Workgroup: HTTPAPI
Internet-Draft:
draft-ietf-httpapi-ratelimit-headers-06
Published: 22 December 2022
Intended Status: Standards Track
Expires: 25 June 2023

Authors: R. Polli — Team Digitale, Italian Government
A. Martinez — Red Hat

RateLimit header fields for HTTP

Abstract

This document defines the RateLimit-Limit, RateLimit-Remaining, RateLimit-Reset and RateLimit-Policy HTTP header fields for servers to advertise their current service rate limits, thereby allowing clients to avoid being throttled.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-httpapi-ratelimit-headers/.

Discussion of this document takes place on the HTTPAPI Working Group mailing list (mailto:httpapi@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/httpapi/. Subscribe at https://www.ietf.org/mailman/listinfo/httpapi/. Working Group information can be found at https://datatracker.ietf.org/wg/httpapi/about/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-httpapi/ratelimit-headers.

Status of This Memo

This Internet-Draft will expire on 25 June 2023.

Copyright Notice

Table of Contents

1. Introduction
   1.1. Goals
   1.2. Notational Conventions
2. Concepts
   2.1. Quota Policy
   2.2. Service Limit
   2.3. Time Window
3. RateLimit header field Definitions
   3.1. RateLimit-Limit
   3.2. RateLimit-Policy
   3.3. RateLimit-Remaining
   3.4. RateLimit-Reset
4. Server Behavior
   4.1. Performance Considerations
5. Client Behavior
   5.1. Intermediaries
   5.2. Caching
6. Security Considerations
   6.1. Throttling does not prevent clients from issuing requests
   6.2. Information disclosure
   6.3. Remaining quota units are not granted requests
   6.4. Reliability of RateLimit-Reset
   6.5. Resource exhaustion
      6.5.1. Denial of Service
7. Privacy Considerations
8. IANA Considerations
   8.1. RateLimit Parameters Registration
9. References
   9.1. Normative References
   9.2. Informative References

Appendix A. Rate-limiting and quotas
A.1. Interoperability issues
Appendix B. Examples
B.1. Unparameterized responses
B.1.1. Throttling information in responses
B.1.2. Use in conjunction with custom fields
B.1.3. Use for limiting concurrency
B.1.4. Use in throttled responses
B.2. Parameterized responses
B.2.1. Throttling window specified via parameter
B.2.2. Dynamic limits with parameterized windows
B.2.3. Dynamic limits for pushing back and slowing down
B.3. Dynamic limits for pushing back with Retry-After and slow down
B.3.1. Missing Remaining information
B.3.2. Use with multiple windows
FAQ
RateLimit header fields currently used on the web
Acknowledgements
Changes
Since draft-ietf-httpapi-ratelimit-headers-03
Since draft-ietf-httpapi-ratelimit-headers-02
Since draft-ietf-httpapi-ratelimit-headers-01
Since draft-ietf-httpapi-ratelimit-headers-00
Authors' Addresses

1. Introduction

Rate limiting HTTP clients has become a widespread practice, especially for HTTP APIs. Typically, servers who do so limit the number of acceptable requests in a given time window (e.g. 10 requests per second). See Appendix A for further information on the current usage of rate limiting in HTTP.

Currently, there is no standard way for servers to communicate quotas so that clients can throttle its requests to prevent errors. This document defines a set of standard HTTP header fields to enable rate limiting:

*RateLimit-Limit: the server's quota for requests by the client in the time window,

*RateLimit-Remaining: the remaining quota in the current window,

*RateLimit-Reset: the time remaining in the current window, specified in seconds, and

*RateLimit-Policy: the quota policy.

These fields allow the establishment of complex rate limiting policies, including using multiple and variable time windows and dynamic quotas, and implementing concurrency limits.

The behavior of the RateLimit-Reset header field is compatible with the delay-seconds notation of Retry-After.

1.1. Goals

The goals of this document are:

Interoperability: Standardization of the names and semantics of rate-limit headers to ease their enforcement and adoption;

Resiliency: Improve resiliency of HTTP infrastructure by providing clients with information useful to throttle their requests and prevent 4xx or 5xx responses;

Documentation: Simplify API documentation by eliminating the need to include detailed quota limits and related fields in API documentation.

The following features are out of the scope of this document:

Authorization: RateLimit header fields are not meant to support authorization or other kinds of access controls.

Throttling scope: This specification does not cover the throttling scope, that may be the given resource-target, its parent path or the whole Origin (see Section 7 of [WEB-ORIGIN]). This can be addressed using extensibility mechanisms such as the parameter registry Section 8.1.

Response status code: RateLimit header fields may be returned in both successful (see Section 15.3 of [HTTP]) and non-successful responses. This specification does not cover whether non Successful responses count on quota usage, nor it mandates any correlation between the RateLimit values and the returned status code.

Throttling policy: This specification does not mandate a specific throttling policy. The values published in the fields, including the window size, can be statically or dynamically evaluated.

