 master ▾


...

[system-design-interview](#) / [problems](#) / Build_Rate_Limiter.md



wuyichen24 Update Build_Rate_Limiter.md

 History

 1 contributor

 115 lines (100 sloc) | 6.3 KB

...

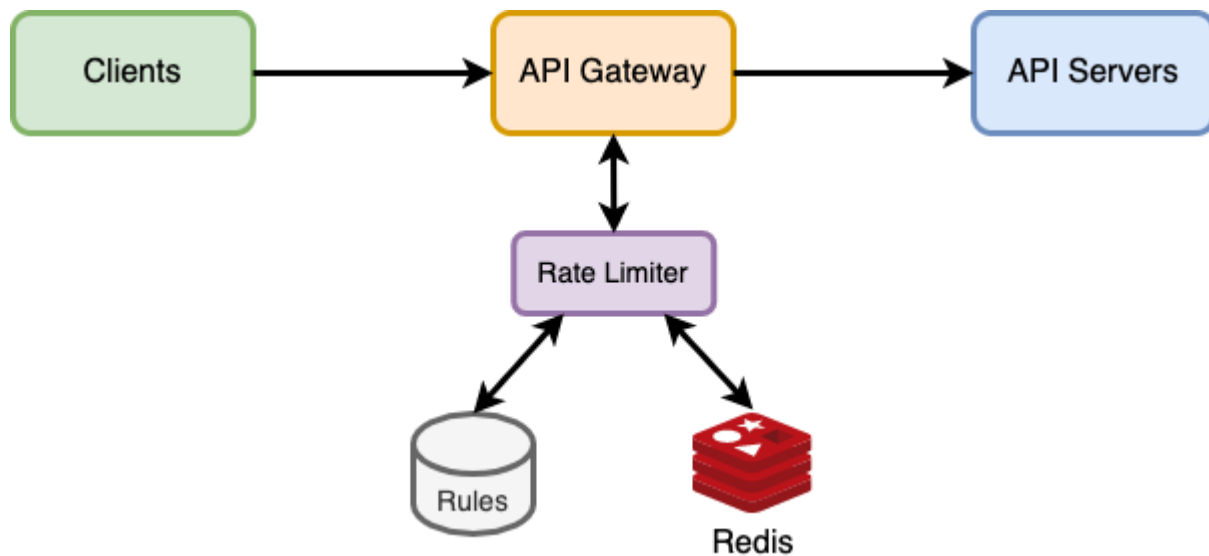
Build Rate Limiter

Real-life examples

Requirements clarification

- **Functional requirements**
 - Limit requests: Limit the number of requests an entity can send to an API within a time window.
 - Exception handling: The user should get an error message when the user excess the threshold.
 - Distribution: The rate limiter can monitor requests among multiple servers.
- **Non-functional requirements**
 - High availability (The rate limiter should always work for protecting the servers from external attacks).
 - Low latency (The rate limiter should not introduce substantial latencies affecting the user experience).

