Rate limiter

- Goals
- Algorithm
  - Token bucket
  - Leaky bucket
  - Fixed window
  - Sliding log
  - Sliding window
- Single machine rate limit
  - Guava rate limiter
    - Implementation
      - Producer consumer pattern
    - Record the next time a token is available
      - Warm up feature
  - Ratelimiter within Resiliency4J
- Distributed rate limit
  - Sticky sessions

    - Nginx based rate limiting
    - Redis based rate limiter
        - Implementation
            - Sliding log implementation using ZSet
            - Sliding window implementation
            - Token bucket implementation
        - Challenges
            - How to handle race conditions
            - How to handle the additional latency introduce by performance
            - How to avoid multiple round trips for different buckets:
            - Performance bottleneck and single point failure due to Redis
            - Static rate limit threshold
    - Ratelimiter within CloudBouncer
    - Redis cell rate limiter

Goals

- Sharing access to limited resources: Requests made to an API where the limited resources are your server capacity, database load, etc.
- Security: Limiting the number of second factor attempts that a user is allowed to perform, or the number of times they’re allowed to get their password wrong.
- Revenue: Certain services might want to limit actions based on the tier of their customer’s service, and thus create a revenue model based on rate limiting.

Algorithm

Token bucket

- The token bucket limits the average inflow rate and allows sudden increase in traffic.
    - Steps
        a. A token is added every t time.

   b. The bucket can hold at most b tokens. If a token arrive when bucket is full the token will be discarded.
   c. When a packet of m bytes arrived m tokens are removed from the bucket and the packet is sent to the network.
   d. If less than n tokens are available no tokens will be removed from the bucket and the packet is considered to be non-comformant.
  - Pros
    - Smooth out the requests and process them at an approximately average rate.
  - Cons
    - A burst of request could fill up the queue with old requests and starve the more recent requests from being processed. Does not guarantee that requests get processed within a fixed amount of time. Consider an antisocial script that can make enough concurrent requests that it can exhaust its rate limit in short order and which is regularly overlimit. Once an hour as the limit resets, the script bombards the server with a new series of requests until its rate is exhausted once again. In this scenario the server always needs enough extra capacity to handle these short intense bursts and which will likely go to waste during the rest of the hour.

Leaky bucket

- The leaky bucket limits the constant outflow rate, which is set to a fixed value. Imagine a bucket partially filled with water and which has some fixed capacity (τ). The bucket has a leak so that some amount of water is escaping at a constant rate (T)
- Steps
  i. Initialize the counter to N at every tick of the clock
  ii. If N is greater than the size of the packet in front of the queue send the packet to network and decrement the counter by the size of the packet.
  iii. Reset the counter and go to Step - 1.
- Pros:
  - The leaky bucket produces a very smooth rate limiting effect. A user can still exhaust their entire quota by filling their entire bucket nearly instantaneously, but after realizing the error, they should still have

    - access to more quota quickly as the leak starts to drain the bucket.
    - The token bucket allows for sudden increase in traffic to some extent, while the leaky bucket is mainly used to ensure the smooth outflow rate.
- Cons:
    - When compared with token bucket, packet will be discarded instead of token.
    - The leaky bucket is normally implemented using a background process that simulates a leak. It looks for any active buckets that need to be drained, and drains each one in turn. The naive leaky bucket's greatest weakness is its "drip" process. If it goes offline or gets to a capacity limit where it can't drip all the buckets that need to be dripped, then new incoming requests might be limited incorrectly. There are a number of strategies to help avoid this danger, but if we could build an algorithm without a drip, it would be fundamentally more stable.

Fixed window

- Steps
    1. A window of size N is used to track the requests.
    2. Each request increments the counter for the window.
    3. If the counter exceeds a threshold, the request is discarded.
- Pros
    - It ensures recent requests get processed without being starved by old requests.
- Cons
    - Stamping elephant problem: A single burst of traffic that occurs near the boundary of a window can result in twice the rate of requests being processed, because it will allow requests for both the current and next windows within a short time.
    - If many consumers wait for a reset window, for example at the top of the hour, then they may stampede your API at the same time.