Service Level Agreement: Conveyed quota hints do not imply any service guarantee. Server is free to throttle respectful clients under certain circumstances.

1.2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the Augmented BNF defined in [RFC5234] and updated by [RFC7405] along with the "#rule" extension defined in Section 5.6.1 of [HTTP].

The term Origin is to be interpreted as described in Section 7 of [WEB-ORIGIN].

This document uses the terms List, Item and Integer from Section 3 of [STRUCTURED-FIELDS] to specify syntax and parsing, along with the concept of "bare item".

The header fields defined in this document are collectively referred to as "RateLimit header fields".

2. Concepts

2.1. Quota Policy

A quota policy is maintained by a server to limit the activity (counted in quota units) of a given client over a period of time (known as the time window (Section 2.3)) to a specified amount (known as the service limit (Section 2.2)).

Quota policies can be advertised by servers (see Section 3.2), but they are not required to be, and more than one quota policy can affect a given request from a client to a server.

A quota policy is expressed in Structured Fields [STRUCTURED-FIELDS] as an Integer that indicates the service limit with associated parameters.

The following Parameters are defined in this specification:

w: The REQUIRED "w" parameter value conveys a time window (Section 2.3).

For example, a quota policy of 100 quota units per minute is expressed as:

```
100;w=60
```

Other parameters are allowed and can be regarded as comments. Parameters for use by more than one implementation or service ought to be registered within the "Hypertext Transfer Protocol (HTTP) RateLimit Parameters Registry", as described in Section 8.1.

Implementation- or service-specific parameters SHOULD be prefixed parameters with a vendor identifier, e.g. acme-policy, acme-burst.

2.2. Service Limit

The service limit is a non-negative Integer indicating the maximum amount of activity that the server is willing to accept from what it identifies as the client (e.g., based upon originating IP or user authentication) during a time window (Section 2.3).

The activity being limited is usually the HTTP requests made by the client; for example "you can make 100 requests per minute". However, a server might only rate limit some requests (based upon URI, method, user identity, etc.), and it might weigh requests differently. Therefore, quota policies are defined in terms of "quota units". Servers SHOULD document how they count quota units.

For example, a server could count requests like /books/{id} once, but count search requests like /books?author=WuMing twice. This might result in the following counters:

```
GET /books/123           ; service-limit=4, remaining: 3, status=200
GET /books?author=WuMing ; service-limit=4, remaining: 1, status=200
GET /books?author=Eco     ; service-limit=4, remaining: 0, status=429
```

Often, the service limit advertised will match the server's actual limit. However, it MAY differ when weight mechanisms, bursts, or other server policies are implemented. In that case the difference SHOULD be communicated using an extension or documented separately.

2.3. Time Window

Quota policies limit the number of acceptable requests within a given time interval, known as a time window.

The time window is a non-negative Integer value expressing that interval in seconds, similar to the "delay-seconds" rule defined in Section 10.2.3 of [HTTP]. Subsecond precision is not supported.

By default, a quota policy does not constrain the distribution of quota units within the time window. If necessary, these details can be conveyed as extension parameters.

For example, two quota policies containing further details via extension parameters:

```
100;w=60;comment="fixed window"
12;w=1;burst=1000;policy="leaky bucket"
```

3. RateLimit header field Definitions

The following RateLimit response header fields are defined.

3.1. RateLimit-Limit

The "RateLimit-Limit" response header field indicates the service limit (Section 2.2) associated with the client in the current time window (Section 2.3). If the client exceeds that limit, it MAY not be served.

The field is an Item and its value is a non-negative Integer referred to as the "expiring-limit". This specification does not define Parameters for this field. If they appear, they MUST be ignored.

The expiring-limit MUST be set to the service limit that is closest to reaching its limit, and the associated time window MUST either be:

- inferred by the value of RateLimit-Reset header field at the moment of the reset, or

- communicated out-of-band (e.g. in the documentation).

Example:

```
RateLimit-Limit: 100
```

The RateLimit-Policy header field (see Section 3.2), might contain information on the associated time window.

This field cannot appear in a trailer section.

3.2. RateLimit-Policy

The "RateLimit-Policy" response header field indicates the quota policies currently associated with the client. Its value is informative.

The field is a non-empty List of Items. Each item is a quota policy (Section 2.1). Two quota policies MUST NOT be associated with the same quota units value.

This field can convey the time window associated with the expiring-limit, as shown in this example:

```
RateLimit-Policy: 100;w=10
RateLimit-Limit: 100
```

These examples show multiple policies being returned:

```
RateLimit-Policy: 10;w=1, 50;w=60, 1000;w=3600, 5000;w=86400
RateLimit-Policy: 10;w=1;burst=1000, 1000;w=3600
```

An example of invalid header field value with two policies associated with the same quota units:

```
RateLimit-Policy: 10;w=1, 10;w=60
```

This field cannot appear in a trailer section.

3.3. RateLimit-Remaining

The "RateLimit-Remaining" response header field indicates the remaining quota units associated with the expiring-limit.