High-level design

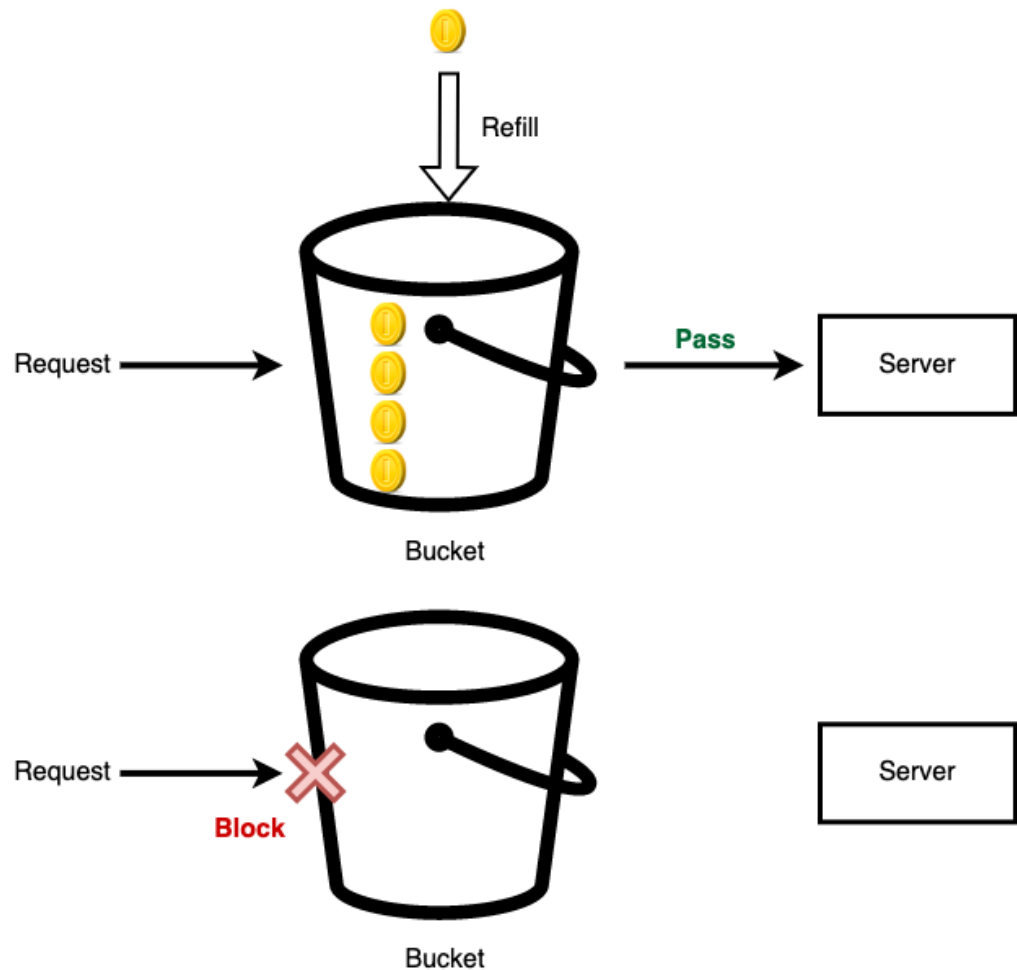


- Redis
 - Use Redis to store the counter.
 - Redis is fast and supports time-based expiration strategy.

Detailed design

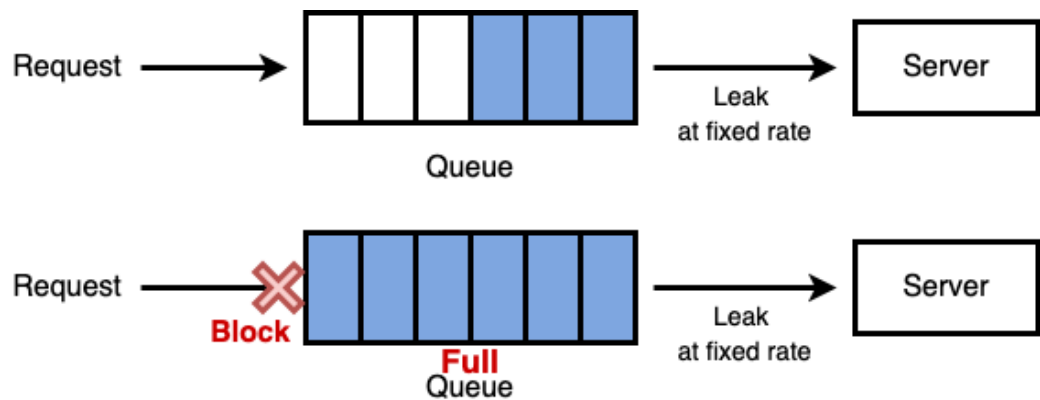
Algorithms for rate limiting

- Token bucket
 - Mechanism
 - The bucket can hold at the most B tokens.
 - A token is added to the bucket every R seconds. If the bucket is full, no more tokens are added.
 - When a request arrives
 - If there is a token in the bucket, the request will take one token out from the bucket and it goes through.
 - If there is no token in the bucket, the request will be blocked.

