Project Report

Rate Limiter

Overview

- Member: Kien Mai
- GitHub: https://github.com/maingockien01/Simple-Rate-Limitter
- Presentation: https://youtu.be/C3jdTIVZJno

Introduction

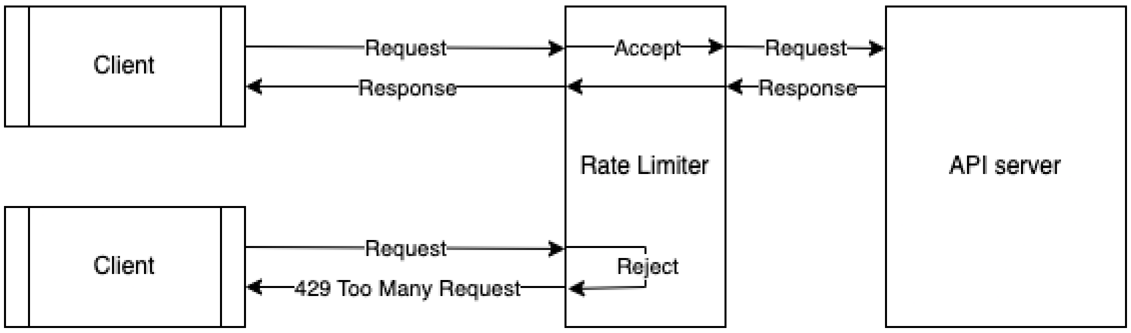
In a network system, application gateway is used to manage the traffic between clients and applications running on backend servers in the context of HTTP protocol. Application gateway has the responsibility to protect the security, the performance, and other requirements of web servers on OSI layer of application. One of the common security method carried out by application gateways is rate limiting. Application gateways would limit the number of requests allowed to be sent from clients over a period of time. Rate limiting is used to maintain the performance of web server by keeping the workload roughly constant or not getting overwhelming. Another benefits of having a rate limiter is to protect servers from Denial of Service attack that attackers utilize a large number of requests to servers.

Architecture

Where to put the rate limiter

There are some options for keeping the rate limiters:

- Client-side implementation: This is possible by enforcing the rules on client though programming. However, it is not safe and it is easy for attackers to bypass this implementation. Moreover, client is not something we have control over.
- Server-side implementation: Rate limiters are implemented as a part of the application server. This approach provides simple and monolith solutions. However, the problem is the application still receive the request before the requests are filtered by rate limiters. Hence, it still face up the overload of incoming traffics and it could congest the traffic to the server.
- Distributed middleware: Another approach is having rate limiter as a reserve proxy server filtering incoming requests before transferring requests to backend servers. This approach allows to separating the traffic coming to rate limiter servers and backend servers. Furthermore, for the difference in requirements, rate limiters which could receive a large incoming requests and has light logic work compared to backend server could utilize cheaper infrastructure with high internet bandwidth. And we can keep the expensive infrastructure for backend servers which would require more logic works there. However, the cons of this approach are the complexity of having another services in the system and the resources to create and maintain an extra service.



One of the most popular architectures used in the industry nowadays is cloud microservices and rate limiters or application gateways would usually be implemented as a microservice. Hence, I decided to implement rate limiter as a microservice separated from back-end servers. Another reason is I do not have the backend servers and implementing with backend server option would require taking backend server architecture into the consideration while having separated rate limiter service would be more flexible.

Rate limiting algorithm

There are different algorithms with pros and cons for rate limiting:

- Token Bucket
- Leaking Bucket
- Fixed Window Counter
- Sliding Window Log
- Sliding Window Counter

In this project, I choose token bucket algorithm for the implementation due the simplicity of it.

In fact, the algorithm is quite common and popular because it is simple and well understood.

The algorithm:

- A token bucket is the representation of request manager for each user on a certain API path. The token bucket would have a limited capacity "MaxTokens" attribute, and each token would have a rate for refilling the token if the bucket is not full. Once the bucket has full of token, it would stop refilling.
- Each request is a token. When a request comes, the token in the corresponding bucket would be deducted. The request is accepted if there is available token in the bucket. Otherwise, it will be rejected.
- Parameters for the algorithm:
  - MaxTokens: the maximum number of tokens allowed in the bucket.
  - Refill Rate: number of tokens are filled in a certain period of time, usually a second.

Setting parameters for algorithm

Maximizing the performance of the rate limiter while maintaining the workload of API server would require dynamical or frequently manual tuning parameters. If the rate or the max tokens are too small, too many requests are rejected and API server does not utilize all possible resources.

