Rate limiting, API gateway & Load balancing

Sanket Saxena · Follow
8 min read · Jul 5, 2023

Rate Limiting

Rate limiting is an essential mechanism in server management, regulating how often a client can make a request in a certain timeframe. It's vital for preventing server abuse and ensuring equitable resource allocation. Now, let's dig into the details of each rate-limiting algorithm, provide some examples, and evaluate their pros and cons.

Token Bucket

The Token Bucket algorithm visualizes rate limiting as a bucket that continuously gets filled with tokens at a constant rate. For every request made, one token is removed. If the bucket is empty, new requests get denied until more tokens accumulate.

Example: Consider a token bucket algorithm that adds 10 tokens per minute and the bucket's capacity is 60 tokens. If a client makes 60 requests at once, all requests are fulfilled. However, if they make another request within that minute, it will be denied because the bucket is out of tokens.

Pros:

- Accommodates for bursts in traffic.
- Simplicity makes it efficient in terms of resource usage.

Cons:

- After a burst, there can be a noticeable pause as the client must wait for tokens to replenish.

Leaky Bucket

The Leaky Bucket algorithm discards any incoming requests if its buffer (bucket) is full, effectively producing output traffic at a constant rate.

Example: Let's assume a leaky bucket with a leak rate of 10 requests per minute. If a client makes 60 requests instantly, only the first 10 requests are processed, and the rest are discarded. New requests are processed only as the leak (request processing) creates room in the bucket.

Pros:

- Smoothens the output rate by discarding excess requests.
- Ensures constant processing rate, preventing server overload.

Cons:

- Potential loss of requests when the bucket is full, which may not be acceptable for all applications.

Sliding Window Log

The Sliding Window Log algorithm maintains a log of all requests within the current window, providing highly precise rate limiting. The 'window' slides over time, hence the name.

Example: Consider a fixed window counter with a limit of 60 requests per minute. If a client makes 60 requests in the first 30 seconds, any requests in the remaining 30 seconds of that minute will be denied.

Pros:

- Simpler to implement and less resource-intensive.
- Capable of handling traffic bursts up to the window limit.

Cons:

- It can unfairly penalize clients if their requests coincide with the window reset, creating potential for double the rate limit to be processed at the boundaries of the windows.

Fixed Window Log

The Fixed Window Log is a variant of the Fixed Window Counter. However, instead of just counting requests, it logs each one, providing more precise control.

Example: Like the Fixed Window Counter, if we set a limit of 60 requests per minute, any client making 60 requests within the first 30 seconds will be denied for the remaining 30 seconds.

Pros:

- Offers more precise control than the Fixed Window Counter as it keeps a log of all requests.

Cons:

- More resource-intensive as it requires storing all logs within the window

- More resource-intensive as it requires storing all logs within the window.
- Similar to the Fixed Window Counter, it may allow burst traffic around the window boundaries.

Understanding Rate Limiting Parameters

Rate limiting depends on a few key parameters: identifiers, time, and limit.

- **Identifiers**: Typically, these are elements like IP address, user ID, or user tier, used to identify the source of the requests. For instance, you might apply different rate limits for different user tiers, such as regular, premium, and VIP. Differentiating rate limits based on identifiers allows the system to be more flexible and adaptable.
- **Time**: This parameter sets the window within which a certain number of requests are allowed. It could be per second, per minute, or any other time unit suitable for the system. For example, an API could allow 1000 requests per hour from a single user ID.
- **Limit**: This is the maximum number of allowed requests within the time window from an identifier. It's the primary lever to adjust based on system load, business decisions, or usage patterns.

Depending on these parameters, the system responds in a few common ways to excess requests:

- **Blocking**: The system denies excess requests outright, often returning an HTTP 429 (Too Many Requests) status code. This is the simplest approach but could frustrate users if they're unaware of the limit or if it's set too low.
- **Throttling**: The system artificially slows down the rate of processing excess requests. For instance, if the limit is reached within a minute, the system might only process a request from the same user every subsequent second, while the others wait in a queue. This slows down the perceived speed but doesn't deny service outright.

