Rate Limiter For The Real World

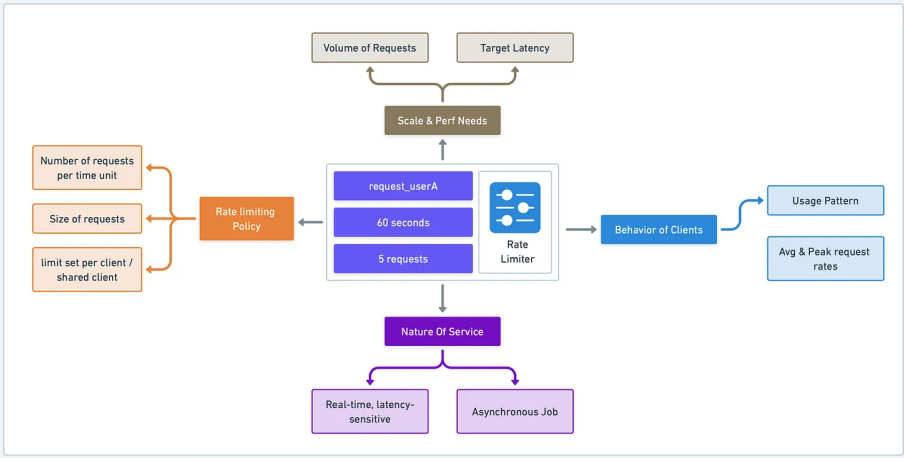
ALEX XU
JUN 8, 2023 · PAID

In the previous article, we discussed the fundamentals of rate limiting and explored several popular rate limiting algorithms. We'll continue by exploring how to build some real-world rate limiters.

There are many factors to consider when developing a rate limiter for the real world. For example, in many use cases, a rate limiter is in the most critical path of an application. It is important in those situations to think through all the failure scenarios and consider the performance implications of placing it in such a critical path.

Gathering Requirements and Key Considerations

Before diving headfirst into building our rate limiter, it is important to understand our goals. This starts with gathering requirements and considering key design factors. The requirements can vary based on the nature of our service and the use cases we're trying to support.



First, we should consider the nature of our service. Is it a real-time, latency-sensitive application or an asynchronous job where accuracy and reliability are more important than real-time performance? This can also guide our decisions around how we handle rate limit violations. Should our system immediately reject any further requests, or should it queue them and process them as capacity becomes available?

Next, we should consider the behavior of our clients. What are their average and peak request rates? Are their usage patterns predictable, or will there be significant spikes in traffic? This analysis is useful for creating rate limiting rules that protect our system without hindering legitimate users.

Then, consider the scale and performance needs. We need to understand the volume of requests we anticipate and set the target latency of our system. Is the system serving millions of customers making billions of requests, or are we dealing with a smaller number? Is latency critical, or can we sacrifice some speed for enhanced reliability?

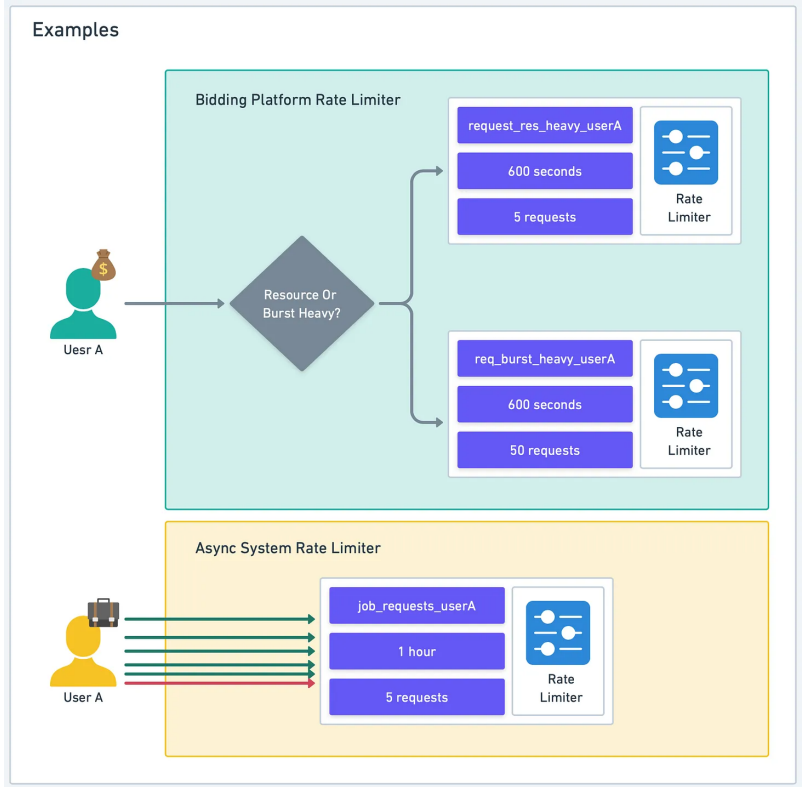
Our rate limiting policy also needs careful consideration. Are we limiting based on the number of requests per unit of time, size of requests, or some other criteria? Is the limit set per client, or shared across all clients? Are there different tiers of clients with different limits? Is the policy strict, or does it allow for bursting?