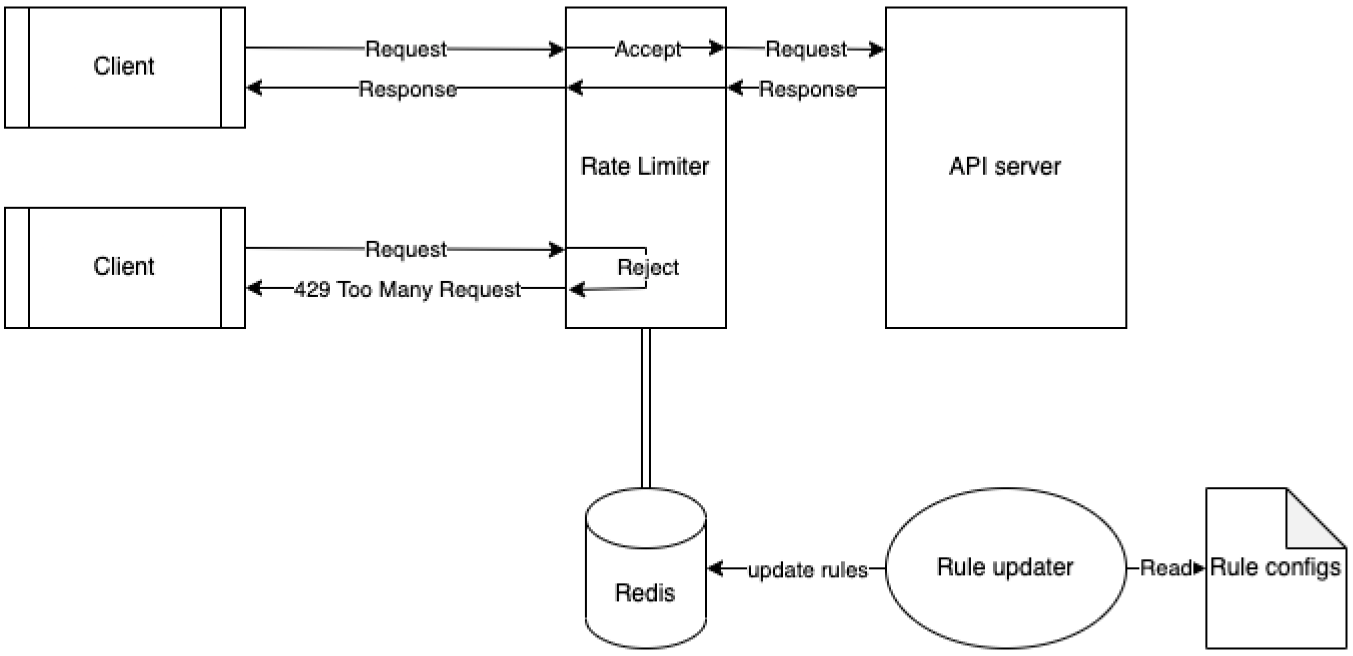
Inspired by how Lyft open-source rate limiting component sets rules, the rate limiter would have a module named Ruler in responsibility of managing rules. Rules can be defined in json format and stored in files to read.

Storage

As the algorithm proceeds, we need to store the data of the bucket. We can store data in relational database like MySQL but it would be slow if we keep reading it frequently. Instead, using Redis would be more efficient for caching data for quick read and change and usually we store data only in a limited session.

Hence, we can store bucket data in Redis as well as bucket rules aka parameters in Redis as well.

