** Functional requirements pertain to the APIs of your system and Non-Functional refers to the performance aspect such as availability, low latency, etc.

3. **Basic design** — start with a simple High-level diagram (a few boxes are fine) explaining the flow and various helper services you can think of. Here it is important to ask for an interview as to where they want you to deep dive. For example, if you are designing a ride-share service (Uber) then it depends on what the interviewer wants to focus on, maybe they are interested in driver pricing optimization or they want to improve the map search portion. Clarify whenever you are uncertain. System design interviews are DISCUSSIONS with NO PERFECT SOLUTION.

4. **Dive deeper** as per instructions and expand your design, it is a good practice to highlight possible shortcomings/bottlenecks in your design while you are making them.

5. **Validation:** validate that your design works, address possible bottlenecks, and seek interviewer feedback

6. **Future scope:** such as a metric service to measure the performance of your design; whether your design will work with 1 million customers, 100 million customers, etc. What may work for 1 million users may not scale for 100 million.

7. **closing points**: this aligns with point 6 but can be a general commentary on your design and you may also mention other solutions briefly.

Really understand the problem before designing

My first takeaway was how much information you actually need to have before diving into the design. While the context in this chapter features the interaction between an interviewer and software engineering candidate, this can apply when you want to design your own rate limiter.

Some questions you might ask are:

- What is the rate limiter for? If you already know the answer then convey your assumptions.
- Is this a client or server-side rate limiter? (We will be discussing server-side rate limiters here)
- How do we throttle API requests? Via IP address, user IDs, or something else?
- What is the scale of the system we are building for?

- Would the limiter be built into the application, or is it a service by itself?
- Do users need to know if they are being rate limited?

Why rate limiting is used?

Avoid resource starvation as a result of a Denial of Service (DoS) attack.

Ensure that servers are not overburdened.

Rate Limiting helps to protect services against abusive behaviors targeting the application layer like Denial-of-service (DOS) attacks, brute-force password attempts, brute-force credit card transactions, etc. These attacks are usually a barrage of HTTP/S requests which may look like they are coming from real users, but are typically generated by machines (or bots).

Functional and Non-Functional requirements:

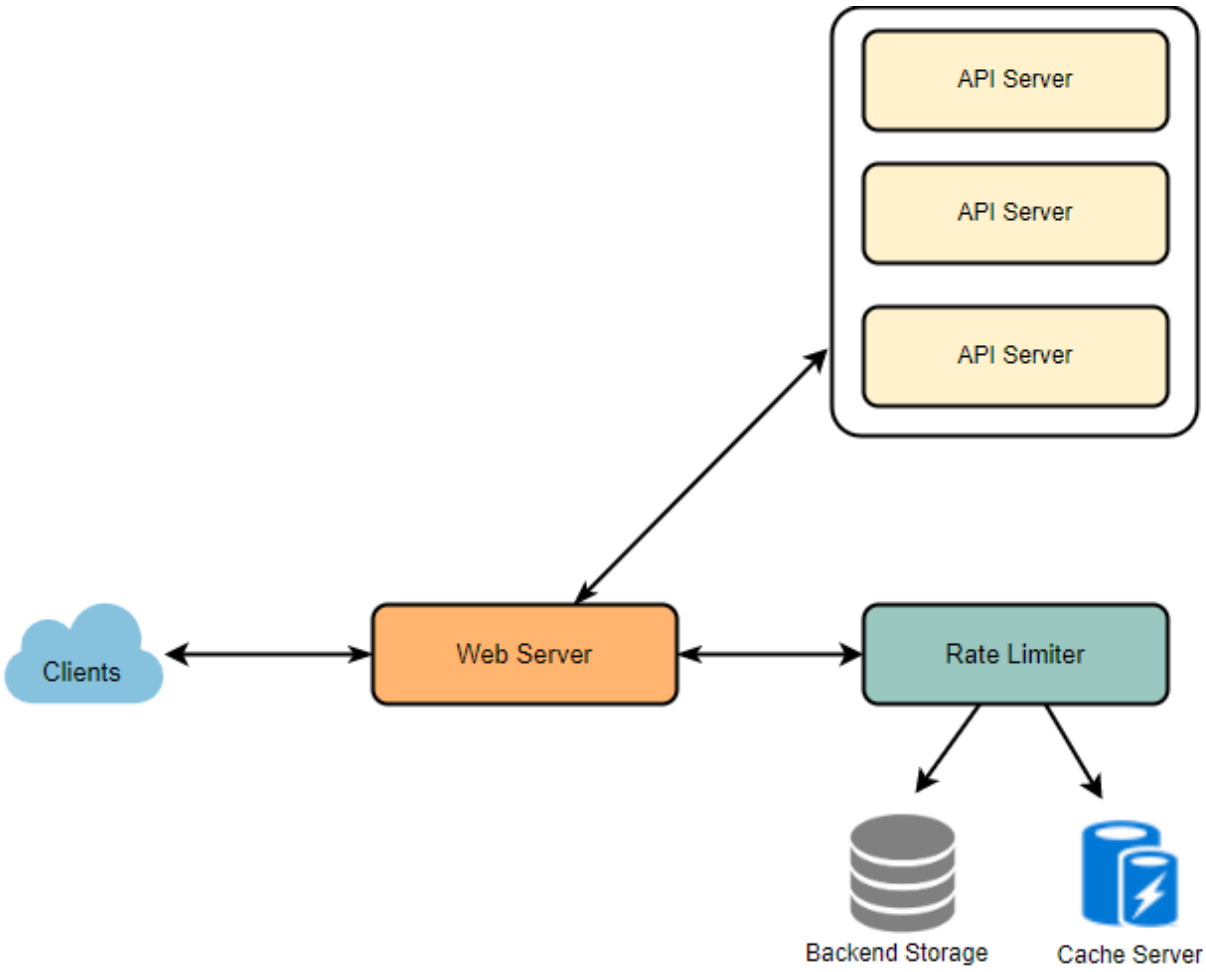
Functional Requirements:

1. Limit the number of requests an entity can send to an API within a time window, e.g., 15 requests per second.
2. The user should get an error message whenever the defined threshold is crossed within a single server or across a combination of servers (proper exception handling).

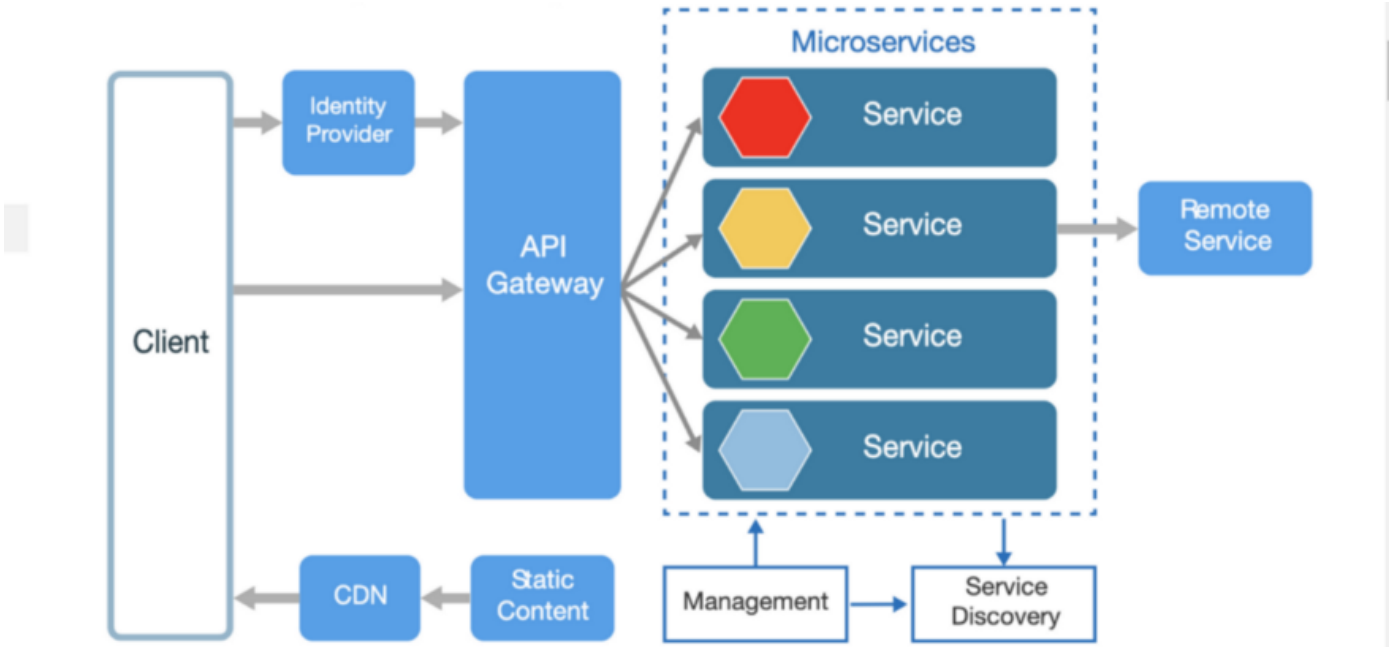
Non-Functional Requirements:

1. Highly available: The system should be highly available. The rate limiter should always work since it protects our service from external attacks.
2. Low latency: Our rate limiter should not introduce substantial latencies affecting the user experience.

High-level design for Rate Limiter



Where to place the rate limiter in your architecture?



We can place it in API Gateway, as an external service, or inside your application server. Mostly a matured architecture would be using an API gateway for authentication, IP whitelisting, SSL termination, etc. If we have sufficient control over the API gateway we can have our rate limiter inside API Gateway and have a fallback mechanism because they can fail too.

If we do not want to commit forever to the API gateway and decouple rate limiter service we can place it as an external service after the API gateway.

It is not so simple and straightforward to decide where to keep your rate limiter. For this discussion, we are assuming the rate limiter as an external service.

RateLimiter Algorithms

The popular ones include:

- ***Token Bucket***
- ***Fixed window counter***
- ***Sliding window***
- ***Sliding window counter***

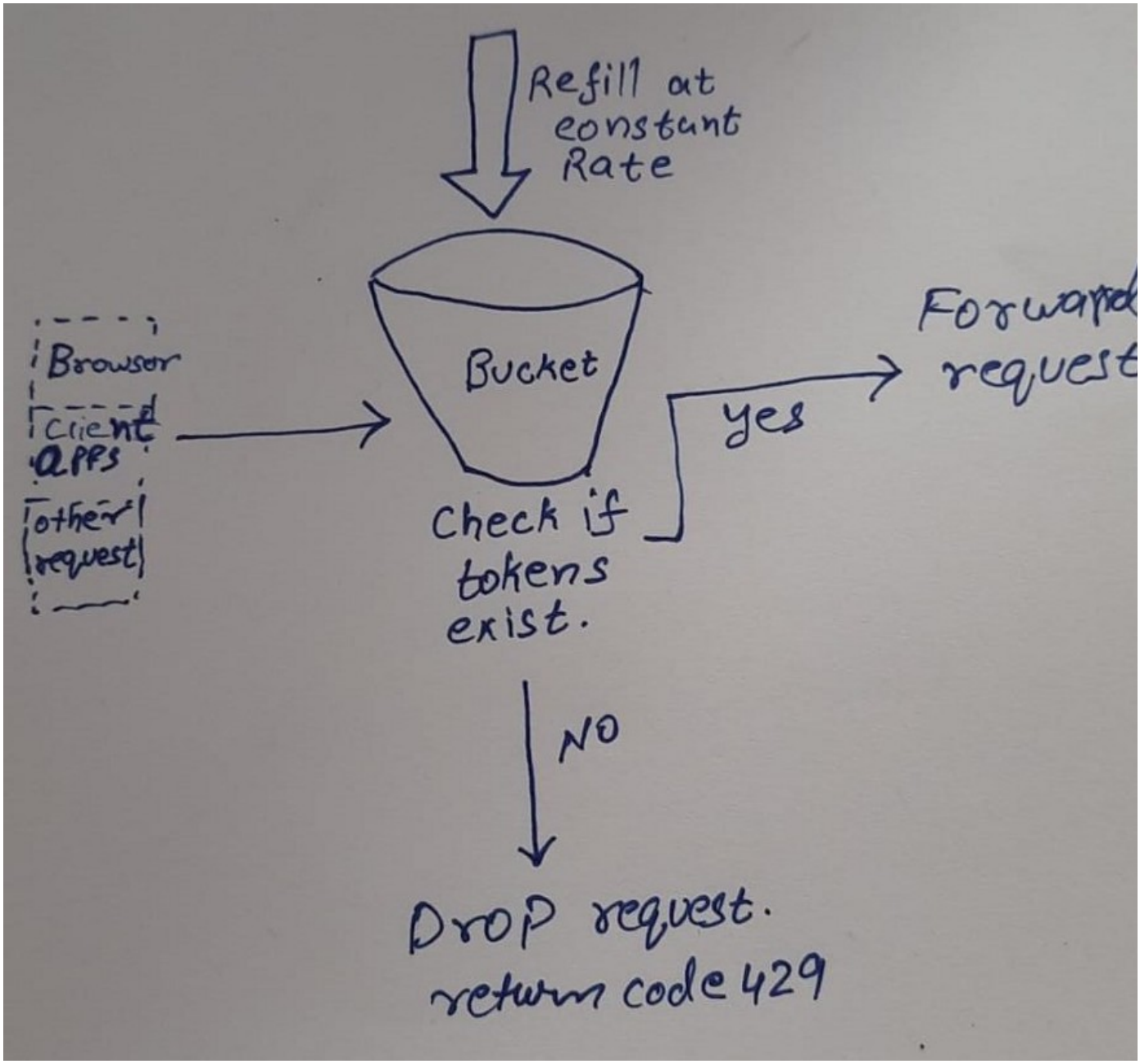
Identifiers can be user, IP, API path, or any other entity we wish to track request count for.