The field is an Item and its value is a non-negative Integer expressed in quota units (Section 2.2). This specification does not define Parameters for this field. If they appear, they MUST be ignored.

Clients MUST NOT assume that a positive RateLimit-Remaining header field value is a guarantee that further requests will be served.

When the value of RateLimit-Remaining is low, it indicates that the server may soon throttle the client (see Section 4).

For example:

```
RateLimit-Remaining: 50
```

This field cannot appear in a trailer section.

3.4. RateLimit-Reset

The "RateLimit-Reset" response header field indicates the number of seconds until the quota associated with the expiring-limit resets.

The field is a non-negative Integer compatible with the delay-seconds rule, because:

- it does not rely on clock synchronization and is resilient to clock adjustment and clock skew between client and server (see Section 5.6.7 of [HTTP]);

- it mitigates the risk related to thundering herd when too many clients are serviced with the same timestamp.

This specification does not define Parameters for this field. If they appear, they MUST be ignored.

For example:

```
RateLimit-Reset: 50
```

The client MUST NOT assume that all its service limit will be reset at the moment indicated by the RateLimit-Reset header field. The server MAY arbitrarily alter the RateLimit-Reset header field value between subsequent requests; for example, in case of resource saturation or to implement sliding window policies.

This field cannot appear in a trailer section.

4. Server Behavior

A server uses the RateLimit header fields to communicate its quota policies. A response that includes the RateLimit-Limit header field MUST also include the RateLimit-Reset. It MAY also include a RateLimit-Remaining header field.

A server MAY return RateLimit header fields independently of the response status code. This includes on throttled responses. This document does not mandate any correlation between the RateLimit header field values and the returned status code.

Servers should be careful when returning RateLimit header fields in redirection responses (i.e., responses with 3xx status codes) because a low RateLimit-Remaining header field value could prevent the client from issuing requests. For example, given the RateLimit header fields below, a client could decide to wait 10 seconds before following the "Location" header field (see Section 10.2.2 of [HTTP]), because the RateLimit-Remaining header field value is 0.

```
HTTP/1.1 301 Moved Permanently
Location: /foo/123
RateLimit-Remaining: 0
RateLimit-Limit: 10
RateLimit-Reset: 10
```

If a response contains both the Retry-After and the RateLimit-Reset header fields, the RateLimit-Reset header field value SHOULD reference the same point in time as the Retry-After field value.

When using a policy involving more than one time window, the server MUST reply with the RateLimit header fields related to the time window with the lower RateLimit-Remaining header field values.

A service using RateLimit header fields MUST NOT convey values exposing an unwanted volume of requests and SHOULD implement mechanisms to cap the ratio between RateLimit-Remaining and RateLimit-Reset header field values (see Section 6.5); this is especially important when a quota policy uses a large time window.

Under certain conditions, a server MAY artificially lower RateLimit header field values between subsequent requests, e.g. to respond to Denial of Service attacks or in case of resource saturation.

4.1. Performance Considerations

Servers are not required to return RateLimit header fields in every response, and clients need to take this into account. For example, an implementer concerned with performance might provide RateLimit header fields only when a given quota is going to expire.

Implementers concerned with response fields' size, might take into account their ratio with respect to the content length, or use header-compression HTTP features such as [HPACK].

5. Client Behavior

The RateLimit header fields can be used by clients to determine whether the associated request respected the server's quota policy, and as an indication of whether subsequent requests will. However, the server might apply other criteria when servicing future requests, and so the quota policy may not completely reflect whether they will succeed.

For example, a successful response with the following fields:

```
RateLimit-Limit: 10
RateLimit-Remaining: 1
RateLimit-Reset: 7
```

does not guarantee that the next request will be successful. Servers' behavior may be subject to other conditions like the one shown in the example from Section 2.2.

A client is responsible for ensuring that RateLimit header field values returned cause reasonable client behavior with respect to throughput and latency (see Section 6.5 and Section 6.5.1).

A client receiving RateLimit header fields MUST NOT assume that future responses will contain the same RateLimit header fields, or any RateLimit header fields at all.

Malformed RateLimit header fields MUST be ignored.

A client SHOULD NOT exceed the quota units conveyed by the RateLimit-Remaining header field before the time window expressed in RateLimit-Reset header field.

A client MAY still probe the server if the RateLimit-Reset header field is considered too high.

The value of the RateLimit-Reset header field is generated at response time: a client aware of a significant network latency MAY behave accordingly and use other information (e.g. the "Date" response header field, or otherwise gathered metrics) to better estimate the RateLimit-Reset header field moment intended by the server.

The details provided in the RateLimit-Policy header field are informative and MAY be ignored.

If a response contains both the RateLimit-Reset and Retry-After fields, the Retry-After field MUST take precedence and the RateLimit-Reset header field MAY be ignored.