Some rate limiters will have persistence requirements. For a long rate limiting window, how do we persist the rate limit states like counters for long-term tracking? This requirement might be especially important in cases where latency is less critical than long-term accuracy, such as in an asynchronous job processing system.

Let's consider some examples to better understand these points.

For a real-time service like a bidding platform where latency could be highly critical, we would need a rate limiter that doesn't add significant latency to the overall request time. We will likely need to make a tradeoff between accuracy and resource consumption. A more precise rate limiter might use more memory and CPU, while a simpler one might allow occasional bursts of requests.

On the other hand, in an asynchronous job processing system that processes large batches of data analysis jobs, our rate limiter doesn't need to enforce the limit in real-time, but must ensure that the total number of jobs submitted by each user doesn't exceed the daily limit. In such cases, storing the rate limiting states durably, for instance in a database, might be crucial.



Rate Limiter Architecture

The main architectural question is: where should we place the rate limiter in our application stack?

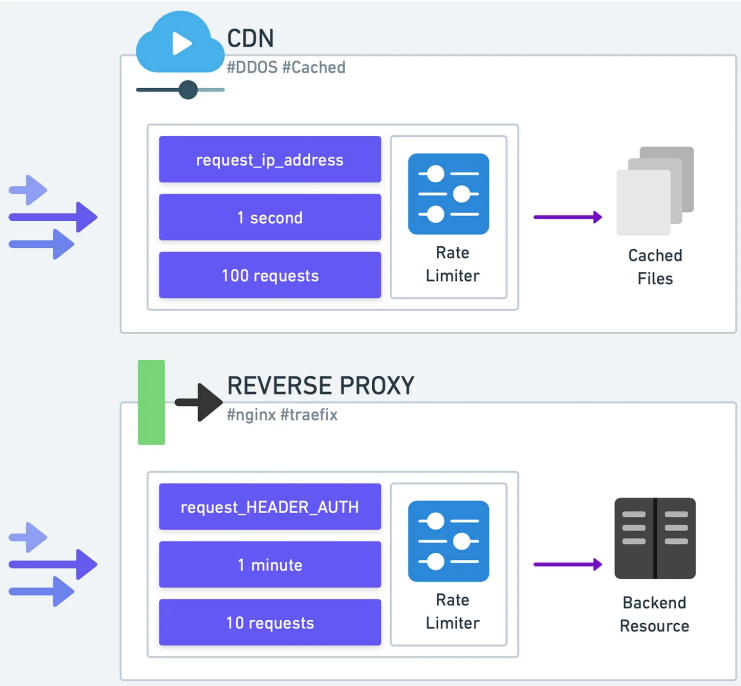
This largely depends on the configuration of the application stack. Let's examine this in more detail.

The location where we integrate our rate limiter into our application stack matters significantly. This decision will depend on the rate limiting requirements we gathered in the previous section.

In the case of typical web-based API-serving applications, we have several layers where rate limiters could potentially be placed.

CDN and Reverse Proxy

The outermost layer is the Content Delivery Network (CDN) which also functions as our application's reverse proxy. Using a CDN to front an API service is becoming an increasingly prevalent design for production environments. For example, Cloudflare, a well-known CDN, offers some rate limiting features based on standard web request headers like IP address or other common HTTP headers.