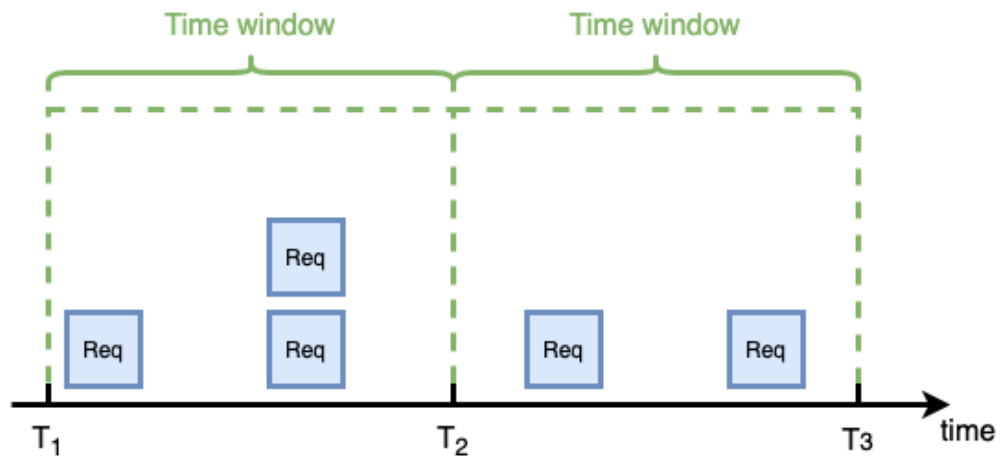


- Implementation details
 - Based on rate-limiting rules, we may need multiple buckets:
 - If we need to throttle request based on IP addresses/users. so each IP address/user requires a bucket.
 - If we need to throttle request based on request types (make a post, get feeds, etc.), so each request type requires a bucket.
- Pros
 - Easy to implement.
 - Memory efficient.
- Cons
 - Hard to tune the bucket size (B) and the refill rate (R) properly.
- **Leaking bucket**
 - Mechanism
 - A FIFO queue works as a bucket and it can hold at the most N requests.
 - The FIFO queue pops out (leak) a request at a fixed rate and let it be processed.
 - When a request arrives

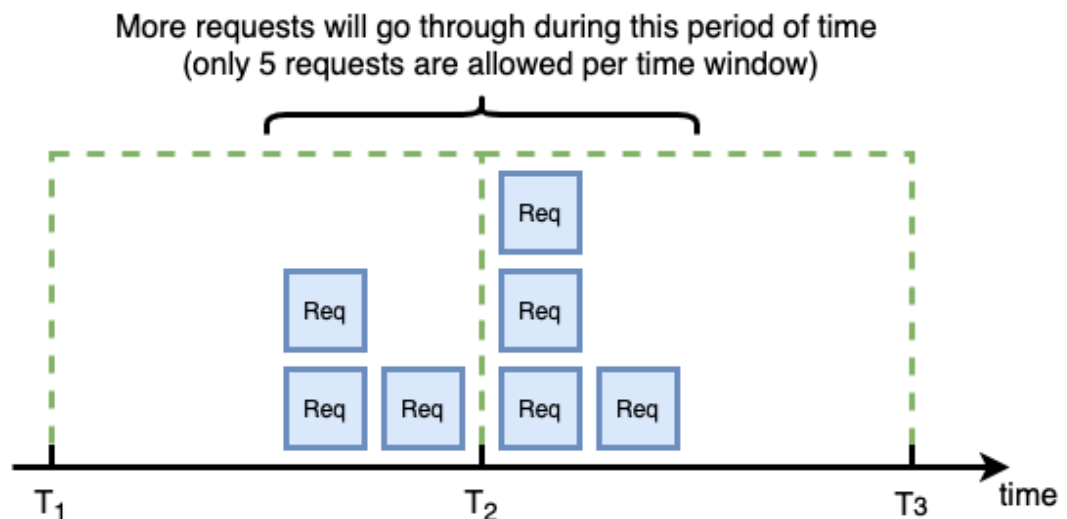
- If the queue is not full, the request will be added to the queue.
- If the queue is full, the request will be blocked.



- Pros
 - Memory efficient.
- Cons
 - Recent requests will be blocked and only old requests will be processed when the queue is full.
 - Token bucket can send large bursts at a faster rate while leaky bucket always sends requests at constant rate.
- **Fixed window counter**
 - Mechanism
 - Divide the timeline to a fixed time window and assign a counter to each window.
 - When a request arrives
 - If the counter of the current time window doesn't reach the threshold, increment the counter by one and the request goes through.
 - If the counter of the current time window reached the threshold, the request will be blocked.

