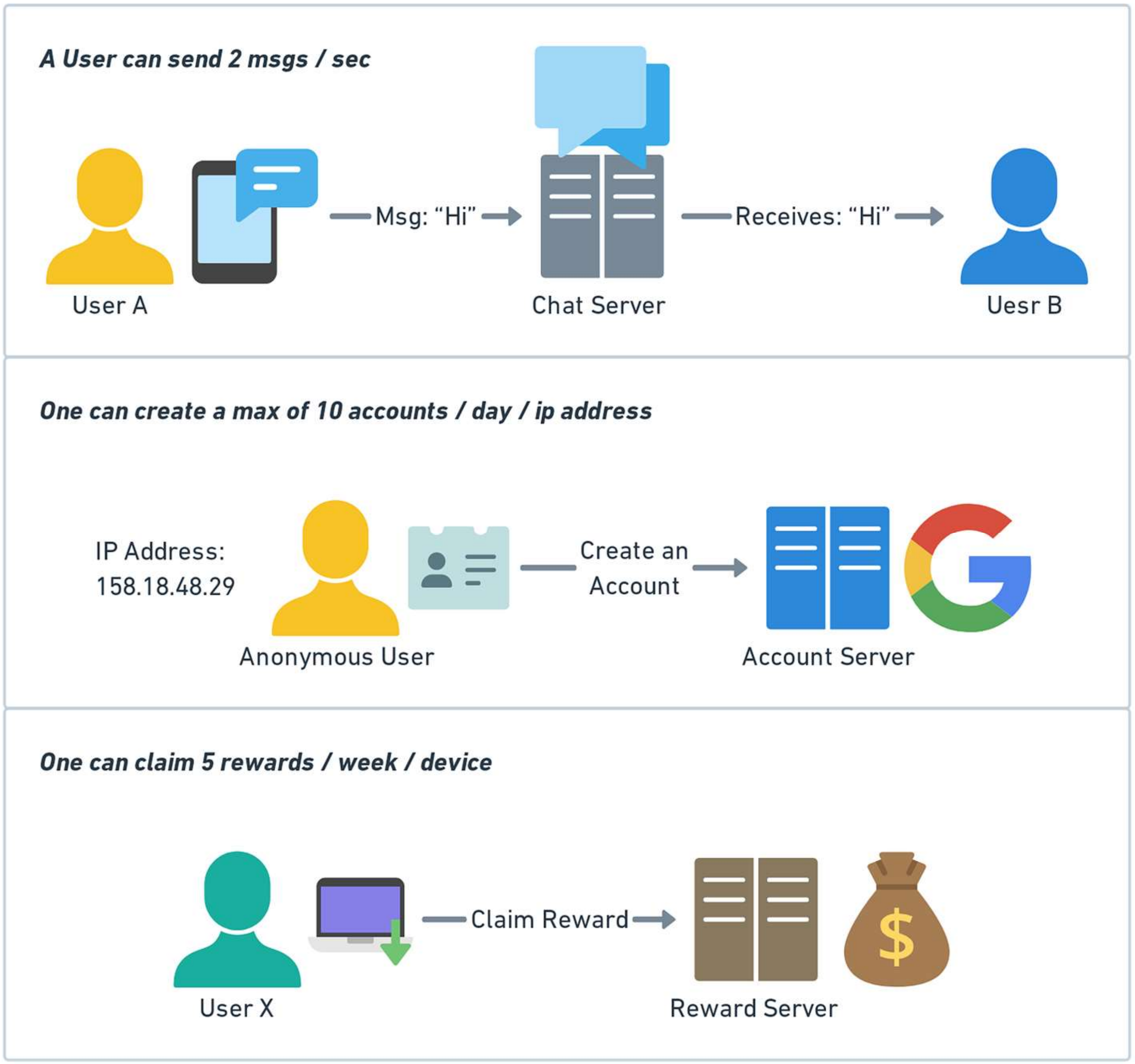
System Design Case Studies

Rate Limiter

Rate Limiter

- In a network system, a rate limiter is used to control the rate of traffic sent by a client or a service.
- In the HTTP world, a rate limiter limits the number of client requests allowed to be sent over a specified period. If the API request count exceeds the threshold defined by the rate limiter, all the excess calls are blocked. Here are a few examples:
  - A user can write no more than 2 posts per second.
  - You can create a maximum of 10 accounts per day from the same IP address.
  - You can claim rewards no more than 5 times per week from the same device.

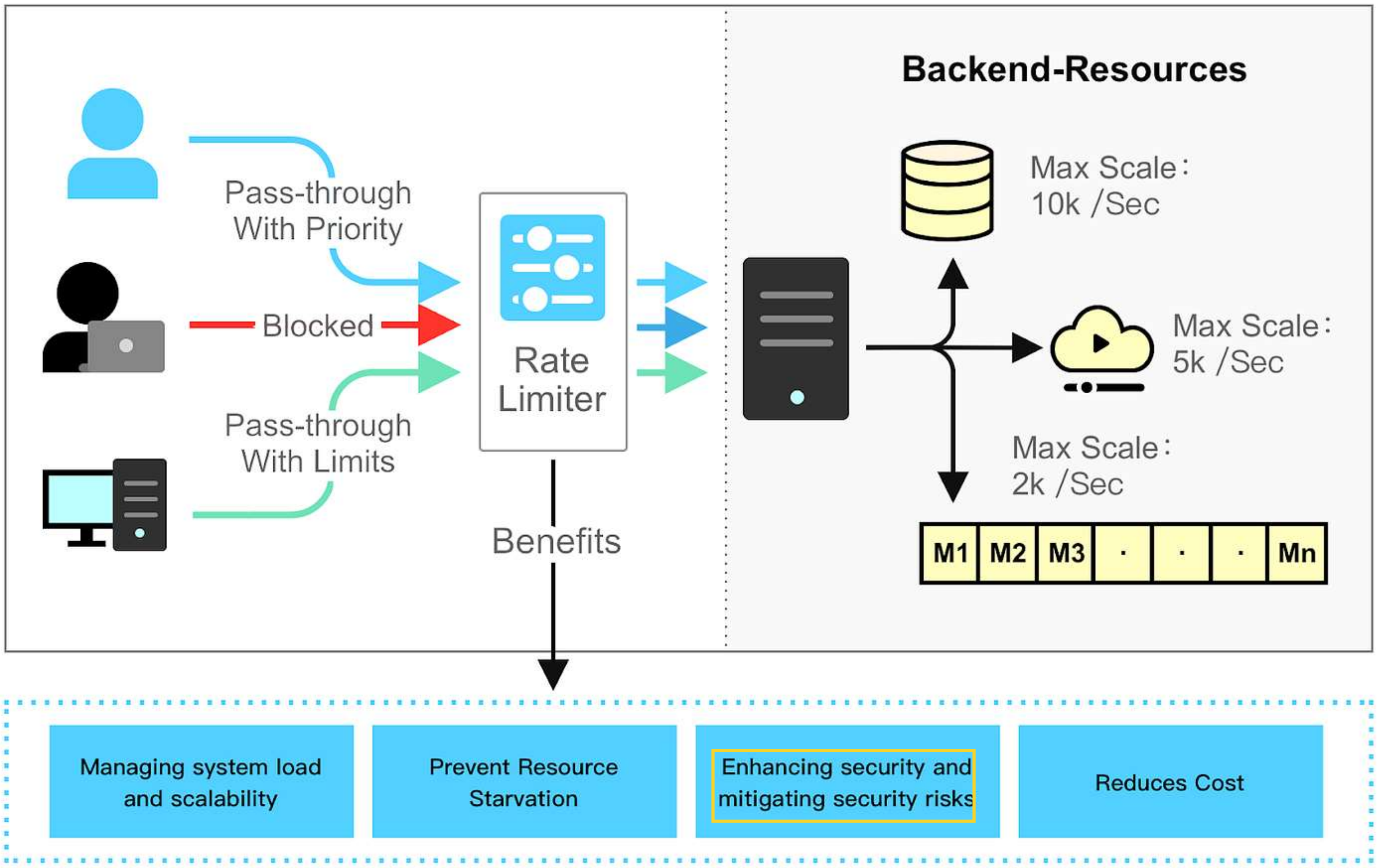

A User can send 2 msgs / sec
Msg: "Hi"
Receives: "Hi"
User A
Chat Server
Uesr B
One can create a max of 10 accounts / day / ip address
IP Address:
158.18.48.29
Create an
Account
Anonymous User
Account Server
One can claim 5 rewards / week / device
Claim Reward
User X
Reward Server