If a CDN is already a part of the application stack, it's a good initial defense against basic abuse. More sophisticated rate limiters can be deployed closer to the application backend to address any remaining rate limiting needs.

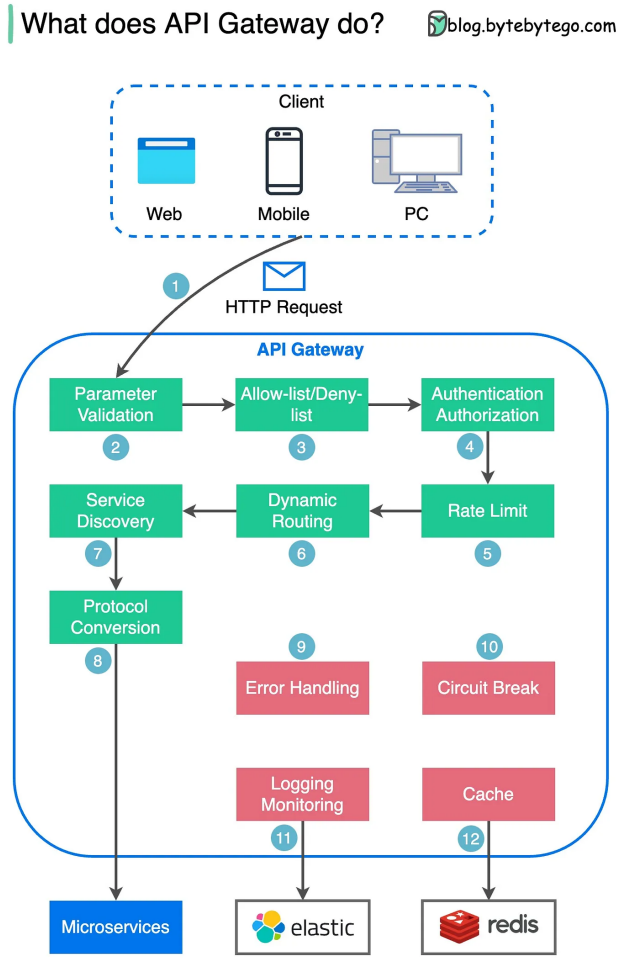
Some production deployments maintain their own reverse proxy, rather than a CDN. Many of these reverse proxies offer rate limiting plugins. Nginx, for example, can limit connections, request rates, or bandwidth usage based on IP address or other variables such as HTTP headers. Traefik is another example of a reverse proxy, popular within the Kubernetes ecosystem, that comes with rate limiting capabilities.

If a reverse proxy supports the desired rate limiting algorithms, it is a suitable location for a basic rate limiter.

However, be aware that large-scale deployments usually involve a cluster of reverse proxy nodes to handle the large volume of requests. This results in rate limiting states distributed across multiple nodes. It can lead to inaccuracies unless the states are synchronized. We'll discuss this complex distributed system issue in more detail later.

API Gateway

Moving deeper into the application stack, some deployments utilize an API Gateway to manage incoming traffic. This layer can host a basic rate limiter, provided the API Gateway supports it. This allows for control over individual routes and lets us apply different rate limiting rules for different endpoints.



[Amazon API Gateway] is an example of this. It handles rate limiting at scale. We don't need to worry about managing rate limiting states across nodes, as would be required with our own cluster of reverse proxy servers. A potential downside is that the rate limiting control might not be as fine-grained as we would like.

Application Framework and Middleware

If our rate limiting needs require more fine-grained identification of the resource to limit, we may need to place the rate limiter closer to the application logic. For example, if user specific attributes like subscription type need limiting, we'll need to implement the rate limiter at this level.

In some cases, the application framework might provide rate limiting functionality via middleware or a plugin. Like in previous cases, if these functions meet our needs, this would be a suitable place for rate limiting.

This method allows for rate limiting integration within our application code as middleware. It offers customization for different use-cases and enhances visibility, but it also adds complexity to our application code and could affect performance.