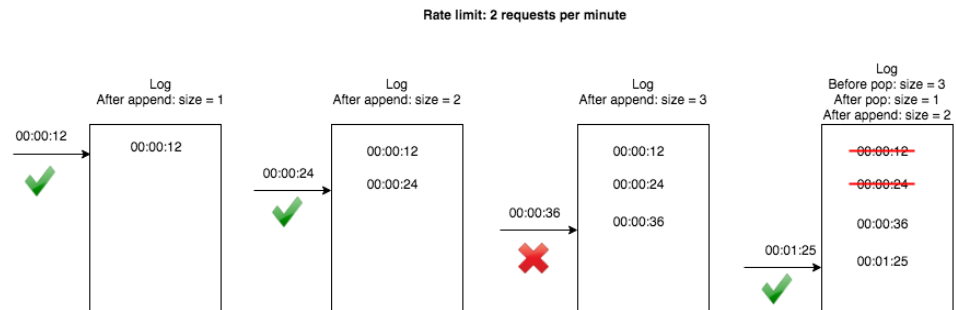


- Pros
 - Easy to implement.
 - Memory efficient.
- Cons
 - A traffic spike at the edges of a window could cause more requests than the allowed quota to go through.

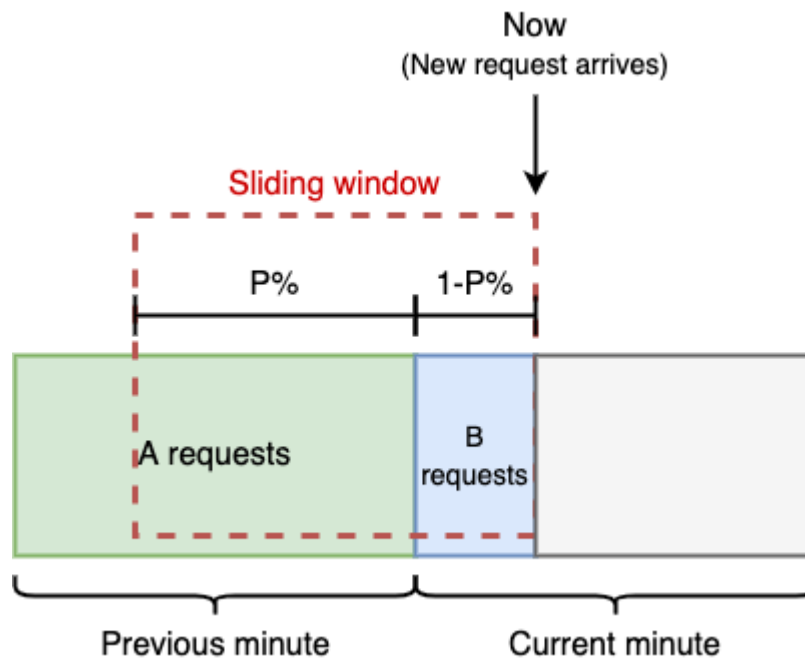


- Sliding window log
 - Mechanism
 - Use timestamp logs to track every request's timestamp.
 - When a request arrives
 - Only focus on the logs are in the current time window (From now to the start of the current time window) and ignore the logs are not in the current time window.
 - Count how many logs are in the current time window

- If the number doesn't reach the threshold, the request will go through.
- If the number reaches the threshold, the request will be blocked.



- Pros
 - Avoid the drawback of the fixed window counter algorithm.
- Cons
 - Memory inefficient (For each request, have to filter logs and count the number of logs)
- **Sliding window counter**
 - Concepts
 - The sliding window counter is the combination of the fixed window counter and the sliding window log.
 - The basic idea of sliding window counter is to add a weighted count in previous time period to the count in current period.
 - Mechanism
 - Given the limiting rate is N requests per minute.
 - For current sliding window, there are overlaps on both the current minute and the previous minute. The overlap on the previous minute is $P\%$.
 - There were A requests received in the previous minute and there were B requests received in the current minute.
 - So the number of requests in the current sliding window will be $B + P\% * A$.
 - Compare the number of requests in the current sliding window with threshold
 - If the number doesn't reach the threshold ($B + P\% * A < N$), the request will go through.
 - If the number reaches the threshold ($B + P\% * A \geq N$), the request will be blocked.



- Pros
 - Memory efficient.

How to handle blocked requests

- **Option 1:** Drop the requests and return the 429 error (too many requests) to the client.
- **Option 2:** Put the requests to a queue so that they can be processed later.

Types of rate limiting

- **Hard rate limiting:** The number of requests cannot exceed the threshold.
- **Soft rate limiting:** The number of requests can exceed the threshold for a short period of time.
- **Elastic/Dynamic rate limiting:** The number of requests can exceed the threshold if the system has some resources available.

Other optimization options

- Use client cache to avoid making too frequent API calls.

References

- <https://medium.com/swlh/rate-limiting-fdf15bfe84ab>
- <https://hechao.li/2018/06/25/Rate-Limiter-Part1/>