Rate Limiter Tech Doc [Personal Project]

Objective and Scope of Document

This document captures the technical solution for the design of a rate limiter.
Github Repo : https://github.com/vaibhavahuja/rate-limiter

Requirements

- Limit the requests sent to the service as per the rate defined.
- Shared amongst different services.
- Allows multiple services to register rules, on the basis of which rate limiting is supposed to happen. Rules can be of the following types :
  - field `x` (from the request) should not appear more than `y` times per minute
  - limit requests to the service to `y` times per 'time_unit'
  - Tells what return message should be sent to the client in case of request rejection. For example :
    - Retry with exponential backoff
    - Retry with fixed time
    - Daily Limit Reached
- Minimal overhead latency
- High fault tolerance

Technical Evaluations

1. Rate Limiting Algorithm Choice

Following rate limiting algorithms were considered, and sliding window counter was chosen amongst them.

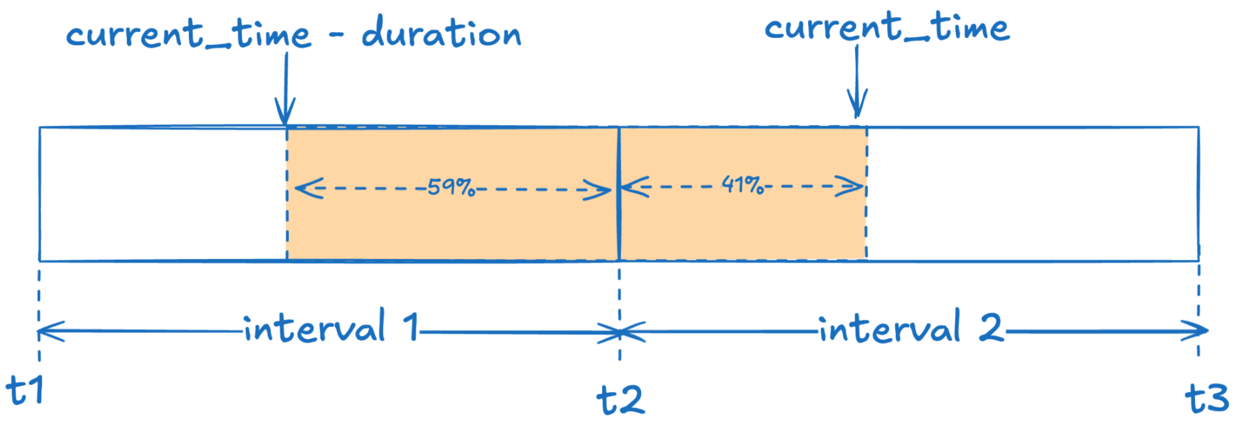
Rate Limiting Algorithm
Token Bucket Algorithm
Leaking Bucket
Fixed Window Counter
Sliding Window Log
Sliding Window Counter

Sliding Window Counter Algorithm Implementation

Sliding Window Counter algorithm takes in account the number of requests in the current sliding bucket. The number of requests in the sliding bucket is calculated as shown in the example below.
If number of requests in sliding bucket > rate_limit_defined -> cancel request