Application

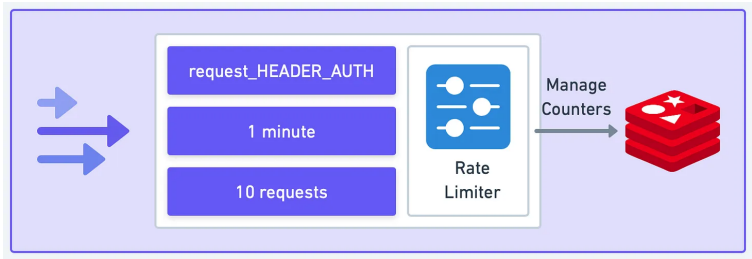
Finally, if necessary, we could incorporate rate limiting logic directly in the application code. In some instances, the rate limiting requirements are so specific that this is the only feasible option.

This offers the highest degree of control and visibility but introduces complexity and potential tight coupling between the rate limiting and business logic.

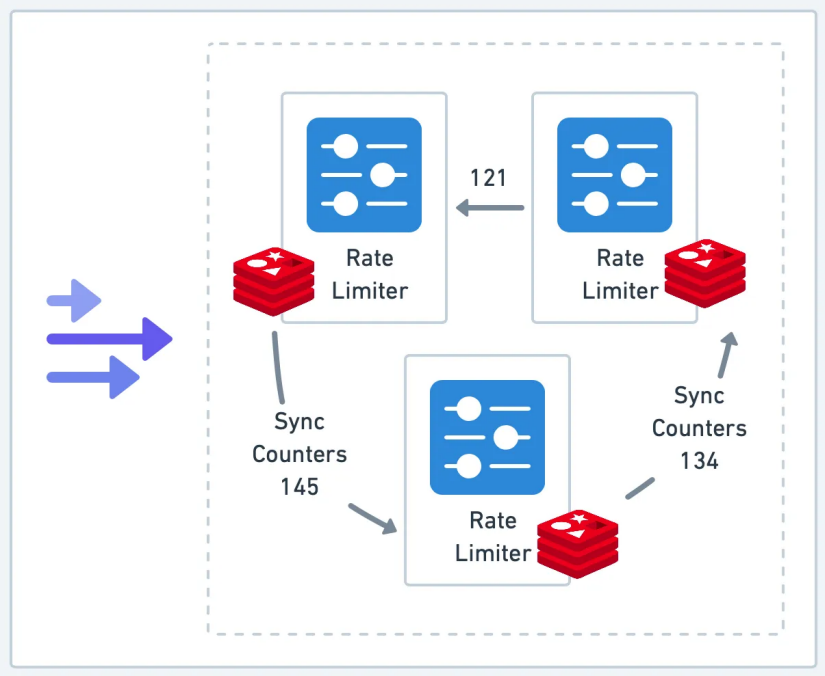
Like before, when operating at scale, all issues related to sharing rate limiting states across application nodes are relevant.

Rate Limiting States

Another significant architectural decision is where to store rate limiting states, such as counters. For a low-scale, simple rate limiter, keeping the states entirely in the rate limiter's memory might be sufficient.



However, this is likely the exception rather than the rule. In most production environments, regardless of the rate limiter's location in the application stack, there will likely be multiple rate limiter instances to handle the load, with rate limiting states distributed across nodes.



We've frequently referred to this as a challenge. Let's dive into some specifics to illustrate this point.

When rate limiters are distributed across a system, it presents a problem as each instance may not have a complete view of all incoming requests. This can lead to inaccurate rate limiting.

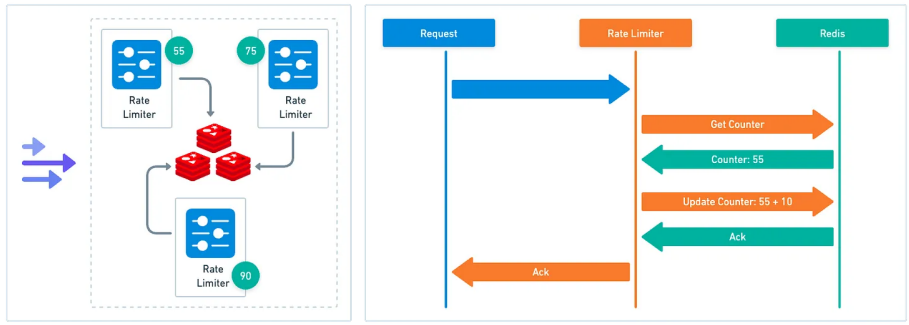
For example, let's assume we have a cluster of reverse proxies, each running its own rate limiter instance. Any of these proxies could handle an incoming request. This leads to distributed and isolated rate limiting states. A user might potentially exceed their rate limit if their requests are handled by multiple proxies.