Benefits of API rate limiter

- **Prevent resource starvation** caused by Denial of Service (DoS) attack. Either intentional or unintentional, by blocking the excess calls. Almost all APIs published by large tech companies enforce some form of rate limiting.
  - Twitter limits the number of tweets to 300 per 3 hours.
  - Google docs APIs default limit: 300 per user per 60 seconds for read requests.
- **Reduce cost:** Limiting excess requests means fewer servers and allocating more resources to high priority APIs. Rate limiting is extremely important for companies that use paid third party APIs.
  - For example, you are charged on a per-call basis for the following external APIs: check credit, make a payment, retrieve health records, etc.
- **Prevent servers from being overloaded.** To reduce server load, a rate limiter is used to filter out excess requests caused by bots or users' misbehavior.

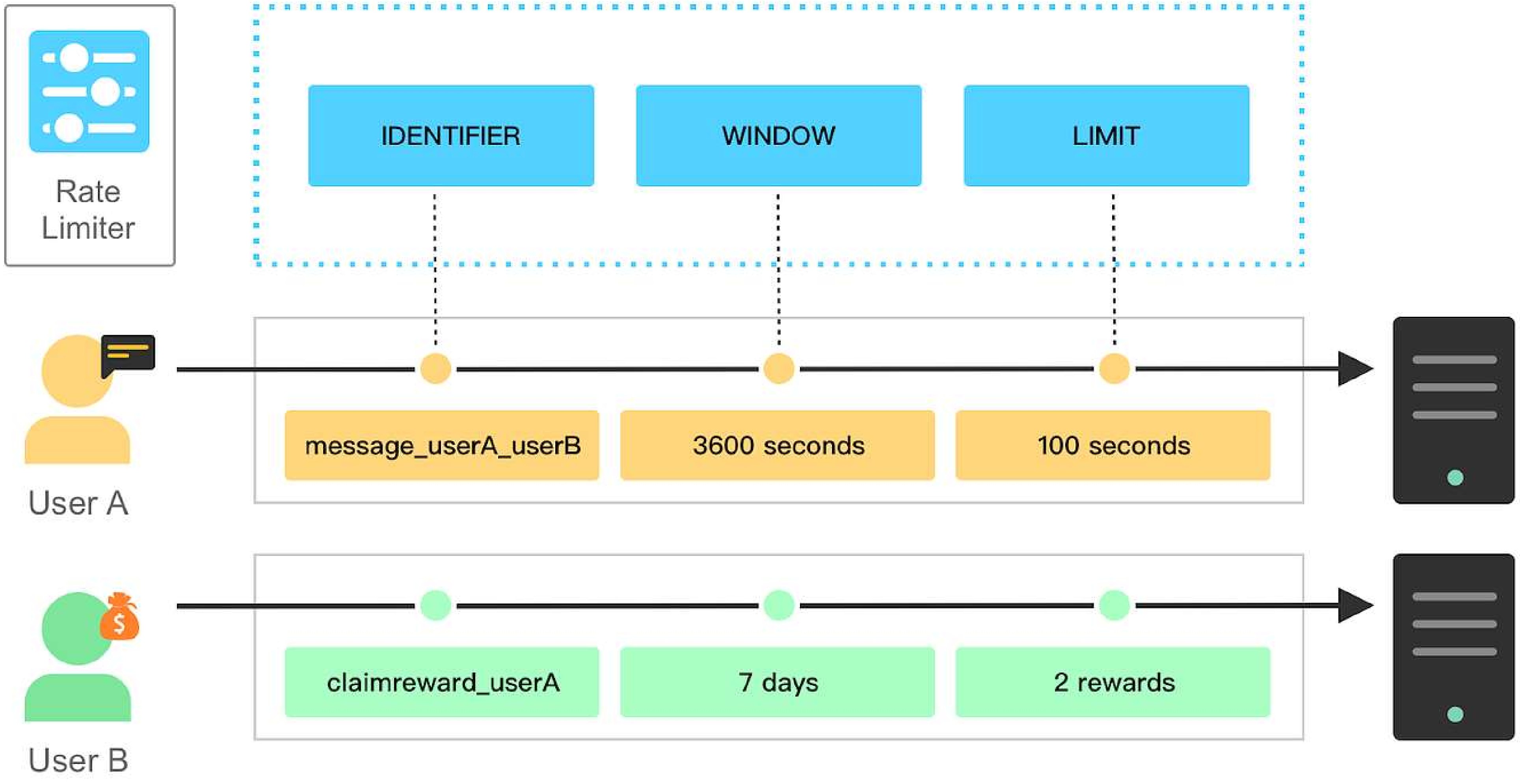
Use Cases

- **User level:** Consider a popular social media platform where users frequently post content and comments. To prevent **spam or malicious bot activity,** the platform might enforce user-level rate limiting. It restricts the number of posts or comments that a user can make in a given hour.
- **Application level:** One example is an online ticketing platform. On the day of a major concert sale, the platform can expect a significant surge in traffic. Application-level rate limit can be very useful in this case. It limits the total **number of ticket purchases per minute**. This practice protects the system from being overwhelmed and ensures a fair chance for everyone to try to secure a ticket.
- **API-level rate limiting:** Consider a cloud storage service that provides an API for uploading and downloading files. To ensure fair use and protect the system from misuse, the service might enforce limits on the number of API calls each user can make per minute.
- **User account levels:** A SaaS platform offering multiple tiers of service can have different usage limits for each tier. **Free tier users** may have a lower rate limit compared to **premium tier users.** This effectively manages resource usage while encouraging users to upgrade to higher limits.


Pass-through
With Priority
Blocked
Pass-through
With Limits
Rate
Limiter
Benefits
Backend-Resources
Max Scale：
10k /Sec
Max Scale：
5k /Sec
Max Scale：
2k /Sec
M1
M2
M3
·
·
·
Mn
Managing system load and scalability
Prevent Resource Starvation
Enhancing security and mitigating security risks
Reduces Cost