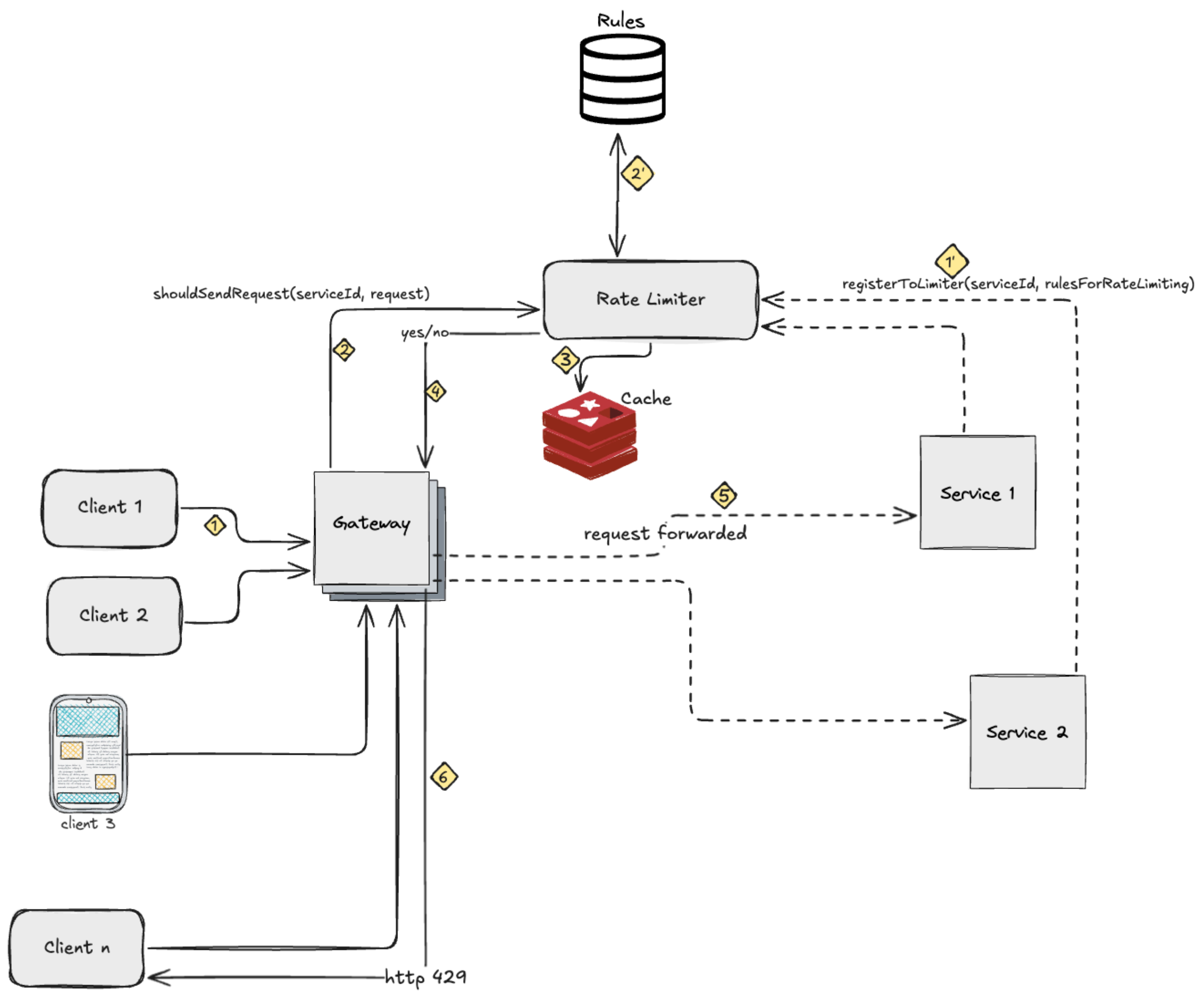
This algorithm assumes that requests were thoroughly distributed over the previous time interval window.

In the below example, say a request X comes at current_time. We need to determine if we should allow the request to go through or not.
Duration is the time interval (t2-t1). t2 - t1 == t3 - t2



```
rate_limit_defined : 40 requests per interval
number of requests in interval#2 until current_time : 5
total number of requests in interval#1 = 50
duration = t2 - t1 = t3 - t2
sliding window current time = current_time - duration
current_time = 41% of interval#2
total_requests_in_sliding_window = ceil((1 - 0.41) * requests_in_interval#1 +
requests_in_interval_2)
= ceil(0.59*50 + 5) = 35
if total_requests_in_sliding_window > rate_limit_defined : reject request!
35 > 40 == false
so request will go through
```