In some cases, the inaccuracies might be acceptable. Referring back to our initial discussion on requirements, if our objective is to provide a basic level of protection, a simple solution allowing each instance to maintain its own states might be adequate.

However, if maintaining accurate rate limiting is a core requirement - for example, if we want to ensure fair usage of resources across all users or we have to adhere to strict rate limit policies for compliance reasons - we will need to consider more sophisticated strategies.

Centralized State Storage

One such strategy is centralized state storage. Here, instead of each rate limiter instance managing its own states, all instances interact with a central storage system to read and update states. This method does, however, come with many tradeoffs.



Implementing a centralized state storage system can add latency and become a bottleneck, especially in high throughput applications. It's also challenging to implement this correctly in a performant manner. Race conditions can arise when multiple processes attempt to update the state simultaneously. This could lead to inaccuracies. These challenges and their solutions are worth exploring in detail.

Handling Race Conditions

In a scenario with a centralized datastore and high concurrent rate limit request volumes, race conditions can become a significant problem. This usually happens when we adopt a pattern where we first retrieve the current rate limit counter, increment it, and then write it back into the datastore. The race condition here is that in the time it takes to complete a read-increment-write cycle, more requests can

come through. These requests try to write the counter back but with an outdated, lower counter value, leading to potential rate limiting bypass.

We could consider locking the key for the rate limit identifier to prevent any other instances from accessing or modifying the counter. However, this method can lead to a serious performance bottleneck and doesn't scale well.

A more effective strategy involves adopting an approach that uses atomic operations. These operations allow for fast counter value incrementing and checking without compromising atomicity. It enables efficient counter management without risking race conditions. This approach is well-supported in data stores commonly used for rate limiting, like Redis, which we will discuss later.

Performance Optimization

Utilizing a centralized data store for tracking rate limit counters invariably adds latency to each request, even with a fast datastore like Redis. This is where in-memory copies of the global states can help. Local, in-memory cached states can significantly reduce latency.

To enable this, we could slightly loosen the rate limit accuracy and use an eventually consistent model. Under this model, each node periodically executes a data synchronization push with the centralized data store. During each push, every node atomically updates the counter for each identifier and time window to the datastore. The node can then fetch the updated values to refresh its in-memory version.

The rate of these pushes should be adjustable. Shorter synchronization intervals reduce data divergence when distributing traffic across multiple nodes in the cluster. On the other hand, longer synchronization intervals reduce read/write pressure on the datastore and minimize overhead on each node to fetch newly synchronized values.

Nginx uses this mechanism to synchronize the rate limit states across nodes.

Utilizing Redis for Rate Limiting

Redis is a high-performance in-memory data structure store. It is a popular tool for implementing rate limiters. Many rate limiter plugins for various reverse proxies and API gateways support using Redis as a backing store as well.

Redis is an attractive option because it is easy to use and highly scalable. Redis supports various data structures and commands that can help in implementing different types of rate limiting algorithms, such as Sliding Window Log, Fixed Window Count, or Token Bucket. It supports Lua scripting as the means to construct sophisticated rate limiting algorithms that might otherwise be challenging to implement.

There are many examples of implementing various rate limiting algorithms in Redis. This one from Stripe is a good example.