- **Shaping**: The system lowers the priority of excess requests, attending to them only when no other higher-priority requests are in the queue. It's a gentler form of throttling that helps maintain service availability but at slower speeds.

Application Level vs. Infrastructure Level Rate Limiting

Rate limiting can be applied at both the application level and the infrastructure level. At the application level, rate limits are usually tied to business logic. For instance, you could apply higher limits to premium users or users who have completed certain in-app tasks.

At the infrastructure level, rate limiting is often a feature of CDN (Content Delivery Network), reverse proxies, or API gateways. These systems manage rate limiting at a more general level, such as limiting the number of requests to certain API endpoints from each IP address.

Redis in Rate Limiting and Atomic Operations

A popular tool for managing rate limits is Redis, an in-memory data structure store that's excellent for high-speed operations. It's often used for caching, which is a crucial aspect of efficient rate limiting. Redis supports atomic operations and Lua scripting, allowing complex, high-speed operations to be performed in a single step, preventing race conditions.

An atomic operation in Redis is an operation that's fully completed or not executed at all, even if multiple clients are making requests simultaneously. For example, in a rate limiting scenario, a Redis atomic operation could decrease a counter and check if the new value exceeds the limit in a single step.

Handling Failures: Fail Open vs. Fail Close

Failures in rate limiting can be handled in two ways: fail open or fail close.

- **Fail Open**: When a system is set to "fail open," it allows traffic through in the event of a failure. This setting is typically used in high-availability applications, where blocking all traffic might be more detrimental than processing potentially rate-limited requests.
- **Fail Close**: Conversely, when set to "fail close," the system blocks all traffic during a failure. This setting is often used for resource-intensive operations where letting through too many requests could be disastrous. For example, if a rate limit is designed to protect a database from too many write operations, a fail close strategy would be best.

These strategies also extend to business decisions. For instance, you might choose to set premium users to "fail open" to ensure they always have access, even during times of high traffic or system failure.

API Gateway

The API Gateway is a server that acts as an entry point into the application from the client side. It aggregates various service requests and manages them effectively.

API gateways perform various functions such as:

- **Validation**: They ensure the data sent in requests is correct.
- **Allow block list**: API Gateways can control access by having a list of allowed or blocked IPs.
- **Authentication and Authorization**: Gateways often manage user authentication and determine what resources a user can access.
- **Rate Limiting**: As described above, they manage the rate at which users can make requests.
- **Dynamic Routing**: API Gateways can route requests to different services based on the request type and other factors.

- **Service Discovery:** Gateways find the network locations of service instances automatically.
- **SSL Offloading**: They can decrypt SSL traffic to reduce the processing burden on internal network resources.
- **Error Handling**: API Gateways handle errors and provide appropriate responses.
- **Logging and Monitoring:** They record logs and monitor traffic, helping in debugging and system optimization.
- **Circuit Breaking**: They stop cascading failures by halting traffic to a faulty service.
- **Caching**: To improve performance, gateways can cache responses.

Load Balancing

Load balancing is the process of distributing network traffic across multiple servers to ensure no single server bears too much load. This improves the responsiveness and availability of applications. Different algorithms determine how the load balancer distributes traffic:

Round Robin

Round Robin algorithm circulates the incoming requests sequentially to all servers in the pool.

Example: If we have 3 servers (A, B, C), the first request goes to A, the second to B, the third to C, the fourth to A again, and so on.

Pros:

- Simple and straightforward to implement.
- Ensures fair distribution of requests when all servers are equal in processing capacity.

Cons:

- Doesn't account for the servers' current load or capacity, potentially leading to imbalances in distribution if the servers have different processing abilities.