This specification does not mandate a specific throttling behavior and implementers can adopt their preferred policies, including:

- slowing down or preemptively back-off their request rate when approaching quota limits;

- consuming all the quota according to the exposed limits and then wait.

5.1. Intermediaries

This section documents the considerations advised in Section 16.3.2 of [HTTP].

An intermediary that is not part of the originating service infrastructure and is not aware of the quota policy semantic used by the Origin Server SHOULD NOT alter the RateLimit header fields' values in such a way as to communicate a more permissive quota policy; this includes removing the RateLimit header fields.

An intermediary MAY alter the RateLimit header fields in such a way as to communicate a more restrictive quota policy when:

- it is aware of the quota unit semantic used by the Origin Server;
- it implements this specification and enforces a quota policy which is more restrictive than the one conveyed in the fields.

An intermediary SHOULD forward a request even when presuming that it might not be serviced; the service returning the RateLimit header fields is the sole responsible of enforcing the communicated quota policy, and it is always free to service incoming requests.

This specification does not mandate any behavior on intermediaries respect to retries, nor requires that intermediaries have any role in respecting quota policies. For example, it is legitimate for a proxy to retransmit a request without notifying the client, and thus consuming quota units.

Privacy considerations (Section 7) provide further guidance on intermediaries.

5.2. Caching

[HTTP-CACHING] defines how responses can be stored and reused for subsequent requests, including those with RateLimit header fields. Because the information in RateLimit header fields on a cached response may not be current, they SHOULD be ignored on responses that come from cache (i.e., those with a positive current_age; see Section 4.2.3 of [HTTP-CACHING]).

6. Security Considerations

6.1. Throttling does not prevent clients from issuing requests

This specification does not prevent clients from making requests. Servers should always implement mechanisms to prevent resource exhaustion.

6.2. Information disclosure

Servers should not disclose to untrusted parties operational capacity information that can be used to saturate its infrastructural resources.

While this specification does not mandate whether non-successful responses consume quota, if error responses (such as 401 (Unauthorized) and 403 (Forbidden)) count against quota, a malicious client could probe the endpoint to get traffic information of another user.

As intermediaries might retransmit requests and consume quota units without prior knowledge of the user agent, RateLimit header fields might reveal the existence of an intermediary to the user agent.

6.3. Remaining quota units are not granted requests

RateLimit header fields convey hints from the server to the clients in order to help them avoid being throttled out.

Clients MUST NOT consider the quota units (Section 2.2) returned in RateLimit-Remaining header field as a service level agreement.

In case of resource saturation, the server MAY artificially lower the returned values or not serve the request regardless of the advertised quotas.

6.4. Reliability of RateLimit-Reset

Consider that service limit might not be restored after the moment referenced by RateLimit-Reset header field, and the RateLimit-Reset header field value may not be fixed nor constant.

Subsequent requests might return a higher RateLimit-Reset header field value to limit concurrency or implement dynamic or adaptive throttling policies.

6.5. Resource exhaustion

When returning RateLimit-Reset header field you must be aware that many throttled clients may come back at the very moment specified.

This is true for Retry-After too.

For example, if the quota resets every day at 18:00:00 and your server returns the RateLimit-Reset header field accordingly

```
Date: Tue, 15 Nov 1994 08:00:00 GMT
RateLimit-Reset: 36000
```

there's a high probability that all clients will show up at 18:00:00.

This could be mitigated by adding some jitter to the field-value.

Resource exhaustion issues can be associated with quota policies using a large time window, because a user agent by chance or on purpose might consume most of its quota units in a significantly shorter interval.

This behavior can be even triggered by the provided RateLimit header fields. The following example describes a service with an unconsumed quota policy of 10000 quota units per 1000 seconds.

```
RateLimit-Limit: 10000
RateLimit-Policy: 10000;w=1000
RateLimit-Remaining: 10000
RateLimit-Reset: 10
```

A client implementing a simple ratio between RateLimit-Remaining header field and RateLimit-Reset header field could infer an average throughput of 1000 quota units per second, while the RateLimit-Limit header field conveys a quota-policy with an average of 10 quota units per second. If the service cannot handle such load, it should return either a lower RateLimit-Remaining header field value or an higher RateLimit-Reset header field value. Moreover, complementing large time window quota policies with a short time window one mitigates those risks.

6.5.1. Denial of Service

RateLimit header fields may contain unexpected values by chance or on purpose. For example, an excessively high RateLimit-Remaining header field value may be:

- *used by a malicious intermediary to trigger a Denial of Service attack or consume client resources boosting its requests;
- *passed by a misconfigured server;

or a high RateLimit-Reset header field value could inhibit clients to contact the server (e.g. similarly to receiving "Retry-after: 1000000").

To mitigate this risk, clients can set thresholds that they consider reasonable in terms of quota units, time window, concurrent requests or throughput, and define a consistent behavior when the RateLimit exceed those thresholds. For example this means capping the maximum number of request per second, or implementing retries when the RateLimit-Reset exceeds ten minutes.