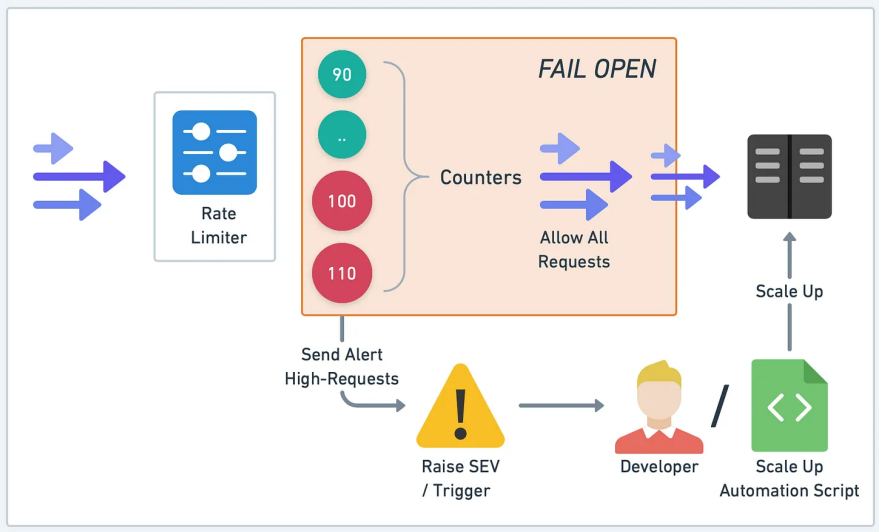
Redis is also highly scalable. A single Redis instance can manage the workload of most medium-sized applications. To scale beyond a single node, there are known techniques like sharding to distribute rate limiter identifiers across many nodes.

However, like any tool, Redis isn't a one-size-fits-all solution. Redis is primarily an in-memory data store. It provides basic persistence capabilities, but it doesn’t provide the same durability guarantee as a conventional database. It's also important to consider the costs and complexities of managing a Redis cluster. However, if the deployment already employs Redis for other functionalities, utilizing it for rate limiting would be a logical choice.

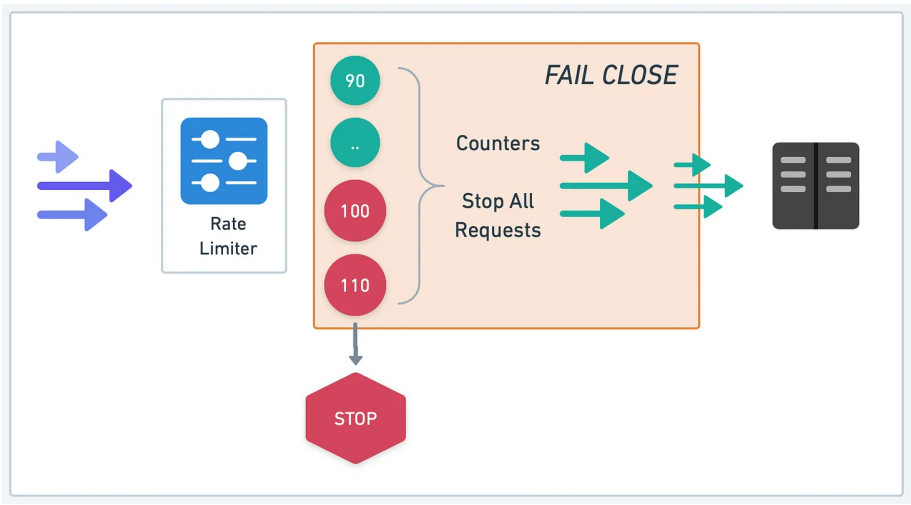
Handling Failures

Even with the most robust rate limiters, we can't entirely avoid failures. We incorporate strategies to manage them gracefully to minimize their impact on the system and the end-user experience.

Let's consider the two common strategies to handle rate limiter failures: "Fail open" and "Fail close". The "Fail open" strategy comes into play when the rate limiting service faces an unexpected error or is unavailable. In such a scenario, the rate limiter 'opens up', allowing all incoming requests to pass through. While this strategy ensures we don't reject any legitimate requests, it potentially risks letting in a flood of traffic that can overwhelm the downstream systems. This trade-off means that while we're maximizing availability, we might be jeopardizing the very reason we implemented rate limiting in the first place - protecting our system from being overwhelmed.



On the other hand, the "Fail close" strategy works differently. When a failure occurs, the rate limiter 'closes', rejecting all incoming requests. This strategy upholds the primary intent of the rate limiter by protecting the system from potential overload, even at the risk of denying legitimate requests. The trade-off here is that we might be sacrificing availability to ensure robust protection.

