# Rate Limiting

## References

* <https://medium.com/figma-design/an-alternative-approach-to-rate-limiting-f8a06cf7c94c>
* <https://www.tibco.com/reference-center/what-is-rate-limiting>
* <https://medium.com/@saisandeepmopuri/system-design-rate-limiter-and-data-modelling-9304b0d18250>
* <https://medium.com/geekculture/system-design-basics-rate-limiter-351c09a57d14>
* <https://medium.com/figma-design/an-alternative-approach-to-rate-limiting-f8a06cf7c94c>

## What is it

* To limit the **number of requests to or from a system in each time period**
* A rate limiter caps how many requests a sender — this could be a user or an IP address — can issue in a specific window of time (e.g., 25 requests per minute)
* **Per user** - When the capacity of a service is shared among many users or consumers, it can apply rate limiting per user to provide fair and reasonable use, without affecting other users
* **IP address is the main way an application identifies who or what is making the request** - Typically tracking IP address and time elapses between each request
* These limits might be applied over longer time periods, or they might be applied to resources that are not measured by rate but by quantity allocated

## Use cases

* Help stop certain kinds of **malicious bot activity**
* It can also **reduce strain on web servers**
* **Preventing resource starvation -** thus improve API availability
* **Security** - prevent Brute force on functionalities like - login attempt, promo code etc. by limiting on user level
* Controlling **Operational costs**

## Applications

* **API rate limiting**
  + **Twitter** - Allows any Third-party app to refresh for new tweets or messages certain no. of times per hour. However, the limits don’t apply for twitter users (same for Instagram)
  + Avoid DoS attacks, spam attack by hackers
  + Limit automated scraping of the site's data - by rate limiting no of pages under a website are accessed

## Throttling

Controlling the no of requests at application or at API level in each period

* + **Process of controlling the usage of APIs** by customers during a given period
  + Can be defined at application level or API level
  + When the throttle limit is crossed, server returns HTTP 429 - Too many requests
  + *Hard throttling*
  + No of API requests cannot exceed the throttle limit
  + *Soft throttling*
  + Set the request limit to exceed a certain percentage
  + *Elastic or Dynamic throttling*
  + No of requests can go beyond the threshold if the system has available resources

## Design considerations

* System
  + On a single machine, single threaded scenario
  + On a single machine, multi-threaded scenario - handling race conditions
  + Distributed system - distributed cache usage like redis (in memory data storage, counter)
  + Limit on Client side - Prevent network calls from client to server for all the excess requests
* Strategy
  + User - Limit per user in each period of time
  + Concurrency - Limit no of parallel sessions allowed for a user - Helps mitigate DDOS attacks
  + Location / ID - Requests not arriving from target demography
  + Server - When specific servers need most of the requests, i.e., servers are strongly coupled to specific functions

## Implementation / Algorithms

* + Using persistent memory stores like SQL is a bad as the time taken for disk seeks can hamper the rate limiter granularity. Using a in memory cache (main memory access) can be much faster
  + *Leaky bucket*
    - Requests beyond the capacity of queue are discarded
    - Here input flow can burst however the output is in a constant rate
    - Cons - Starving of new requests
  + *Token bucket*
    - Like leaky bucket. Here we assign **tokens on a user level**
    - For a given time duration, no of requests that a user can receive is defined
    - Every time a new request arrives at server - fetch token and update token
    - Fetch - Current no of tokens for a user is fetched. If greater, request is dropped
    - Update - If lesser for the time duration, request is accepted, and token is appended
    - Cons - Can cause race condition in a distributed environment
  + *Fixed Window counters*
    - Very basic way of tackling the problem
    - One bucket for each of the unit time window
    - Each bucket maintains the count of no. of requests in that particular window

{

"1AM-2AM": 7,

"2AM-3AM": 8

}

* Cons - 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/hour
* *Sliding Window logs*
  + **For every user, a queue of timestamps** representing the times at which all the historical calls have occurred within the timespan of recent most window is maintained
  + Involves maintaining a time stamped log of requests at the user level and then discard all requests with timestamps beyond a threshold
  + Look out for older requests and filter them out
  + Cons - High memory footprint as all the request timestamps needs to be maintained and expensive to compute no of requests each time. So, it does not scale well
* *Sliding window counters*
  + Hybrid of Fixed window counters (low processing cost) and Sliding window logs (improved boundary)
  + **Entire window time is broken down into smaller buckets.** Size of each bucket depends on how much elasticity is allowed for the rate limiter
  + Each bucket stores the request count **corresponding to the bucket range**

{

"2:00AM-2:20AM": 10,

"2:20AM-2:40AM": 20,

"2:40AM-3:00AM": 30

}

* NOTE: 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 isn’t considered, 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
* Pros - Flexible to scale, avoids starvation problem and bursting problem
* Cons - Works only for not-so-strict look back window times, especially for smaller unit times

## Rate limiting in Distributed systems

* Diff between Single server and distributed server implementation
* Even more complicated if there are multiple rate limited services distributed across diff server regions - **Inconsistency** and **Race conditions**
* *Inconsistency*
  + App servers distributed across regions have their own limiters - We need to define a **global rate limiter**
  + A consumer could surpass the global rate limiter individually if it receives a lot of requests in a small-time frame. The greater the number of nodes, the more likely the user will exceed the global limit
  + Solution 1 - **Sticky Session** - Sticky sessions in your load balancers so that each consumer gets sent to exactly one node. Downside - fault tolerance and scaling problems when node gets overloaded
  + Solution 2 - **Centralized data store** - Like Redis or Cassandra to handle counts for each window and consumer. Added latency could be a problem but flexibility is elegant
* *Race Conditions*
  + Happens in a get-then-set approach with high concurrency. Where each request gets the value of counter and later tries to increment it
  + Solution - Using locks on read-write operation thus making it atomic. But this comes at a performance cost as it becomes bottleneck causing more latency