The considerations above are not limited to RateLimit header fields, but apply to all fields affecting how clients behave in subsequent requests (e.g. Retry-After).

7. Privacy Considerations

Clients that act upon a request to rate limit are potentially re-identifiable (see Section 5.2.1 of [PRIVACY]) because they react to information that might only be given to them. Note that this might apply to other fields too (e.g. Retry-After).

Since rate limiting is usually implemented in contexts where clients are either identified or profiled (e.g. assigning different quota units to different users), this is rarely a concern.

Privacy enhancing infrastructures using RateLimit header fields can define specific techniques to mitigate the risks of re-identification.

8. IANA Considerations

IANA is requested to update one registry and create one new registry.

Please add the following entries to the "Hypertext Transfer Protocol (HTTP) Field Name Registry" registry ([HTTP]):

Field Name	Status	Specification
RateLimit-Limit	permanent	Section 3.1 of RFC nnnn
RateLimit-Remaining	permanent	Section 3.3 of RFC nnnn
RateLimit-Reset	permanent	Section 3.4 of RFC nnnn
RateLimit-Policy	permanent	Section 3.2 of RFC nnnn

Table 1

8.1. RateLimit Parameters Registration

IANA is requested to create a new registry to be called "Hypertext Transfer Protocol (HTTP) RateLimit Parameters Registry", to be located at https://www.iana.org/assignments/http-ratelimit-parameters. Registration is done on the advice of a Designated Expert, appointed by the IESG or their delegate. All entries are Specification Required ([IANA], Section 4.6).

Registration requests consist of the following information:

*Parameter name: The parameter name, conforming to [STRUCTURED-FIELDS].

*Field name: The RateLimit header field for which the parameter is registered. If a parameter is intended to be used with multiple fields, it has to be registered for each one.

*Description: A brief description of the parameter.

*Specification document: A reference to the document that specifies the parameter, preferably including a URI that can be used to retrieve a copy of the document.

*Comments (optional): Any additional information that can be useful.

The initial contents of this registry should be:

Field Name	Parameter name	Description	Specification	Comments (optional)
RateLimit-Policy	w	Time window	Section 2.1 of RFC nnnn	

Table 2

9. References

9.1. Normative References

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[IANA] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/rfc/rfc7405>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[STRUCTURED-FIELDS] Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/rfc/rfc8941>.

[WEB-ORIGIN] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/rfc/rfc6454>.

9.2. Informative References

[HPACK] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/rfc/rfc7541>.

[HTTP-CACHING] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/rfc/rfc9111>.

[PRIVACY] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/rfc/rfc3339>.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <https://www.rfc-editor.org/rfc/rfc6585>.

[UNIX] The Open Group, "The Single UNIX Specification, Version 2 - 6 Vol Set for UNIX 98", February 1997.

Appendix A. Rate-limiting and quotas

Servers use quota mechanisms to avoid systems overload, to ensure an equitable distribution of computational resources or to enforce other policies - e.g. monetization.

A basic quota mechanism limits the number of acceptable requests in a given time window, e.g. 10 requests per second.

When quota is exceeded, servers usually do not serve the request replying instead with a 4xx HTTP status code (e.g. 429 or 403) or adopt more aggressive policies like dropping connections.

Quotas may be enforced on different basis (e.g. per user, per IP, per geographic area, ..) and at different levels. For example, an user may be allowed to issue:

*10 requests per second;

*limited to 60 requests per minute;

*limited to 1000 requests per hour.

Moreover system metrics, statistics and heuristics can be used to implement more complex policies, where the number of acceptable requests and the time window are computed dynamically.

To help clients throttling their requests, servers may expose the counters used to evaluate quota policies via HTTP header fields.

Those response headers may be added by HTTP intermediaries such as API gateways and reverse proxies.

On the web we can find many different rate-limit headers, usually containing the number of allowed requests in a given time window, and when the window is reset.

The common choice is to return three headers containing:

- the maximum number of allowed requests in the time window;

- the number of remaining requests in the current window;

- the time remaining in the current window expressed in seconds or as a timestamp;

A.1. Interoperability issues

A major interoperability issue in throttling is the lack of standard headers, because:

- each implementation associates different semantics to the same header field names;

- header field names proliferates.

User agents interfacing with different servers may thus need to process different headers, or the very same application interface that sits behind different reverse proxies may reply with different throttling headers.

Appendix B. Examples

B.1. Unparameterized responses

B.1.1. Throttling information in responses

The client exhausted its service-limit for the next 50 seconds. The time-window is communicated out-of-band or inferred by the field values.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example

```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 100
Ratelimit-Remaining: 0
Ratelimit-Reset: 50

{"hello": "world"}
```

Since the field values are not necessarily correlated with the response status code, a subsequent request is not required to fail. The example below shows that the server decided to serve the request even if RateLimit-Remaining header field value is 0. Another server, or the same server under other load conditions, could have decided to throttle the request instead.