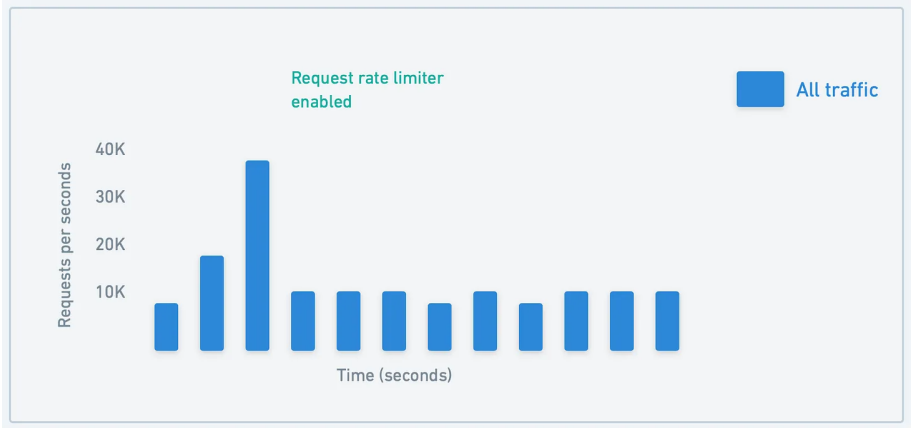


Choosing between "Fail open" and "Fail close" strategies often depends on the specific characteristics and requirements of the service. For systems where availability is the most important, "Fail open" might be the preferred strategy. For systems more concerned about ensuring strict compliance with rate limits and safeguarding downstream resources, "Fail close" could be more suitable.

However, these aren't the only strategies we can consider. Other alternatives might involve a hybrid strategy, where we combine aspects of "Fail open" and "Fail close". For example, we might choose to 'Fail open' for trusted clients but 'Fail close' for others. Or we might 'Fail open' but at a lower threshold, allowing traffic but at reduced rates.

Case Study

Stripe is a leading online payment processing platform. It offers a comprehensive rate limiting system to ensure the stability and reliability of its services. This case study explores how Stripe implements and uses rate limiting.



As Stripe's user base grew exponentially, they faced a challenge common to many successful APIs: how to maintain service reliability and availability despite sudden traffic spikes, which could potentially lead to service outages. They solved this by implementing various types of rate limiters.

Problem

These are the scenarios where Stripe was concerned about:

1. A single user causing a spike in traffic
2. A user with a misbehaving script unintentionally overloading the servers
3. Users sending too many low-priority requests, threatening to degrade the quality of high-priority traffic.
4. Internal system issues impacting their ability to service all regular traffic.

Solution

Stripe implemented four types of rate limiters for these problems.

First is the Request Rate Limiter. This limiter restricts each user to a certain number of requests per second. This was the most used limiter at Stripe and the most important one to prevent abuse. They implemented a feature that allows for brief bursting above the limit to accommodate for large-scale events, such as flash sales.

They also added the Concurrent Request Limiter. This limiter caps the number of API requests in progress at the same time. This was particularly useful for controlling resource-intensive endpoints.

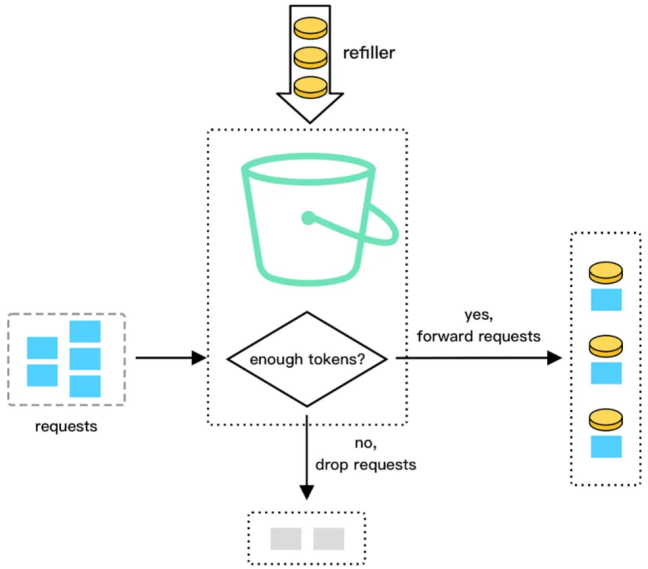
The last two were load shedders. The Fleet Usage Load Shedder ensures a specific percentage of the fleet is always available for crucial API requests. It reserves a fraction of the infrastructure for critical requests, and any non-critical request over their allowed allocation is rejected.