Sticky Round Robin

Sticky Round Robin is a variant of the Round Robin where the client's first request goes to a randomly selected server, and subsequent requests from the same client stick to the same server.

Example: If a client's first request is served by Server A, all future requests from this client will be directed to Server A unless it goes down or becomes unavailable.

Pros:

- Useful when maintaining session state is critical.
- Reduces the need for session replication across servers.

Cons:

- If a single server holds the sessions for many high-traffic users, it can lead to an imbalance in load distribution.

Weighted Round Robin

Weighted Round Robin allows assignment of different traffic-handling capacities to each server, distributing requests based on these weights.

Example: If Server A has a weight of 3, and Server B has a weight of 1, Server A will handle three requests for each request handled by Server B.

Pros:

- Better distribution when servers have different processing capacities.
- Can handle more complex network infrastructures.

Cons:

- Requires a clear understanding and configuration of server capacities.

Least Connection

The Least Connection algorithm directs traffic to the server with the fewest active connections. It's often used when there are significant and potentially long-lasting differences in the load produced by each connection.

Example: If Server A has 20 active connections and Server B has 10, the new incoming request will be routed to Server B.

Pros:

- More dynamic load balancing, especially useful when the sessions are long-lasting.
- Prevents overloading a single server with many heavy sessions.

Cons:

- Higher overhead due to tracking connection counts.

Least Response Time

The Least Response Time algorithm directs traffic to the server with the fewest active connections and the lowest average response time.

Example: Between two servers A and B, if both have an equal number of active connections but Server A has a lower average response time, the new request will be routed to Server A.

Pros:

- It considers both server load and performance, ensuring an efficient distribution.
- Improves user experience by decreasing wait time.

Cons:

- More complex to implement due to the need for tracking both connection counts and response times.

Remember, the choice of load balancing method will depend on the specifics of your setup and the nature of the applications you're managing.

To sum up, rate limiting, API gateways, and load balancing are essential components of any large-scale web service. Understanding their underlying mechanisms and knowing how to use them effectively is crucial to maintain a robust and efficient infrastructure.

Backend

Written by Sanket Saxena

Follow

137 Followers

I love writing about software engineering and all the cool things I learn along the way.

More from Sanket Saxena

Sanket Saxena

System Design of Collaborative Editing Tool

The widespread shift towards remote and collaborative work necessitates advanced

Jun 17, 2023 4

Sanket Saxena

Database—Part Three

Distributed Locking: A Detailed Overview

Jul 10, 2023 2 1

Sanket Saxena

System Design of Netflix

Netflix is a global entertainment giant with over 200 million subscribers. To comprehend

Sanket Saxena

Concurrency in Python

This article delves into advanced techniques and best practices for leveraging Python's

Jun 20, 2023 14

Jun 15 4

See all from Sanket Saxena

Recommended from Medium

Priya Patidar in The Developer's Diary

Introduction to API Rate Limiting: Understanding the Basics and Its

Introduction

Jan 28 311 4

Fahim Fahad in AWS Tip

API Gateway Custom Authorizer

In this article I will demonstrate how we can configure custom authorizer in API Gateway

May 7 5

Lists

Coding & Development

11 stories · 827 saves

Stories to Help You Grow as a Software Developer

19 stories · 1385 saves

Rabinarayan Patra

Why 1==1 is true but 128==128 is false in Java

Ever wondered why comparing 1==1 returns true, but 128==128 returns false in Java? Let's

Sep 11 663 17

Dylan Smith in Javarevisited

Interview: Why is Redis so fast even though it is single-threaded?

My articles are open to everyone; non-member readers can read the full article by

Aug 31 1.1K 6

Master Spring Ter

Implementing Rate Limiting in Spring Boot with Bucket4j

As web applications grow and handle more traffic, ensuring their reliability and

Jun 16 10

Pooja Modi

API Rate limiter: A savior!

The story is from 2022 when I was moving ahead in my career from my first company.

May 20

See more recommendations