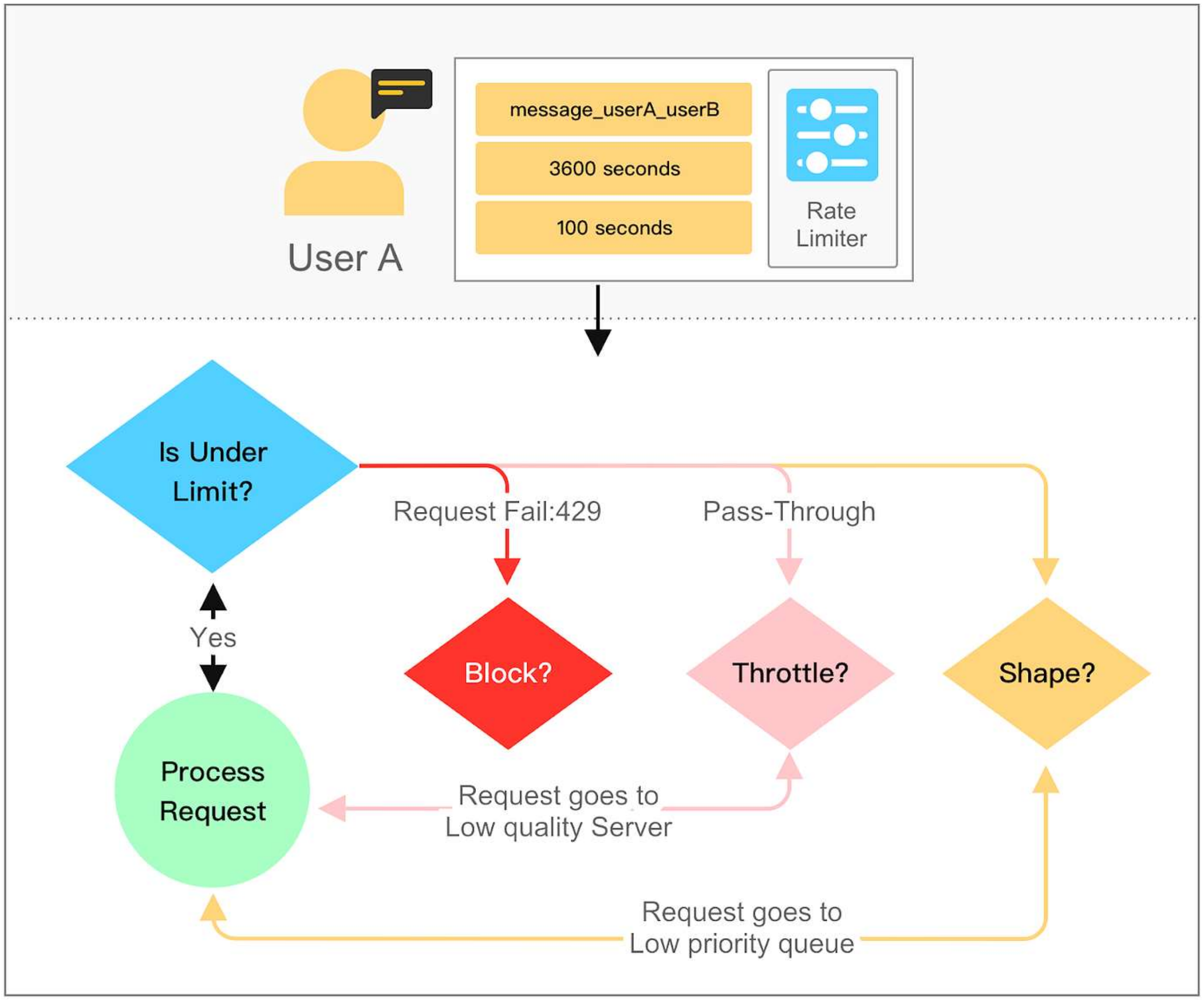
Core Concepts

- The **limit** defines the ceiling for allowable requests or actions within a designated time span. For example, we might allow a user to send no more than 100 messages every hour.
- The **window** is the time period where the limit comes into play. It could be any length of time, whether it be an hour, a day, or a week. Longer durations do have their own implementation challenges, like storage durability, that we'll discuss later.
- The **identifier** is a unique attribute that differentiates between individual callers. A user ID or IP address is a common example.

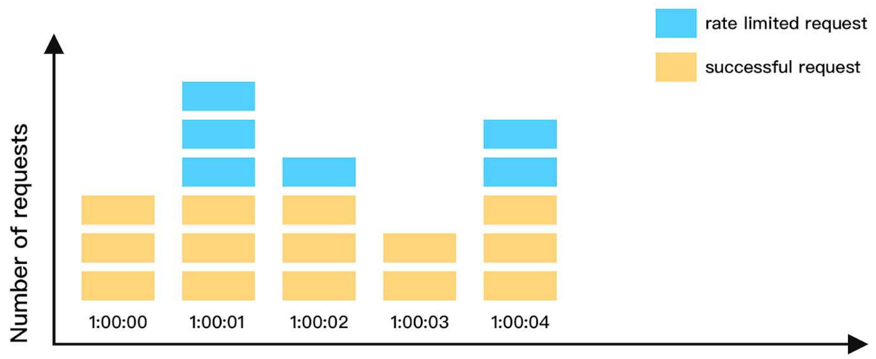

Rate Limiter
IDENTIFIER
WINDOW
LIMIT
User A
message_userA_userB
3600 seconds
100 seconds
User B
claimreward_userA
7 days
2 rewards

Responses

- **Blocking** takes place when requests exceeding the limit are denied access to the resource. It is commonly expressed as an error message such as HTTP status code 429 (Too Many Requests).
- **Throttling**, by comparison, involves slowing down or delaying the requests that go beyond the limit. An example would be a video streaming service reducing the quality of the stream for users who have gone over their data cap.
- **Shaping,** on the other hand, allows requests that surpass the limit. But those requests are assigned lower priority. This ensures that users who abide by the limits receive quality service. For example, in a content delivery network, requests from users who have crossed their limits may be processed last, while those from normal users are prioritized.


User A
message_userA_userB
3600 seconds
100 seconds
Rate Limiter
Is Under Limit?
Yes
Process Request
Request Fail:429
Pass-Through
Block?
Throttle?
Shape?
Request goes to Low quality Server
Request goes to Low priority queue

Rate Limiting Algorithms

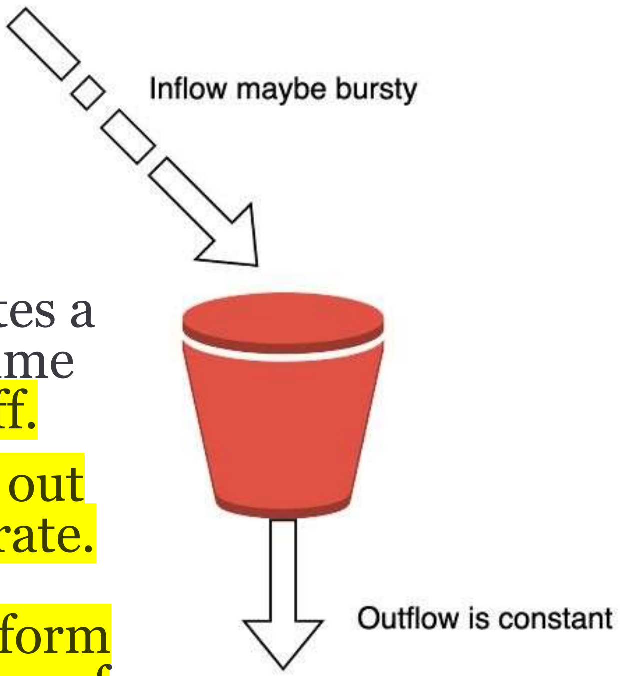


Fixed Window Counter

- One of the most basic rate limiting mechanisms. We keep a counter for a given duration of time and continue incrementing it for every request we get. Once the limit is reached, we drop all further requests until the time duration is reset.
- **Advantage:** Most recent requests served w/o being starved by old requests.
- **Problem:** A single burst of traffic right at the edge of the limit might hoard all the available slots for both the current and next time slot.
- Consumers might bombard the server at the edge in an attempt to maximize the number of requests served.