Token Bucket Algorithm:

We specify 2 parameters :

1. **Bucket size (maximum number of requests allowed for an identifier)**
2. **Refill rate (refill the bucket with bucket size every X seconds)**

At each incoming request, we check the remaining number of tokens of the identifier, if the remaining number of tokens is above the permitted threshold in the rule, we allow the request else we drop it.



👍Pros:

This algorithm is memory efficient as we are saving less amount of data per user for our application

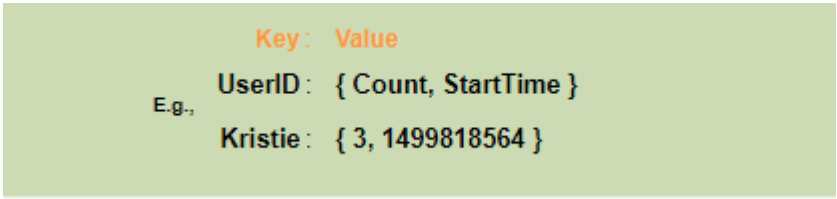
👎Cons:

1. Some other process has to feed tokens at regular intervals.
2. It can cause race conditions in a distributed environment.

Fixed window counter:

We specify 2 parameters :

1. **We would keep a count representing how many requests the identifier has made.**
2. **A timestamp when we started counting the requests. (starting time)**



We can keep it in a hashtable, where the 'key' would be the 'UserID' and 'value' would be a structure containing an integer for the 'Count' and an integer for the Epoch time.

For example, a rate limiter for a service that allows only 10 requests per hour will have the data model below. Here, the buckets are the windows of one hour, with values storing the counts of the requests seen in that hour.

```
{
 "1AM-2AM": 7,
 "2AM-3AM": 8
}
```

With the current model in the above example, if a new request is seen at 2:45AM, we get the count from the current bucket(2AM-3AM) which is 8 and verify that if processing one more request exceeds the permissible limit of 10, if that is the case, an exception is raised; if not(which is the case here), count of the bucket is incremented by unit(to 9) and the request is allowed to be processed. Only the counts of the current window are stored and older windows are deleted when a new window is created(i.e in the above case, if the hour changes, older bucket is deleted).

👍Pros:

- Easy to implement
- Less memory footprint, 'cause all that is being done is storing the counts

- Can use inbuilt concurrency with redis like technologies

👎Cons:

- This in incorrect. Explanation: In the above case, if all the 7 requests in the 1AM-2AM bucket occurs from 1:30AM-2AM, and all the 8 requests from 2AM-3AM bucket occur from 2AM-2:30AM, then effectively we have 15(7 + 8) requests in the time range of 1:30AM-2:30AM, which is violating the condition of 10req/hr.