Design

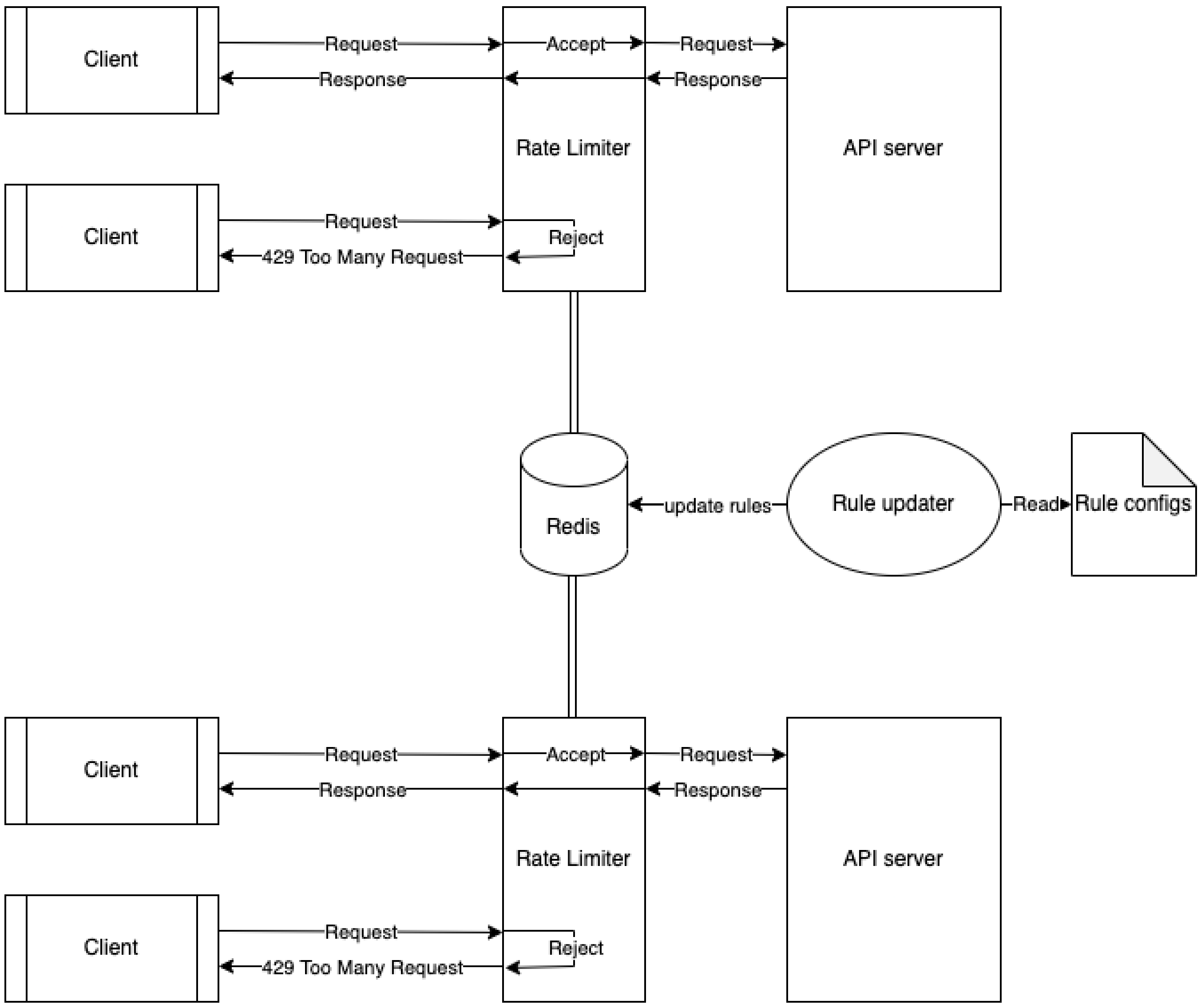


When clients send request to API server, the request first go through the rate limiter. Rate limiter get bucket data from Redis and check if there is available token for new requests. If yes, the request is accepted and passed to API server. If no, rate limiter responses back to client with status code 429 Too Many Response with Header X-Retry-After indicating time client should wait before sending another request.

Meanwhile, rules are written in config file and ruler which is a parallel process read files and update rules to Redis every an amount of time. And rate limiter repeatedly fetch rules from Redis and update rules for buckets after a certain time as well.

Expanding to multiple rate limiters

In fact, in the distributed environment, there would be multiple instances of API backend servers. Each API backend server would be connected with a rate limiters.



The system would be synchronized through Redis as a centralized data storage. However, there is race condition when we use Redis for parallel servers. For example, if there is only 1 available token left, rate limiter 1 and rate limiter 2 receive same request for the last token left in Redis. Both of them would read the data from Redis same time that there is one token left for the incoming request. Then, both of them accept the request while only one of them would accept the request.

To solve this, we can implement Lua script with atomic transaction or using a distributed lock to lock the data when a rate limiter read and write data. The lock approach would be slower but much more simpler. For the project, I choose the lock approach as the solution.

Detailed Infrastructure

I used Docker to set up the infrastructure. Each rate limiter would be in a Docker container and it is the same for back-end server. The Redis is running in a container while the Ruler worker is also run in another container. All rate limiter containers and Ruler workers share the same set up folder, hence the rules are all read from the same config file on initiating the services.

Please refer to Dockerfile and docker-compose.yml files in the GitHub link for the details.

Detailed Implementation

The rate limiter server is actually a reserve proxy that take requests from clients, pass requests to api serviers then recieve response and return it back to clients.

The rate limiter service contains 3 modules:

- Server: in charge of implementing reserve proxy server
- RateLimiter: in charge of implementing rate limiting algorithm
- Ruler: in charge of managing rules from files and from Redis.

The back-end server is **whoami** servers. The Docker image of the backend server is from Traefik lab. The project is written in Golang.

References

1. Rate-limiting strategies and techniques (https://cloud.google.com/solutions/rate-limiting-strategies-techniques)
2. Google docs usage limits (https://developers.google.com/docs/api/limits)
3. Throttle API requests for better throughput (https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html)
4. Better Rate Limiting With Redis Sorted Sets (https://engineering.classdojo.com/blog/2015/02/06/rolling-rate-limiter/)
5. System Design - Rate limiter and Data modelling (https://medium.com/@saisandeepmopuri/system-design-rate-limiter-and-data-modelling-9304b0d18250)
6. How we built rate limiting capable of scaling to millions of domains (https://blog.cloudflare.com/counting-things-a-lot-of-different-things/)
7. Redis website (https://redis.io/)
8. Lyft rate limiting (https://github.com/lyft/ratelimit)
9. Scaling your API with rate limiters (https://gist.github.com/ptarjan/e38f45f2dfe601419ca3af937fff574d#request-rate-limiter)
10. OSI model (https://en.wikipedia.org/wiki/OSI_model#Layer_architecture)
11. ByteByteGo System Design Blog (https://blog.bytebytego.com/)