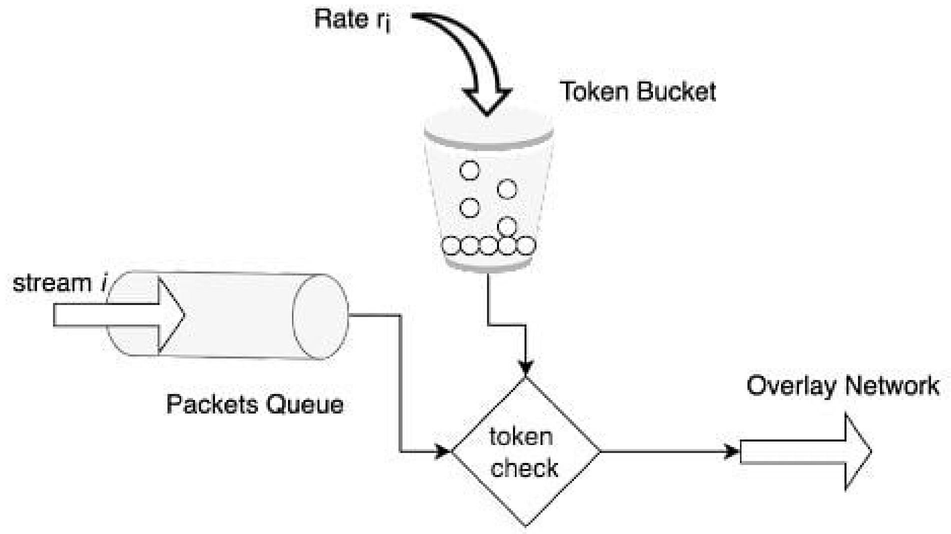
Leaky Bucket

- Leaky bucket is a simple, intuitive algorithm. It creates a queue with a finite capacity. All requests in a given time frame beyond the capacity of the queue are spilled off.
- The advantage of this algorithm is that it smoothens out bursts of requests and processes them at a constant rate. It's also easy to implement on a load balancer and is memory efficient for each user. A constant, near-uniform flow is applied to the server irrespective of the number of requests.
- The downside of this algorithm is that a burst of requests can fill up the bucket leading to the starvation of new requests. It also provides no guarantee that requests get completed in a given amount of time.



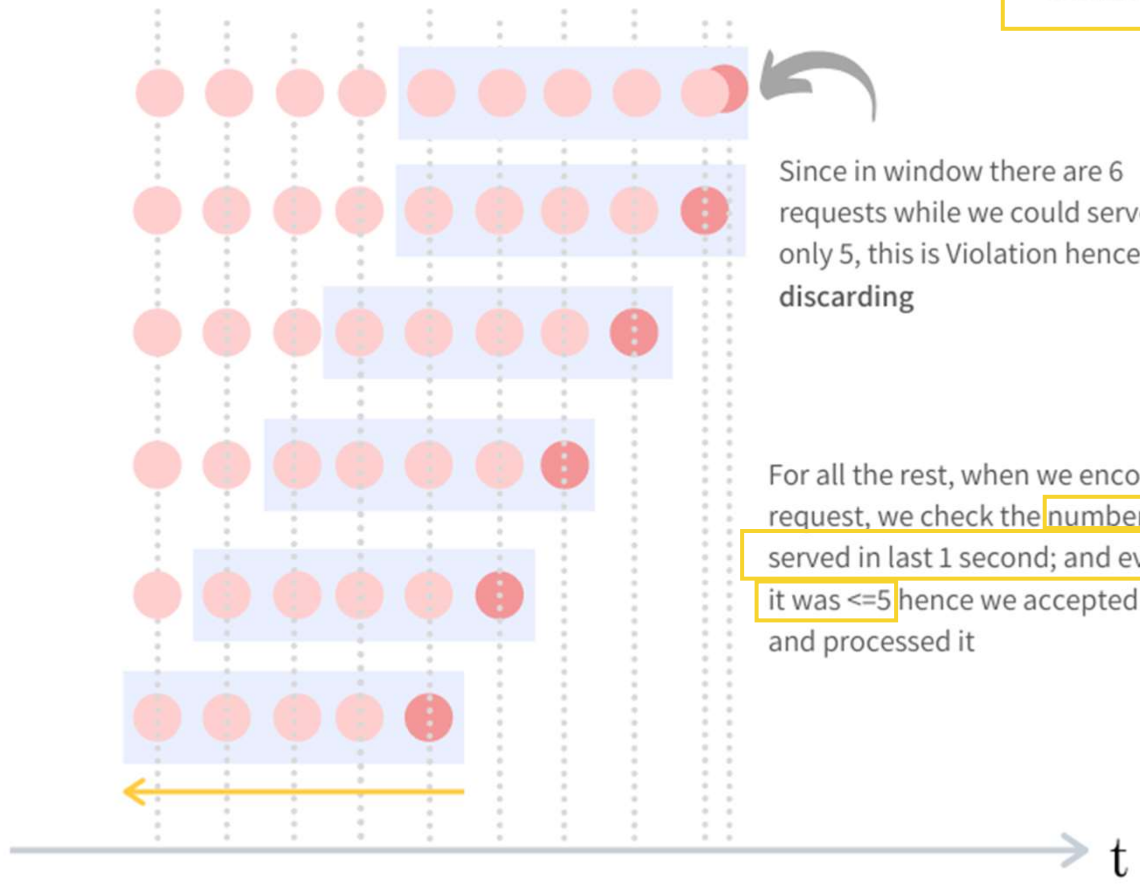
Token Bucket

- The token bucket algorithm is similar to leaky bucket, but instead, we assign tokens on a user level.
- For a given time duration *d*, the number of request *r* packets that a user can receive is defined. Every time a new request arrives at a server, there are two operations that happen:



•**Fetch token:** The current number of tokens for that user is fetched. If it is greater than the limit defined, then the request is dropped.
•**Update token:** If the fetched token is less than the limit for the time duration d, then the request is accepted and the token is appended.

Sliding Window Rate Limiting for 5 req/sec



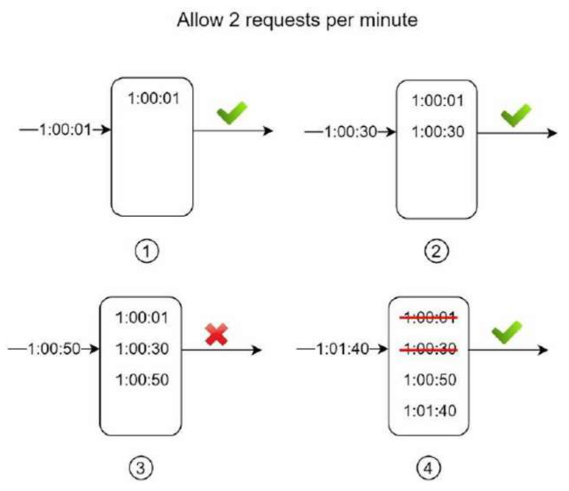
● New request ● Request

▨ Time window

Since in window there are 6 requests while we could serve only 5, this is Violation hence **discarding**

For all the rest, when we encountered a new request, we check the number of requests it served in last 1 second; and evry other time it was <=5 hence we accepted the request and processed it

Sliding Window Log



- Maintain a time stamped log of requests at the user level.
- The system keeps these requests time sorted in a set or a table. It discards all requests with timestamps beyond a specified threshold.
- Every minute we look out for older requests and filter them out. Then we calculate the sum of logs to determine the request rate. If the request would exceed the threshold rate, then it is held. Otherwise, the request is served.
- The advantage is that it does not suffer from the boundary conditions of fixed windows. Enforcement of the rate limit will remain precise. Since the system tracks the sliding log for each consumer, you don't have the stampede effect that challenges fixed windows.
- However, it can be costly 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. As a result, it **does not scale well enough to handle large bursts of traffic or denial of service attacks**.