Sliding Window algorithm:

- For every user, a queue of timestamps representing the times at which all the historical calls have occurred within the timespan of the recent most window is maintained.
- Whenever a new request occurs, a check is made for any timestamps older than the window time and these are deleted as they are no longer relevant(Instead of doing this at every request, this step can also be done periodically after every 'n' mins or when a certain length of the queue is reached)
- The new timestamp is appended to the user's queue.
- If the number of elements in the queue is not more than that of the allowed count, the request is let through, else an exception is raised.

👎Cons:

- High memory footprint. All the request timestamps need to be maintained for a window time, thus requiring lots of memory to handle multiple users or large window times
- The high time complexity for removing the older timestamps.

Sliding window counters:

- This is a hybrid of Fixed Window Counters and Sliding Window logs
- The entire window time is broken down into smaller buckets.
- Each bucket stores the request count corresponding to the bucket range.

For example, in order to build a rate limiter of 100 req/hr, say a bucket size of 20 mins is chosen, then there are 3 buckets in the unit time

For a window time of 2AM to 3AM, the buckets are:

```
{
 "2AM-2:20AM": 10,
 "2:20AM-2:40AM": 20,
 "2:40AM-3:00AM": 30
}
```

If a request is received at 2:50 AM, we find out the total requests in the last 3 buckets including the current, and add them, in this case, they sum up to 60 (<100), so a new request is added to the bucket of 2:40 AM–3:00 AM giving...

```
{
 "2AM-2:20AM": 10,
 "2:20AM-2:40AM": 20,
 "2:40AM-3:00AM": 31
}
```

This is not a completely correct, for example: At 2:50, a time interval from 1:50 to 2:50 should be considered, but in the above example the first 10 mins aren't considered (as it is a hyrid of fixed window and sliding window)and it may happen that in this missed 10 mins, there might've been a traffic spike and the request count might be 100 and hence the request is to be rejected. But by tuning the bucket size, we can reach a fair approximation of the ideal rate limiter.

👎Cons:

Works only for not-so-strict look-back window times, especially for smaller unit times.

Should we rate the limit by IP or by the user?

Let's understand the two scenarios captured below to answer this:

How does rate-limiting work with user logins?

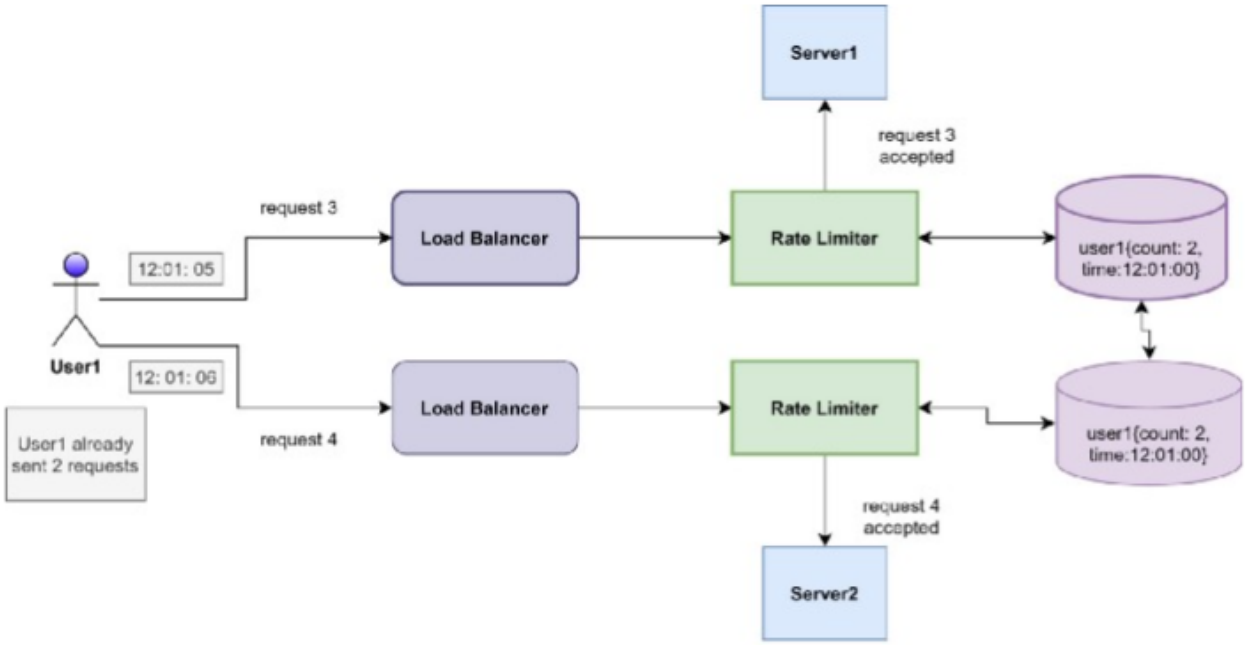
Users may find themselves locked out of an account if they unsuccessfully attempt to log in too many times in a short amount of time. This occurs when a website has login rate-