The Worker Utilization Load Shedder protects the set of workers that handle incoming requests from getting overwhelmed. It slowly starts shedding less-critical requests, starting with test mode traffic, to ensure workers remain available for critical requests.

Implementation

Stripe uses the token bucket algorithm for rate limiting. The algorithm uses a centralized bucket host where tokens are taken on each request, with new tokens

gradually dripped into the bucket. If the bucket is empty, the request is rejected.

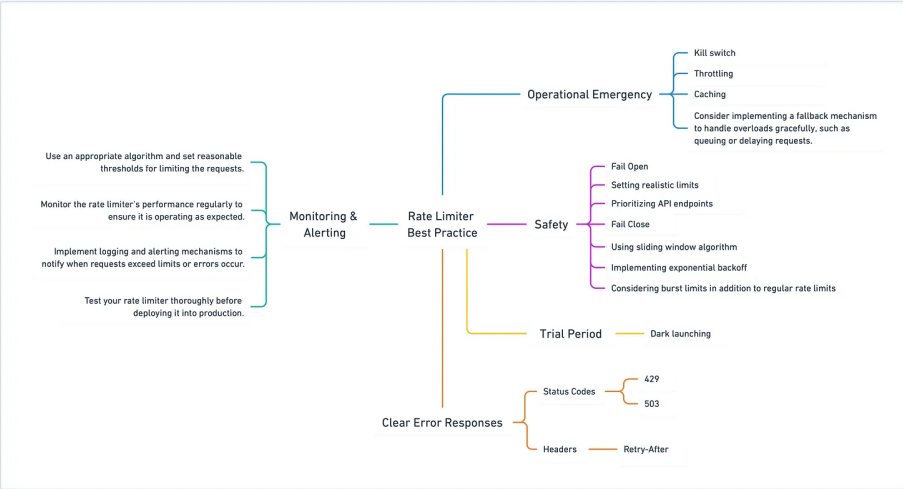


They implemented their rate limiters using Redis, which could either be self-hosted or through a managed service like Amazon's ElastiCache.

Implementing rate limiters proved to be an effective solution for Stripe to manage their growing user base and maintain the reliability of their API. They began with the Request Rate Limiter and gradually introduced other limiters over time as they scaled their operations. Their experience showcases the importance of proper rate limiting strategies and practices to maintain service reliability and prevent abuse.

Best Practices

Here are some best practices to consider when building a rate limiter for the real world.



Safety

A rate limiter is usually in the most critical path of an application. Make sure that if there were bugs in the rate limiting code, requests could continue to be processed. This means carefully catching errors at all levels so any bugs or operational errors would fail open. This includes handling any errors from the data store like Redis if the rate limiter relies on one.

One thing to note is that while fail-open should be a default, it could also expose the application to abuse. For some services, it might make sense to fail open for paying customers, while fail close for non-paying ones.

Operational Emergency

With the rate limiter in such a critical path, we should consider adding a kill switch to disable it if necessary. Feature flag is a fine mechanism to implement such a kill switch.

Monitoring and Alerting

It is important to have regular monitoring, alerts, and metrics to help understand when limiters are triggered and if adjustments are necessary.

Clear Error Responses

We should show the users clearly when the rate limit is hit. This gives the caller an opportunity to handle the rate limit errors correctly (instead of blindly retrying, and putting undue pressure on the service).

There is no industry standard for showing rate limit errors. For APIs, it is common to return HTTP 429 (Too Many Requests) or HTTP 503 (Service Unavailable). Additional information is also included in the response headers to inform the caller three pieces of information - rate limit ceiling, rate limit remaining, and rate limit reset time.

We should also include the standard Retry-After header to inform the caller when to retry.

Trial Period

Before fully enforcing rate limiters, dark launching can be useful to observe the potential impact. By observing the rate limiter without actually denying requests, we can gauge whether the right traffic is being flagged and fine-tune the thresholds.

Conclusion

Rate limiting is a fundamental tool in our toolkit when preparing our service for scalability. While it may not be necessary to implement all the different rate limiting strategies from day one, they can be gradually introduced as the need arises.

When implemented correctly, it provides an efficient mechanism to prevent abuse and ensure fair usage, all while maintaining a positive user experience. It's a balancing act, but with careful planning and attention to detail, it's entirely achievable.

Comments