HLD



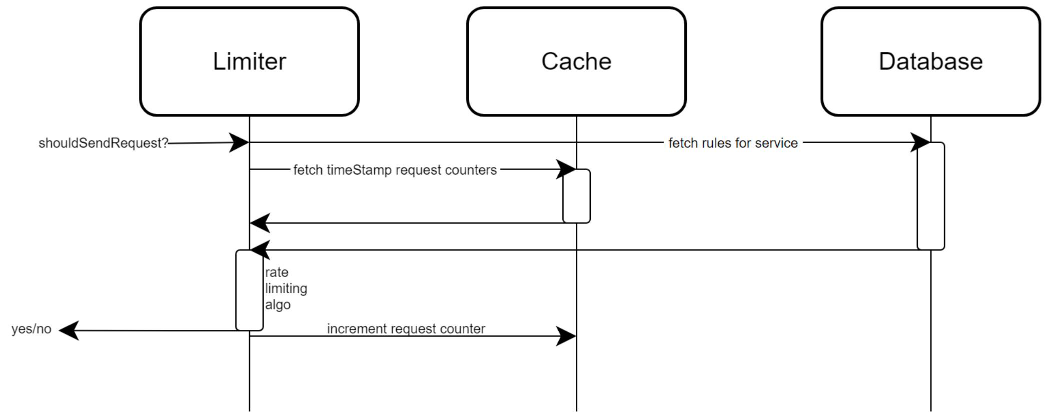
Rate Limiting Flow

1. Clients hit the gateway to send the request to Service x.
2. Gateway asks the Rate Limiter Service if it shouldSendRequest to service x.
3. Rate Limiter checks in the cache, for the given request user id, and rules for that service and makes a decision.
4. Rate Limiter Service returns true/false depending if client has breached the rate limit or not.
5. If true, Gateway forwards the request to the service.
6. If false, Gateway responds with 429 to the client and other meta information informing that the user has reached the rate limit.

Service registry flow

1. Service registers to the rate limiter, giving information about it and the rate limiting rules it wishes to apply.
2. Rate limiter stores the rules in the datastore, and periodically caches the rules in memory for evaluation.

LLD



Database Design

Service_id (primary key)	rules
1	`{ "field": "user_id", "rate": { "requests_per_unit": 5, "unit": "minute" }, "request_rejection_message":"retry-with-exponential-backoff" }`
2	`{ "rate": { "requests_per_unit": 10, "unit": "day" }, "request_rejection_message": "exhausted-daily-limit" }`

Cache Design

Key, Value struct

Key : will serialize the CacheKey as defined below
Value : Counter (Integer)

```
CacheKey (deserialized)
{
  "ServiceId": 1,
  "Field": "Hello",
  "TimeValue": "2023-02-07T20:23:00Z"
}
```

TimeValue for cacheKey will be same for the entire duration_unit.
Example :
duration_unit : *hour*
CurrentTime : "2023-02-07T20:23:12Z"
TimeValue : "2023-02-07T20:00:00Z" (stored in cacheKey)
The timeValue will be the same for the entire duration_unit (hour in above example)

TTL

Every key value pair will have a TTL of 2*duration_units.
For example : If the duration unit is `minutes`, TTL of that key would be `2 minutes`.

At every interval, we would be requiring the information of interval t1 and t2 (previous duration)
After time = 2 * t1 we will not read the key, hence it will be evicted.
Duration : 1 minute

5	3	4
t1	t2	t3

At t3, we will only be needing to read the value of t2, to see if the rate limit has reached or not.
Value at t1 is useless, so it can be evicted.
t2 - t1 = t3 - t2 (duration will be same) -> 1 * time_unit

Contracts

Description	API	Request	Response

Registers Service to rate limiter	rpc RegisterService(RegisterServiceRequest) ResgiterServiceResponse	message RegisterServiceRequest {
		    int32 serviceId = 1;
		    Rule rule = 2;
		}
		message Rule {
		    string field = 1;
			message RegisterServiceResponse {
			    google.rpc.Status status = 1;
			    string error = 2;
			}

		Rate rate = 2;
		string errorMessage = 3;
		}
		
		message Rate {
		int requestsPerUnit=1;
		string unit = 2;
		}	

Checks if rate limit has reached or not	rpc ShouldForwardRequest(ShouldForwardRequestRequest) ShouldForwardRequestResponse	message ShouldForwardRequestRequest {
		int32 serviceId = 1;
		string request = 2;
		}
			message ShouldForwardRequestResponse {
			bool shouldForward = 1;
			}

Technical Specifications

- Language : goLang
- dataStore : dynamoDB
- Cache : Redis