limiting in place.

Rate limiting on a login page can be applied according to the IP address of the user trying to log in, or according to the user's username. Ideally, it would use a combination of the two, because:

- If rate limiting is only applied by IP address, brute force attackers could bypass this by attempting logins from multiple IP addresses (perhaps by using a botnet).
- If it's only done by username, an attacker that has a list of known usernames can try a variety of commonly used passwords with those usernames and is likely to successfully break into at least a few accounts, all from the same IP address.

RateLimiter in a distributed system:



Challenges in a distributed system:

The above algorithms work very well for single-server applications. But the problem becomes very complicated when there is a distributed system involved with multiple nodes or app servers. It becomes more complicated if there are multiple rate-limited services distributed across different server regions. The two broad problems that come across in these situations are ***Inconsistency*** and ***Race Conditions***.

Inconsistency:

consider the following events in order

The rule for the user — maximum 4 requests allowed per second.

1. 'User 1' sends a request to Rate limiter server "A"
2. Server "A" updates the counter in **its** Redis cache node, the number of tokens is now 3 (was previously 4).
3. The same 'User 1' again sends a request but to server "B" whose Redis cache instance does not have the updated count for this user (user 1), makes the counter 3 (should be 2).

The solution to the above problem:

1. ***Sticky sessions:***
   Most clients will reach the data center closer to their application (via our geo-DNS), so if we enable "stickiness" on the load balancer, a client should always reach the same instance. And this would allow us to use a simple "local" rate-limiting. But this is not fault-tolerant, and adding or removing servers will cause issues, it is mostly recommended to go for stateless web-tier.
2. ***Locks and gossip Protocol:***
   We can use locks and use gossip protocol to propagate the change to other servers so that when the next request comes every node has updated data, this will increase the inter-service communication and thereby increase latency.
3. ***Centralized Redis store:***
   We can instead have a centralized Redis store and have all the rate limiter servers reach out to the same centralized store, but this does not scale well. Our rate limiter nodes can overwhelm the centralized Redis store.

The solution to the above-centralized Redis store: We can shard Redis store based on the 'UserID' to distribute the user's data. For fault tolerance and replication we should use Consistent Hashing. If we want to have different throttling limits for different APIs, we

can choose to shard per user per API. Take the example of URL Shortener; we can have different rate limiter for `createURL()` and `deleteURL()` APIs for each user or IP.

Race Conditions:

Race conditions happen in a ***get-then-set*** approach with high concurrency. Each request gets the value of the counter (in the case of sliding window counter and token bucket algorithm)and then tries to increment it. But by the time that writing operation is completed, several other requests have read the value of the counter(which is not correct). Thus a very large number of requests are sent than what was intended. This can be mitigated using locks on the read-write operation, thus making it atomic. But this comes at a performance cost as it becomes a bottleneck causing more latency.

Why Redis is chosen as a DB for the rate limiter, what problem does it solve?