Sliding Window Counter

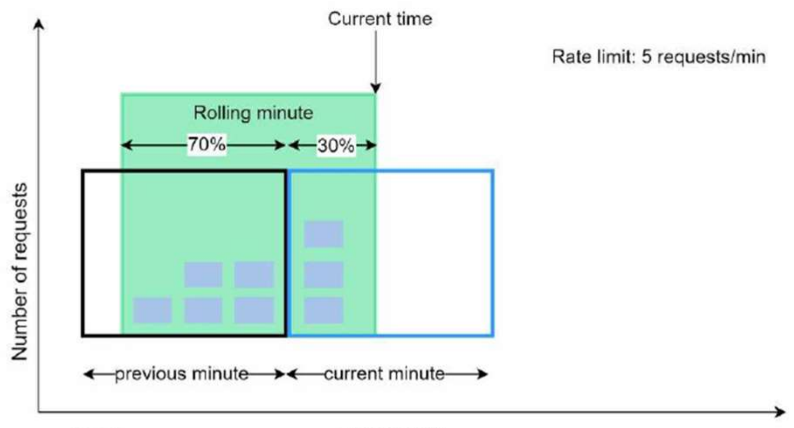
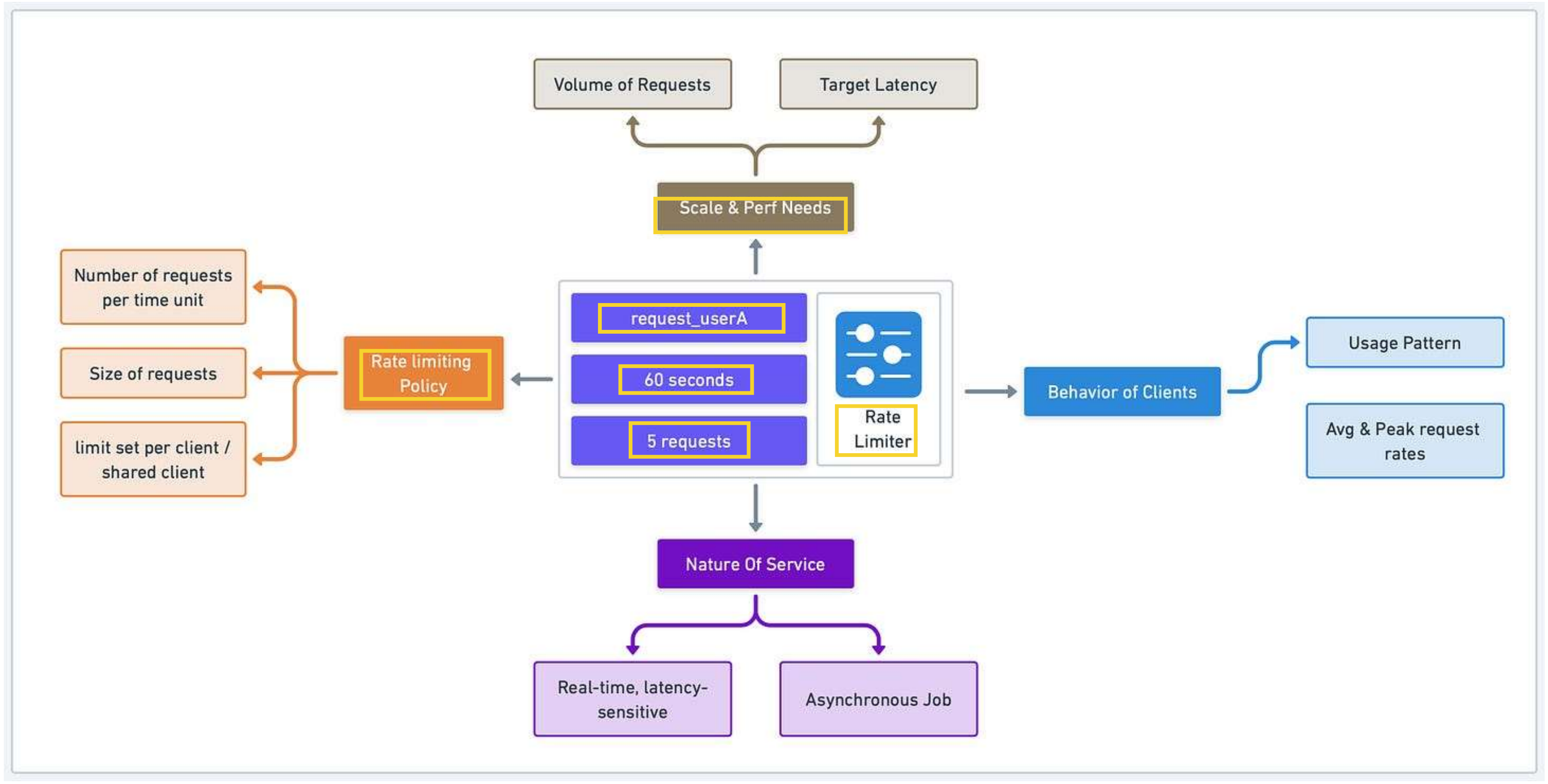


Figure 4-11

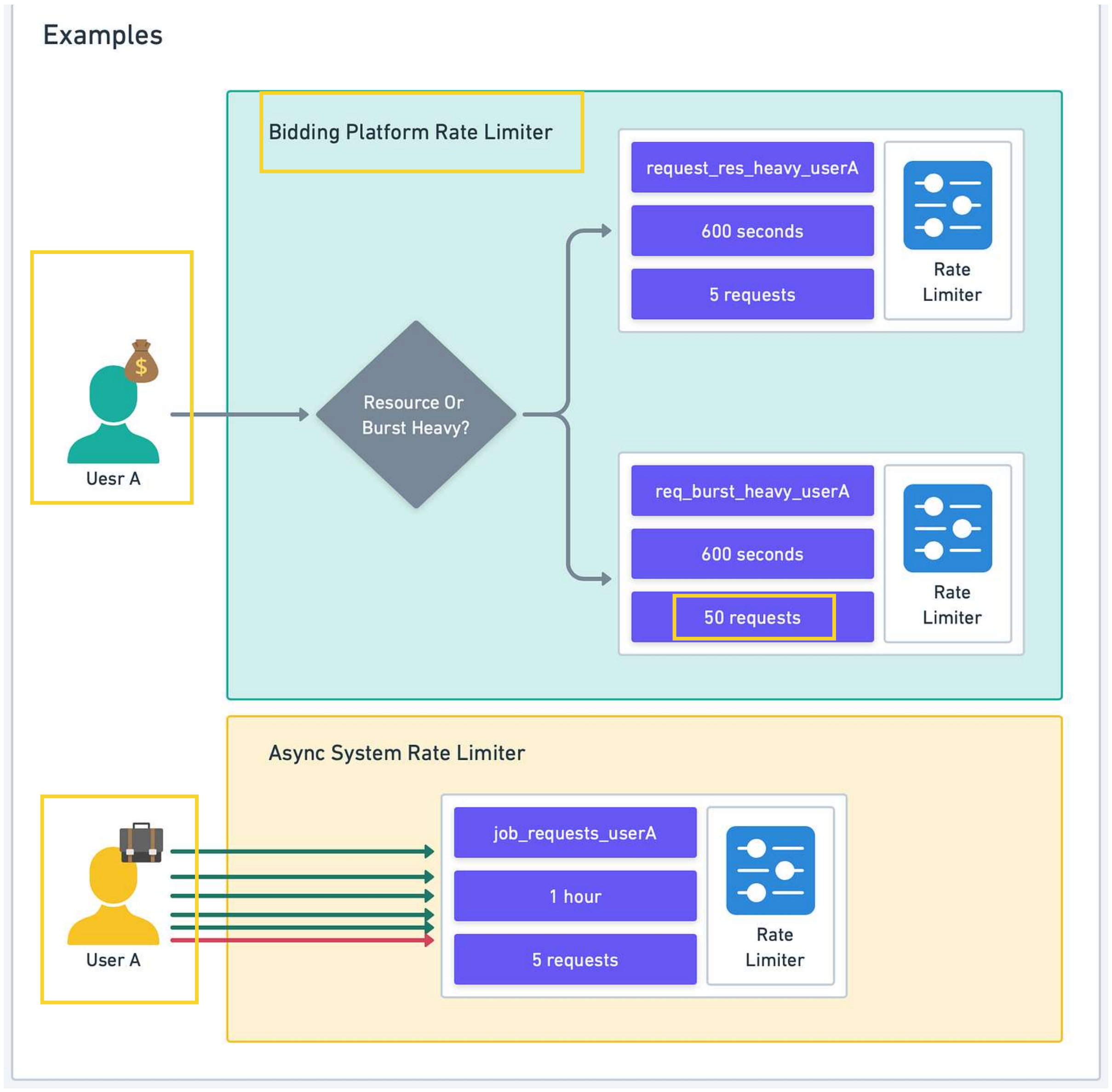
- This is similar to the sliding log algorithm, but it's more memory efficient. It combines the fixed window algorithm's low processing cost and the sliding log's improved boundary conditions.
- With this algorithm, we keep a list/table of time sorted entries, each being a hybrid and **containing the timestamp and the number of requests at that point.**
- We keep a sliding window of our time duration and only service requests in our window for the given rate. If the sum of counters is more than the given rate of the limiter, then we take only the first sum of entries equal to the rate limit.
- The sliding window approach is the best because it gives you the flexibility to scale rate limiting while still maintaining good performance.
- The rate windows are an intuitive way to present rate limit data to API consumers. It also avoids the starvation problem of the leaky bucket and the bursting problems of fixed window implementations.