Request:

```
GET /items/456 HTTP/1.1
Host: api.example

```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 100
Ratelimit-Remaining: 0
Ratelimit-Reset: 48

{"still": "successful"}
```

B.1.2. Use in conjunction with custom fields

The server uses two custom fields, namely acme-RateLimit-DayLimit and acme-RateLimit-HourLimit to expose the following policy:

*5000 daily quota units;

*1000 hourly quota units.

The client consumed 4900 quota units in the first 14 hours.

Despite the next hourly limit of 1000 quota units, the closest limit to reach is the daily one.

The server then exposes the RateLimit header fields to inform the client that:

*it has only 100 quota units left;

*the window will reset in 10 hours.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
acme-RateLimit-DayLimit: 5000
acme-RateLimit-HourLimit: 1000
RateLimit-Limit: 5000
RateLimit-Remaining: 100
RateLimit-Reset: 36000

{"hello": "world"}
```

B.1.3. Use for limiting concurrency

RateLimit header fields may be used to limit concurrency, advertising limits that are lower than the usual ones in case of saturation, thus increasing availability.

The server adopted a basic policy of 100 quota units per minute, and in case of resource exhaustion adapts the returned values reducing both RateLimit-Limit and RateLimit-Remaining header field values.

After 2 seconds the client consumed 40 quota units

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 100
RateLimit-Remaining: 60
RateLimit-Reset: 58

{"elapsed": 2, "issued": 40}
```

At the subsequent request - due to resource exhaustion - the server advertises only RateLimit-Remaining: 20.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 100
RateLimit-Remaining: 20
RateLimit-Reset: 56

{"elapsed": 4, "issued": 41}
```

B.1.4. Use in throttled responses

A client exhausted its quota and the server throttles it sending Retry-After.

In this example, the values of Retry-After and RateLimit-Reset header field reference the same moment, but this is not a requirement.

The 429 (Too Many Request) HTTP status code is just used as an example.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/json
Date: Mon, 05 Aug 2019 09:27:00 GMT
Retry-After: Mon, 05 Aug 2019 09:27:05 GMT
RateLimit-Reset: 5
RateLimit-Limit: 100
Ratelimit-Remaining: 0

{
"title": "Too Many Requests",
"status": 429,
"detail": "You have exceeded your quota"
}
```

B.2. Parameterized responses

B.2.1. Throttling window specified via parameter

The client has 99 quota units left for the next 50 seconds. The time window is communicated by the w parameter, so we know the throughput is 100 quota units per minute.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 100
RateLimit-Policy: 100;w=60
Ratelimit-Remaining: 99
Ratelimit-Reset: 50

{"hello": "world"}
```

B.2.2. Dynamic limits with parameterized windows

The policy conveyed by the RateLimit-Limit header field states that the server accepts 100 quota units per minute.

To avoid resource exhaustion, the server artificially lowers the actual limits returned in the throttling headers.

The RateLimit-Remaining header field then advertises only 9 quota units for the next 50 seconds to slow down the client.

Note that the server could have lowered even the other values in the RateLimit-Limit header field: this specification does not mandate any relation between the field values contained in subsequent responses.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 10
RateLimit-Policy: 100;w=60
Ratelimit-Remaining: 9
Ratelimit-Reset: 50

{
  "status": 200,
  "detail": "Just slow down without waiting."
}
```

B.2.3. Dynamic limits for pushing back and slowing down

Continuing the previous example, let's say the client waits 10 seconds and performs a new request which, due to resource exhaustion, the server rejects and pushes back, advertising RateLimit-Remaining: 0 for the next 20 seconds.

The server advertises a smaller window with a lower limit to slow down the client for the rest of its original window after the 20 seconds elapse.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/json
RateLimit-Limit: 0
RateLimit-Policy: 15;w=20
Ratelimit-Remaining: 0
Ratelimit-Reset: 20

{
  "status": 429,
  "detail": "Wait 20 seconds, then slow down!"
}
```

B.3. Dynamic limits for pushing back with Retry-After and slow down

Alternatively, given the same context where the previous example starts, we can convey the same information to the client via Retry-After, with the advantage that the server can now specify the policy's nominal limit and window that will apply after the reset, e.g. assuming the resource exhaustion is likely to be gone by then, so the advertised policy does not need to be adjusted, yet we managed to stop requests for a while and slow down the rest of the current window.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/json
Retry-After: 20
RateLimit-Limit: 15
RateLimit-Policy: 100;w=60
Ratelimit-Remaining: 15
Ratelimit-Reset: 40

{
  "status": 429,
  "detail": "Wait 20 seconds, then slow down!"
}
```

Note that in this last response the client is expected to honor Retry-After and perform no requests for the specified amount of time, whereas the previous example would not force the client to stop requests before the reset time is elapsed, as it would still be free to query again the server even if it is likely to have the request rejected.

B.3.1. Missing Remaining information

The server does not expose RateLimit-Remaining header field values (for example, because the underlying counters are not available). Instead, it resets the limit counter every second.

It communicates to the client the limit of 10 quota units per second always returning the couple RateLimit-Limit and RateLimit-Reset header field.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 10
Ratelimit-Reset: 1

{"first": "request"}
```