Revise the token bucket algorithm:

- Each user has two keys associated with them: the token bucket, and a timestamp of the last time the bucket was refilled.
- When a user attempts to perform an action, we fetch the stored timestamp.
- We calculate how many tokens the user should have been granted since that last request.
- We can then proceed with the algorithm, using this new token count.

it fails when two processes/instances/servers need to share the rate limiter. Redis can batch operations into one atomic action, but to calculate how many tokens we need to give the user, we need at least two trips to redis: one to get the last timestamp, and one to set the new token count. we couldn't find a way to combine all needed operations into a single atomic set of Redis actions. Because of this, if two clients using the rate limiter both tried to verify a user action at the same time, we could have the following sequence of operations:

- The user has enough tokens left for one action.
- Not enough time has passed since the last action for more tokens to be granted.

- Client 1 gets the stored timestamp and token count.
- Client 2 gets the stored timestamp and token count.
- Client 1 calculates that there is no need to add tokens, allows the action, and tells redis to set the token count to 0.
- Client 2 **also** calculates that there is no need to add tokens, allows the action, and tells redis to set the token count to 0.

Needless to say, this is not ideal. If we have several dozen workers all processing push notification payloads, it would be possible for a user to be spammed with dozens of pushes all at once.

A better approach — sorted sets:

Fortunately, Redis has another data structure that we can use to prevent these race conditions — the **sorted set**. Here's the algorithm we came up with:

- Each user has a sorted set associated with them. The keys and values are identical, and equal to the (microsecond) times when actions were attempted.
- When a user attempts to perform an action, we first drop all elements of the set which occured before one interval ago. This can be accomplished with Redis's `ZREMRANGEBYSCORE` command.
- We fetch all elements of the set, using `ZRANGE(0, -1)`.
- We add the current timestamp to the set, using `ZADD`.
- We set a TTL equal to the rate-limiting interval on the set (to save space).
- After all operations are completed, we count the number of fetched elements. If it exceeds the limit, we don't allow the action.
- We also can compare the largest fetched element to the current timestamp. If they're too close, we also don't allow the action.

The advantage of this approach is that **all Redis operations can be performed as an atomic action**, using the MULTI command.

Redis Transactions allow the execution of a group of commands in a single step, they are centered around the commands `MULTI`*,* `EXEC`*,* `DISCARD` *and* `WATCH`*. Redis Transactions make two important guarantees:*

1. All the commands in a transaction are serialized and executed sequentially. A request sent by another client will never be served ***in the middle*** *of the execution of a Redis Transaction. This guarantees that the commands are executed as a single isolated operation.*

2. The `EXEC` *command triggers the execution of all the commands in the transaction, so if a client loses the connection to the server in the context of a transaction before calling the* `EXEC` *command none of the operations are performed.*