Sliding log

- Steps

   i. Tracking a time stamped log for each consumer's request.
   ii. These logs are usually stored in a hash set or table that is sorted by time. Logs with timestamps beyond a threshold are discarded.
   iii. When a new request comes in, we calculate the sum of logs to determine the request rate. If the request would exceed the threshold rate, then it is held.
- Pros
  - It does not suffer from the boundary conditions of fixed windows. The rate limit will be enforced precisely. - Since the sliding log is tracked for each consumer, you don't have the stampede effect that challenges fixed windows
- Cons
  - It can be very expensive to store an unlimited number of logs for every request. It's also expensive to compute because each request requires calculating a summation over the consumer's prior requests, potentially across a cluster of servers.

Sliding window

- Steps
   i. Like the fixed window algorithm, we track a counter for each fixed window.
   ii. Next, we account for a weighted value of the previous window's request rate based on the current timestamp to smooth out bursts of traffic.
- Pros
  - It avoids the starvation problem of leaky bucket.
  - It also avoids the bursting problems of fixed window implementations.
- Please see the section on https://hechao.li/2018/06/25/Rate-Limiter-Part1/ for detailed rate limiter implementations.

Single machine rate limit

Guava rate limiter

- Implemented on top of token bucket. It has two implementations:

- SmoothBursty / SmoothWarmup (The RateLimiterSmoothWarmingUp method has a warm-up period after teh startup. It gradually increases the distribution rate to the configured value. This feature is suitable for scenarios where the system needs some time to warm up after startup.)

Implementation

Producer consumer pattern

- Def: Use a producer thread to add token, the thread who uses rate limiter act as consumer.
- Cons:
  - High cost for maintaining so many threads: Suppose use server cron timer as producer to add token. Suppose the goal is to rate limit on user visiting frequency andd there are 6 million users, then 6 million cron functionality needs to be created.
  - Rate limiting are usually used under high server loads. During such peak traffic time the server timer might not be that accurate and reliable.

Record the next time a token is available

- Each time a token is expected, first take from the storedPermits; If not enough, then compare against nextFreeTicketMicros (update simultaneously using resync function) to see whether freshly generated tokens could satisfy the requirement. If not, sleep until nextFreeTicketMicros to acquire the next available fresh token.

```
// The number of currently stored tokens
double storedPermits;
// The maximum number of stored tokens
double maxPermits;
// The interval to add tokens
double stableIntervalMicros;
/**
 * The time for the next thread to call the acquire() method
 * RateLimiter allows preconsumption. After a thread preconsumes
any tokens,
 the next thread needs to wait until nextFreeTicketMicros to
acquire tokens.
 */
```

```
private long nextFreeTicketMicros = 0L;
```

```
/**
 * Updates {@code storedPermits} and {@code
nextFreeTicketMicros} based on the current time.
 */
void resync(long nowMicros) {
    // if nextFreeTicket is in the past, resync to now
    if (nowMicros > nextFreeTicketMicros) {
      double newPermits = (nowMicros - nextFreeTicketMicros) /
coolDownIntervalMicros();
      storedPermits = min(maxPermits, storedPermits +
newPermits);
      nextFreeTicketMicros = nowMicros;
    }
}
```

Warm up feature

- Motivation: How to gracefully deal past underutilization
    - Past underutilization could mean that excess resources are available. Then, the RateLimiter should speed up for a while, to take advantage of these resources. This is important when the rate is applied to networking (limiting bandwidth), where past underutilization typically translates to "almost empty buffers", which can be filled immediately.
    - Past underutilization could mean that "the server responsible for handling the request has become less ready for future requests", i.e. its caches become stale, and requests become more likely to trigger expensive operations (a more extreme case of this example is when a server has just booted, and it is mostly busy with getting itself up to speed).
- Implementation
    - When the RateLimiter is not used, this goes right (up to maxPermits)
    - When the RateLimiter is used, this goes left (down to zero), since if we have storedPermits, we serve from those first
    - When *unused*, we go right at a constant rate! The rate at which we move to the right is chosen as maxPermits / warmupPeriod. This

ensures that the time it takes to go from 0 to maxPermits is equal to warmupPeriod.