Designing for the real world

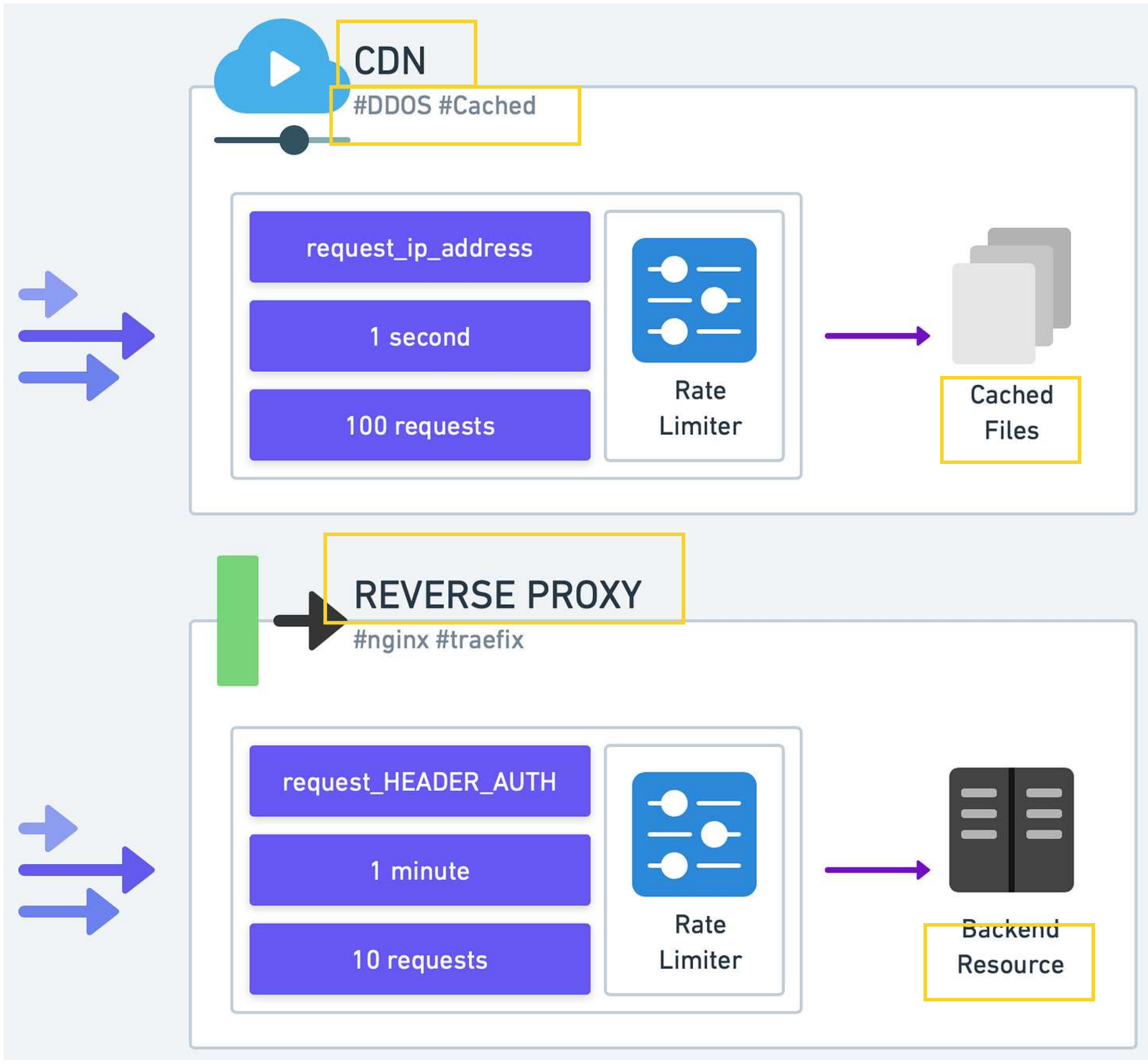
(Requirements)



Understanding the Use Cases

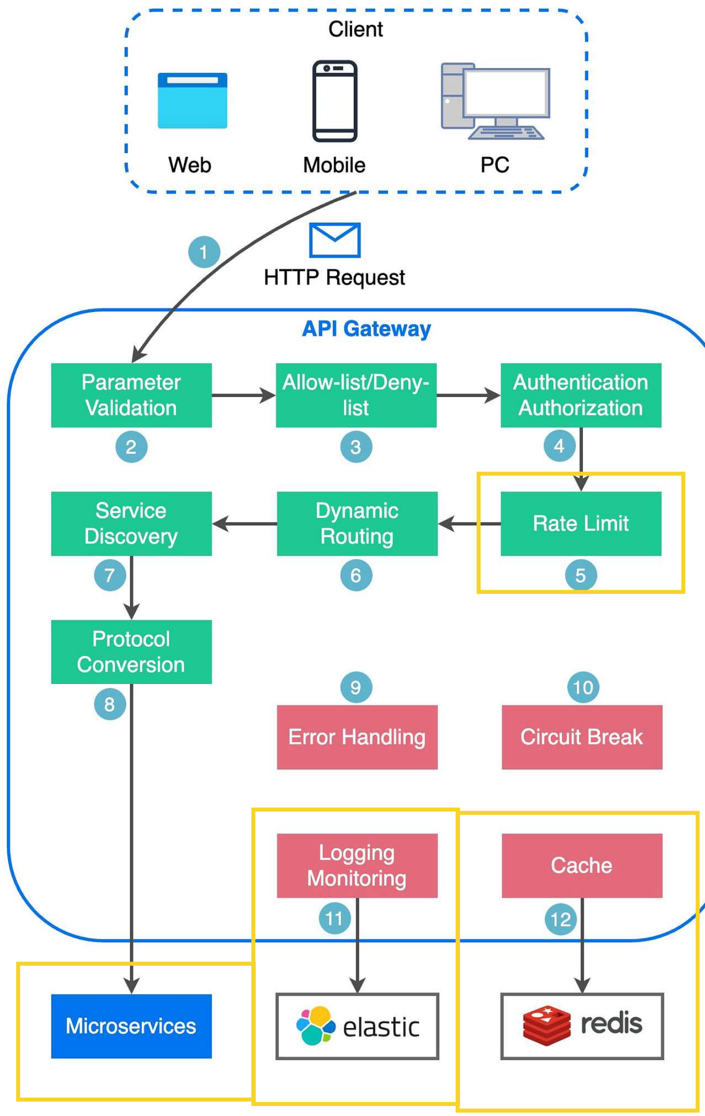


Where to Implement Rate Limiter?