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Limit: 10
Ratelimit-Reset: 1

{"second": "request"}
```

B.3.2. Use with multiple windows

This is a standardized way of describing the policy detailed in Appendix B.1.2:

- 5000 daily quota units;

- 1000 hourly quota units.

The client consumed 4900 quota units in the first 14 hours.

Despite the next hourly limit of 1000 quota units, the closest limit to reach is the daily one.

The server then exposes the RateLimit header fields to inform the client that:

- it has only 100 quota units left;

- the window will reset in 10 hours;

- the expiring-limit is 5000.

Request:

```
GET /items/123 HTTP/1.1
Host: api.example
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
RateLimit-Limit: 5000
RateLimit-Policy: 1000;w=3600, 5000;w=86400
RateLimit-Remaining: 100
RateLimit-Reset: 36000

{"hello": "world"}
```

FAQ

This section is to be removed before publishing as an RFC.

1. Why defining standard fields for throttling?

   To simplify enforcement of throttling policies.

2. Can I use RateLimit header fields in throttled responses (eg with status code 429)?

   Yes, you can.

3. Are those specs tied to RFC 6585?

   No. [RFC6585] defines the 429 status code and we use it just as an example of a throttled request, that could instead use even 403 or whatever status code. The goal of this specification is to standardize the name and semantic of three RateLimit header fields widely used on the internet. Stricter relations with status codes or error response payloads would impose behaviors to all the existing implementations making the adoption more complex.

4. Why don't pass the throttling scope as a parameter?

   The word "scope" can have different meanings: for example it can be an URL, or an authorization scope. Since authorization is out of the scope of this document (see Section 1.1), and that we rely only on [HTTP], in Section 1.1 we defined "scope" in terms of URL.

   Since clients are not required to process quota policies (see Section 5), we could add a new "RateLimit-Scope" field to this spec. See this discussion on a similar thread

   Specific ecosystems can still bake their own prefixed parameters, such as acme-auth-scope or acme-url-scope and ensure that clients process them. This behavior cannot be relied upon when communicating between different ecosystems.

   We are open to suggestions: comment on this issue

5. Why using delay-seconds instead of a UNIX Timestamp? Why not using subsecond precision?

   Using delay-seconds aligns with Retry-After, which is returned in similar contexts, eg on 429 responses.

   Timestamps require a clock synchronization protocol (see Section 5.6.7 of [HTTP]). This may be problematic (e.g. clock adjustment, clock skew, failure of hardcoded clock synchronization servers, IoT devices, ..). Moreover timestamps may not be monotonically increasing due to clock adjustment. See Another NTP client failure story

   We did not use subsecond precision because:

   - that is more subject to system clock correction like the one implemented via the adjtimex() Linux system call;

   - response-time latency may not make it worth. A brief discussion on the subject is on the httpwg ml

   - almost all rate-limit headers implementations do not use it.

6. Why not support multiple quota remaining?

   While this might be of some value, my experience suggests that overly-complex quota implementations results in lower effectiveness of this policy. This spec allows the client to easily focusing on RateLimit-Remaining and RateLimit-Reset.

7. Shouldn't I limit concurrency instead of request rate?

   You can use this specification to limit concurrency at the HTTP level (see {#use-for-limiting-concurrency}) and help clients to shape their requests avoiding being throttled out.

   A problematic way to limit concurrency is connection dropping, especially when connections are multiplexed (e.g. HTTP/2) because this results in unserviced client requests, which is something we want to avoid.

   A semantic way to limit concurrency is to return 503 + Retry-After in case of resource saturation (e.g. thrashing, connection queues too long, Service Level Objectives not meet, ..). Saturation conditions can be either dynamic or static: all this is out of the scope for the current document.

8. Do a positive value of RateLimit-Remaining header field imply any service guarantee for my future requests to be served?

   No. FAQ integrated in Section 3.3.

9. Is the quota-policy definition Section 2.1 too complex?

   You can always return the simplest form of the 3 fields

```
RateLimit-Limit: 100
RateLimit-Remaining: 50
RateLimit-Reset: 60
```

The key runtime value is the first element of the list: expiring-limit, the others quota-policy are informative. So for the following field:

```
RateLimit-Limit: 100
RateLimit-Policy: 100;w=60;burst=1000;comment="sliding window", 5000;w=3
```

the key value is the one referencing the lowest limit: 100

1. Can we use shorter names? Why don't put everything in one field?

The most common syntax we found on the web is X-RateLimit-* and when starting this I-D we opted for it

The basic form of those fields is easily parseable, even by implementers processing responses using technologies like dynamic interpreter with limited syntax.

Using a single field complicates parsing and takes a significantly different approach from the existing ones: this can limit adoption.

1. Why don't mention connections?

   Beware of the term "connection": &#65532; &#65532; - it is just *one* possible saturation cause. Once you go that path &#65532; you will expose other infrastructural details (bandwidth, CPU, .. see Section 6.2) &#65532; and complicate client compliance; &#65532; - it is an infrastructural detail defined in terms of server and network &#65532; rather than the consumed service. This specification protects the services first, and then the infrastructures through client cooperation (see Section 6.1). &#65532; &#65532; RateLimit header fields enable sending *on the same connection* different limit values &#65532; on each response, depending on the policy scope (e.g. per-user, per-custom-key, ..) &#65532;

2. Can intermediaries alter RateLimit header fields?

   Generally, they should not because it might result in unserviced requests. There are reasonable use cases for intermediaries mangling RateLimit header fields though, e.g. when they enforce stricter quota-policies, or when they are an active component of the service. In those case we will consider them as part of the originating infrastructure.

3. Why the w parameter is just informative? Could it be used by a client to determine the request rate?

   A non-informative w parameter might be fine in an environment where clients and servers are tightly coupled. Conveying policies with this detail on a large scale would be very complex and implementations would be likely not interoperable. We thus decided to leave w as an informational parameter and only rely on RateLimit-Limit, RateLimit-Remaining header field and RateLimit-Reset header field for defining the throttling behavior.

4. Can I use RateLimit fields in trailers? Servers usually establish whether the request is in-quota before creating a response, so the RateLimit field values should be already available in that moment. Supporting trailers has the only advantage that allows to provide more up-to-date information to the client in case of slow responses. However, this complicates client implementations with respect to combining fields from

headers and accounting for intermediaries that drop trailers. Since there are no current implementations that use trailers, we decided to leave this as a future-work.

RateLimit header fields currently used on the web

This section is to be removed before publishing as an RFC.

Commonly used header field names are:

- X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset;
- X-Rate-Limit-Limit, X-Rate-Limit-Remaining, X-Rate-Limit-Reset.

There are variants too, where the window is specified in the header field name, eg:

- x-ratelimit-limit-minute, x-ratelimit-limit-hour, x-ratelimit-limit-day
- x-ratelimit-remaining-minute, x-ratelimit-remaining-hour, x-ratelimit-remaining-day

Here are some interoperability issues:

- X-RateLimit-Remaining references different values, depending on the implementation:
  - seconds remaining to the window expiration
  - milliseconds remaining to the window expiration
  - seconds since UTC, in UNIX Timestamp [UNIX]
  - a datetime, either IMF-fixdate [HTTP] or [RFC3339]
- different headers, with the same semantic, are used by different implementers:
  - X-RateLimit-Limit and X-Rate-Limit-Limit
  - X-RateLimit-Remaining and X-Rate-Limit-Remaining
  - X-RateLimit-Reset and X-Rate-Limit-Reset

The semantic of RateLimit-Remaining depends on the windowing algorithm. A sliding window policy for example may result in having a RateLimit-Remaining header field value related to the ratio between the current and the maximum throughput. e.g.

```
RateLimit-Limit: 12
RateLimit-Policy: 12;w=1
RateLimit-Remaining: 6          ; using 50% of throughput, that is 6 uni
RateLimit-Reset: 1
```

If this is the case, the optimal solution is to achieve

```
RateLimit-Limit: 12
RateLimit-Policy: 12;w=1
RateLimit-Remaining: 1          ; using 100% of throughput, that is 12 u
RateLimit-Reset: 1
```

At this point you should stop increasing your request rate.

Acknowledgements

Thanks to Willi Schoenborn, Alejandro Martinez Ruiz, Alessandro Ranellucci, Amos Jeffries, Martin Thomson, Erik Wilde and Mark Nottingham for being the initial contributors of these specifications. Kudos to the first community implementers: Aapo Talvensaari, Nathan Friedly and Sanyam Dogra.

In addition to the people above, this document owes a lot to the extensive discussion in the HTTPAPI workgroup, including Rich Salz, Darrel Miller and Julian Reschke.

Changes

This section is to be removed before publishing as an RFC.

Since draft-ietf-httpapi-ratelimit-headers-03

This section is to be removed before publishing as an RFC.

- Split policy informatio in RateLimit-Policy #81

Since draft-ietf-httpapi-ratelimit-headers-02

This section is to be removed before publishing as an RFC.

- Address throttling scope #83

Since draft-ietf-httpapi-ratelimit-headers-01

This section is to be removed before publishing as an RFC.

- Update IANA considerations #60

- Use Structured fields #58

- *Reorganize document #67

Since draft-ietf-httpapi-ratelimit-headers-00

This section is to be removed before publishing as an RFC.

- *Use I-D.httpbis-semantics, which includes referencing delay-seconds instead of delta-seconds. #5

Authors' Addresses

Roberto Polli
Team Digitale, Italian Government
Italy

Email: robipolli@gmail.com

Alejandro Martinez Ruiz
Red Hat

Email: alex@flawedcode.org