- When *used*, the time it takes, as explained in the introductory class note, is equal to the integral of our function, between X permits and X-K permits, assuming we want to spend K saved permits.

```
             ^ throttling
             |
       cold  +                  /
    interval |                 /.
             |                / .
             |               /  .   ← "warmup period" is the
area of the trapezoid between
             |              /   .     thresholdPermits and
maxPermits
             |             /    .
             |            /     .
             |           /      .
      stable +----------/  WARM .
    interval |          .   UP  .
             |          . PERIOD.
             |          .       .
           0 +----------+-------+--------------→ storedPermits
             0 thresholdPermits maxPermits
```

- References
  i. https://segmentfault.com/a/1190000012875897?spm=a2c65.11461447.0.0.74817a50Dt3FUO
  ii. https://www.alibabacloud.com/blog/detailed-explanation-of-guava-ratelimiters-throttling-mechanism_594820

Ratelimiter within Resiliency4J

- https://dzone.com/articles/rate-limiter-internals-in-resilience4j
- https://blog.csdn.net/mickjoust/article/details/102411585

Distributed rate limit

Sticky sessions

- The simplest way to enforce the limit is to set up sticky sessions in your load balancer so that each consumer gets sent to exactly one node. The disadvantages include a lack of fault tolerance and scaling problems when nodes get overloaded.

Nginx based rate limiting

Redis based rate limiter

- Use a centralized data store such as Redis to store the counts for each window and consumer.

Implementation

Sliding log implementation using ZSet

- See Dojo engineering blog for details
    i. Each identifier/user corresponds to a sorted set data structure. The keys and values are both equal to the (microsecond) times at which actions were attempted, allowing easy manipulation of this list.
    ii. When a new action comes in for a user, all elements in the set that occurred earlier than (current time - interval) are dropped from the set.
    iii. If the number of elements in the set is still greater than the maximum, the current action is blocked.
    iv. If a minimum difference has been set and the most recent previous element is too close to the current time, the current action is blocked.
    v. The current action is then added to the set.
    vi. Note: if an action is blocked, it is still added to the set. This means that if a user is continually attempting actions more quickly than the allowed rate, all of their actions will be blocked until they pause or slow their requests.
    vii. If the limiter uses a redis instance, the keys are prefixed with namespace, allowing a single redis instance to support separate rate limiters.
    viii. All redis operations for a single rate-limit check/update are performed as an atomic transaction, allowing rate limiters running on separate processes or machines to share state safely.

Sliding window implementation

- https://blog.callr.tech/rate-limiting-for-distributed-systems-with-redis-and-lua/
- https://github.com/wangzheng0822/ratelimiter4j

Token bucket implementation

- https://github.com/vladimir-bukhtoyarov/bucket4j

Challenges

How to handle race conditions

1. One way to avoid this problem is to put a "lock" around the key in question, preventing any other processes from accessing or writing to the counter. This would quickly become a major performance bottleneck, and does not scale well, particularly when using remote servers like Redis as the backing datastore.
2. A better approach is to use a "set-then-get" mindset, relying on Redis' atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.
3. Use Lua scripts for atomic and better performance.

How to handle the additional latency introduce by performance

1. In order to make these rate limit determinations with minimal latency, it's necessary to make checks locally in memory. This can be done by relaxing the rate check conditions and using an eventually consistent model. For example, each node can create a data sync cycle that will synchronize with the centralized data store.
2. Each node periodically pushes a counter increment for each consumer and window it saw to the datastore, which will atomically update the values. The node can then retrieve the updated values to update it's in-memory version. This cycle of converge → diverge → reconverge among nodes in the cluster is eventually consistent.

- https://konghq.com/blog/how-to-design-a-scalable-rate-limiting-algorithm/

-

How to avoid multiple round trips for different buckets:

- Use Redis Pipeline to combine the INCRE and EXPIRE commands
- If using N multiple bucket sizes, still need N round trips to Redis.
  - TODO: Could we also combine different bucket size together? How will the result for multiple results being passed back from Redis pipeline
- Redis rate limiter implementation in python

Performance bottleneck and single point failure due to Redis

- Solution: ??

Static rate limit threshold

- Concurrency rate limit
  - Netflix Concurrency Limits: https://github.com/Netflix/concurrency-limits
  - Resiliency 4j said no for cache-based distributed rate limit: https://github.com/resilience4j/resilience4j/issues/350
  - Resiliency 4j adaptive capacity management: https://github.com/resilience4j/resilience4j/issues/201

Ratelimiter within CloudBouncer

- Use gossip protocol to sync redis counters
  - https://yahooeng.tumblr.com/post/111288877956/cloud-bouncer-distributed-rate-limiting-at-yahoo

Redis cell rate limiter

- An advanced version of GRCA algorithm
- References
  - You could find the intuition on https://jameslao.com/post/gcra-rate-limiting/
  - It is implemented in Rust because it offers more memory security. https://redislabs.com/blog/redis-cell-rate-limiting-redis-module/