CDN
#DDOS #Cached
request_ip_address
1 second
100 requests
Rate Limiter
Cached Files
REVERSE PROXY
#nginx #traefix
request_HEADER_AUTH
1 minute
10 requests
Rate Limiter
Backend Resource

What does API Gateway do?

blog.bytebytego.com



Application Framework and Middleware

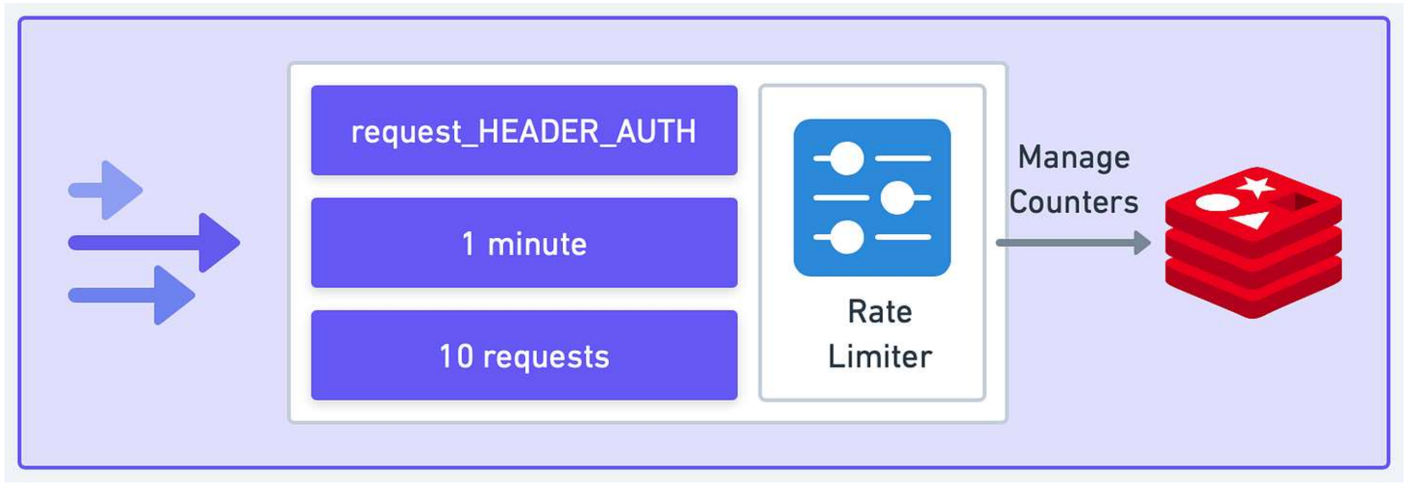
- If our rate limiting needs require more fine-grained identification of the resource to limit, we may need to place the rate limiter closer to the application logic. For example, if user specific attributes like subscription type need limiting, we'll need to implement the rate limiter at this level.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

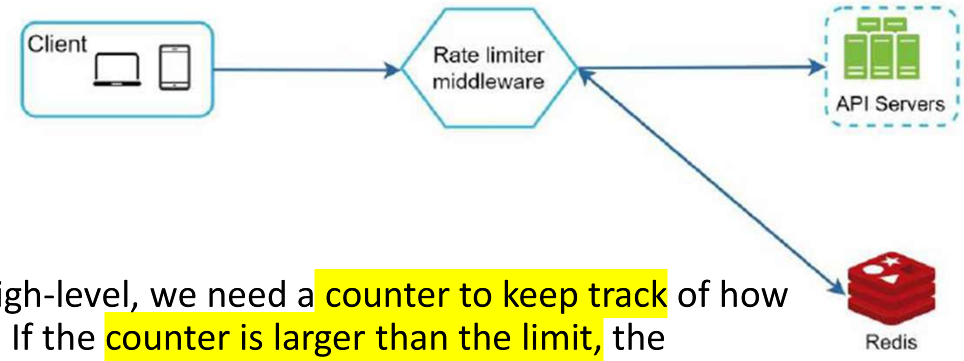
- Finally, if necessary, we could incorporate rate limiting logic directly in the application code. In some instances, the rate limiting requirements are so specific that this is the only feasible option.
- This offers the highest degree of control and visibility but introduces complexity and potential 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.

How to implement?

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.

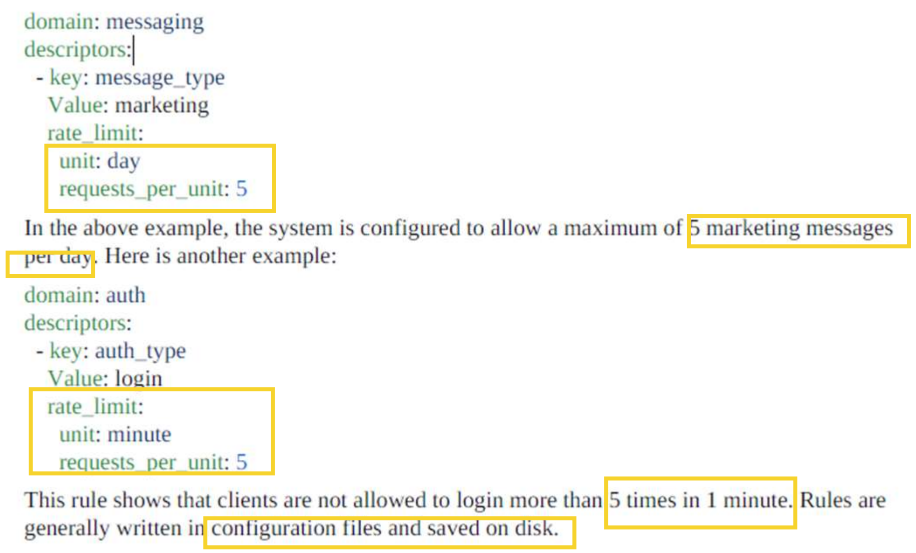


Implementation Details

- The basic idea of rate limiting algorithms is simple. At the high-level, we need a counter to keep track of how many requests are sent from the same user, IP address, etc. If the counter is larger than the limit, the request is disallowed.
- Where shall we store counters? Using the database is not a good idea due to slowness of disk access. In-memory cache is chosen because it is fast and supports time-based expiration strategy. For instance, Redis [11] is a popular option to implement rate limiting. It is an in-memory store that offers two commands: INCR and EXPIRE.
- INCR: It increases the stored counter by 1.
- EXPIRE: It sets a timeout for the counter. If the timeout expires, the counter is automatically deleted
- The client sends a request to rate limiting middleware.
- Rate limiting middleware fetches the counter from the corresponding bucket in Redis and checks if the limit is reached or not.
- If the limit is reached, the request is rejected.
- If the limit is not reached, the request is sent to API servers. Meanwhile, the system increments the counter and saves it back to Redis.