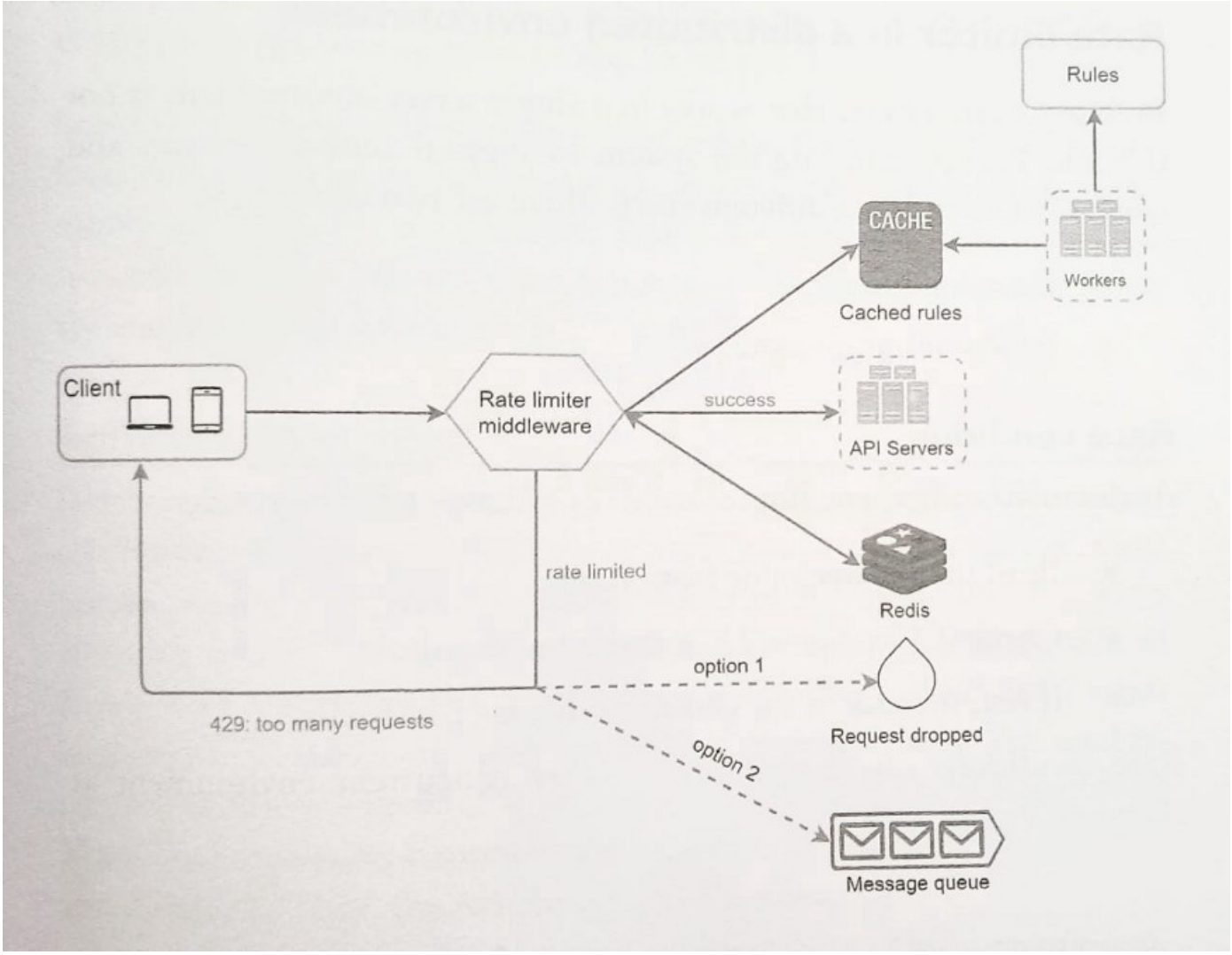
Do we need caching here? What should be the cache writing policy and what should be the cache eviction policy?

This system may get huge benefits from caching recent active users. Servers can quickly check if the cache contains the counter value for the user before hitting backend servers. This should be faster than going to the DB each time.

We may use the **Write-back cache** strategy by updating all counters and timestamps in cache only. Then, we can write to the database done at fixed intervals, e.g., 1 hour. This can ensure minimum latency added to the user's requests, which should improve the Lock latency problem.

When the server reads data, it can always hit the cache first. This will be useful when the user has hit their maximum limit. The rate limiter reads data without any updates.

We may use the Least Recently Used (LRU) as a cache eviction policy for our system.



How does a client know it is being throttled?

Clients can also handle rates at which they send requests if informed about limits. Certain HTTP response headers can be used to achieve this :

1. *X-Ratelimit-Remaining*: remaining number of allowed requests in the window
2. *X-Ratelimit-Limit*: number of calls client can make in time window
3. *X-Ratelimit-Retry-After*: number of seconds to wait

At the end of the article we should be able to answer the following questions:

1. What Questions should start asking the problem setter?
2. Where does the rate limiter sit in your system?
3. RateLimiter Algorithms. Discuss the advantages and the disadvantages of each.

4. Rate limiting should be done based on IP address or User id or something else?
5. In a distributed system where the rate limiter sits, how does it look?
6. What are the challenges in distributed rate limiter?
7. Why Redis is chosen as a DB and cache by many for rate limiter, what problem does it solve?
8. Do we need caching here? What should be the cache writing policy and what should be the cache eviction policy?
9. How does the client know it is being throttled?

References:

1. https://engineering.classdojo.com/blog/2015/02/06/rolling-rate-limiter/
2. https://medium.com/@saisandeepmopuri/system-design-rate-limiter-and-data-modelling-9304b0d18250
3. https://www.educative.io/courses/grokking-the-system-design-interview/3jYKmrVAPGQ#5.-What-are-different-types-of-throttling?

If you enjoyed the article, clap and consider following. Thank you :)