Design Deep Dive

- How are rate limiting rules created? Where are the rules stored?
- How to handle requests that are rate limited?
- In this section, we will first answer the questions regarding rate limiting rules and then go over the strategies to handle rate-limited requests.
- Finally, we will discuss rate limiting in distributed environment, a detailed design, performance optimization and monitoring.



```
domain: messaging
descriptors:
  - key: message_type
    Value: marketing
    rate_limit:
      unit: day
      requests_per_unit: 5
```

In the above example, the system is configured to allow a maximum of 5 marketing messages per day. Here is another example:

```
domain: auth
descriptors:
  - key: auth_type
    Value: login
    rate_limit:
      unit: minute
      requests_per_unit: 5
```

This rule shows that clients are not allowed to login more than 5 times in 1 minute. Rules are generally written in configuration files and saved on disk.

Exceeding Rate Limit

- In case a request is rate limited, APIs return a HTTP response code 429 (too many requests) to the client. Depending on the use cases, we may enqueue the rate-limited requests to be processed later. For example, if some orders are rate limited due to system overload, we may keep those orders to be processed later.
- **Rate limiter headers**
- How does a client know whether it is being throttled? And how does a client know the number of allowed remaining requests before being throttled? The answer lies in HTTP response headers. The rate limiter returns the following HTTP headers to clients:
  - *X-Ratelimit-Remaining*: The remaining number of allowed requests within the window.
  - *X-Ratelimit-Limit:* It indicates how many calls the client can make per time window.
  - *X-Ratelimit-Retry-After:* The number of seconds to wait until you can make a request again without being throttled.
- When a user has sent too many requests, a 429 too many requests error and *X-Ratelimit-Retry-After* header are returned to the client

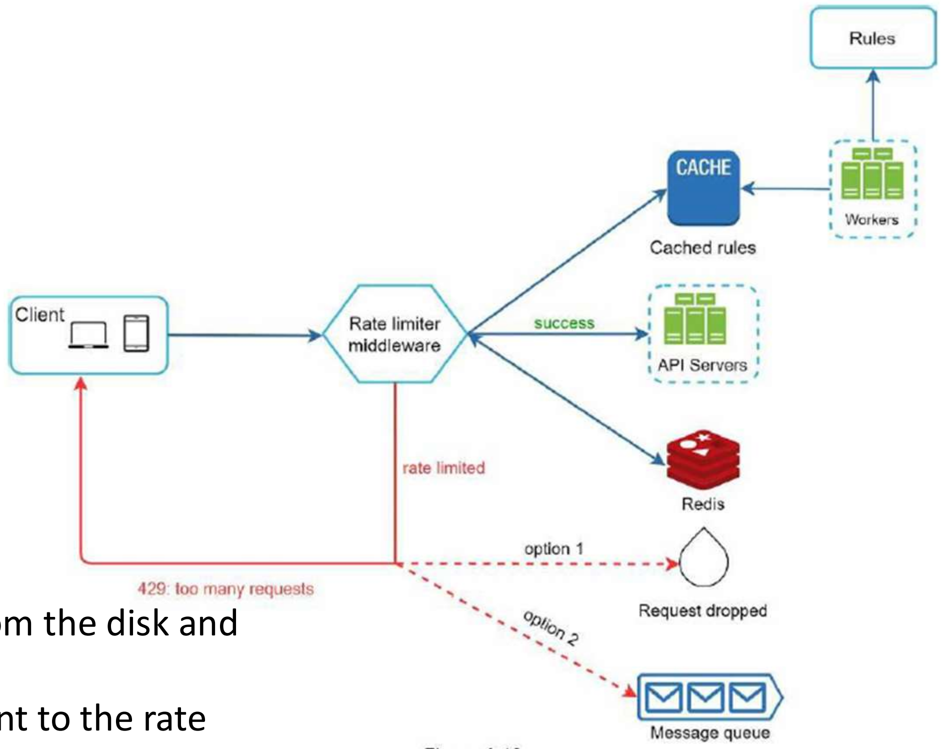
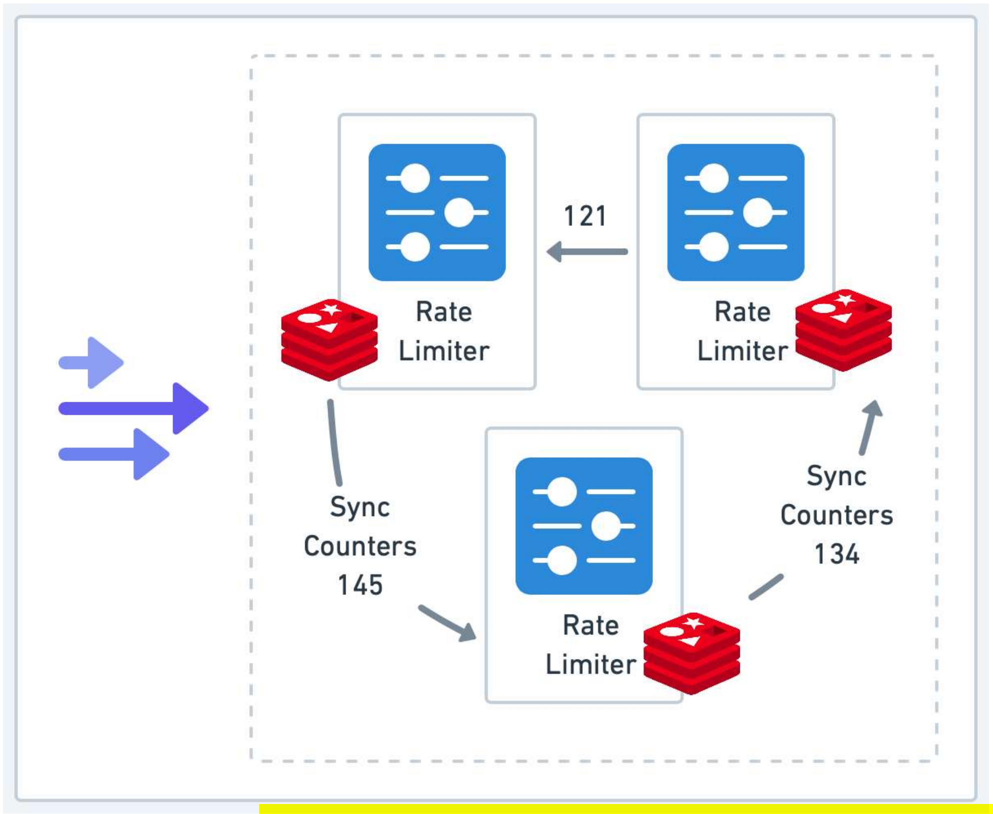


Figure 4-13

- Rules are stored on the disk. Workers frequently pull rules from the disk and store them in the cache.
- When a client sends a request to the server, the request is sent to the rate limiter middleware first.
- Rate limiter middleware loads rules from the cache. It fetches counters and last request timestamp from Redis cache. Based on the response, the rate limiter decides:
- if the request is not rate limited, it is forwarded to API servers.
- if the request is rate limited, the rate limiter returns 429 too many requests error to the client. In the meantime, the request is either dropped or forwarded to the queue.



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.

Solution Options

- Locks could be slow.
- Lua script and sorted sets in Redis are options.

- First, multi-data center setup is crucial for a rate limiter because latency is high for users located far away from the data center. Most cloud service providers build many edge server locations around the world. For example, as of 5/20 2020, Cloudflare has 194 geographically distributed edge servers [14]. Traffic is automatically routed to the closest edge server to reduce latency.
- Second, synchronize data with an eventual consistency model.