Workgroup: HTTPAPI
Internet-Draft:
draft-ietf-httpapi-ratelimit-headers-00
Published: 18 December 2020
Intended Status: Standards Track
Expires: 21 June 2021
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 fields for HTTP, thus allowing servers to publish current request quotas and clients to shape their request policy and avoid being throttled out.

Note to Readers

RFC EDITOR: please remove this section before publication

Discussion of this draft takes place on the HTTP working group mailing list (httpapi@ietf.org), which is archived at https://lists.w3.org/Archives/Public/ietf-httpapi-wg/.

The source code and issues list for this draft can be found at https://github.com/ietf-wg-httpapi/ratelimit-headers.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 21 June 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

The widespreading of HTTP as a distributed computation protocol requires an explicit way of communicating service status and usage quotas.

This was partially addressed with the Retry-After header field defined in [SEMANTICS] to be returned in 429 Too Many Requests or 503 Service Unavailable responses.

Still, there is not a standard way to communicate service quotas so that the client can throttle its requests and prevent 4xx or 5xx responses.

1.1. 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 - eg. monetization.

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

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

Quotas may be enforced on different basis (eg. 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 request per minute;

*limited to 1000 request per hour.

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

1.2. Current landscape of rate-limiting headers

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;

1.2.1. 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.

Client applications 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.

1.3. This proposal

This proposal defines syntax and semantics for the following fields:

*RateLimit-Limit: containing the requests quota in the time window;

*RateLimit-Remaining: containing the remaining requests quota in the current window;

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

The behavior of RateLimit-Reset is compatible with the delta-seconds notation of Retry-After.

The fields definition allows to describe complex policies, including the ones using multiple and variable time windows and dynamic quotas, or implementing concurrency limits.

1.4. Goals

The goals of this proposal are:

1. Standardizing the names and semantic of rate-limit headers;

2. Improve resiliency of HTTP infrastructures simplifying the enforcement and the adoption of rate-limit headers;

3. Simplify API documentation avoiding expliciting rate-limit fields semantic in documentation.

The goals do not include:

Authorization: The rate-limit headers described here 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 [RFC6454] section 7.

Response status code: The rate-limit headers may be returned in both Successful and non Successful responses. This specification does not cover whether non Successful responses count on quota usage.

Throttling policy: This specification does not mandate a specific throttling policy. The values published in the headers, 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.

1.5. 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 7 of [MESSAGING].

The term Origin is to be interpreted as described in [RFC6454] section 7.

The "delta-seconds" rule is defined in [CACHING] section 1.2.1.

2. Expressing rate-limit policies

2.1. Time window

Rate limit policies limit the number of acceptable requests in a given time window.

A time window is expressed in seconds, using the following syntax:

```
time-window = delta-seconds
```

Subsecond precision is not supported.

2.2. Request quota

The request-quota is a value associated to the maximum number of requests that the server is willing to accept from one or more clients on a given basis (originating IP, authenticated user, geographical, ..) during a time-window as defined in Section 2.1.

The request-quota is expressed in quota-units and has the following syntax:

```
request-quota = quota-units
quota-units = 1*DIGIT
```

The request-quota SHOULD match the maximum number of acceptable requests.

The request-quota MAY differ from the total number of acceptable requests when weight mechanisms, bursts, or other server policies are implemented.

If the request-quota does not match the maximum number of acceptable requests the relation with that SHOULD be communicated out-of-band.

Example: A server could

*count once requests like /books/{id}

*count twice search requests like /books?author=Camilleri

so that we have the following counters

```
GET /books/123                  ; request-quota=4, remaining: 3, status=
GET /books?author=Camilleri     ; request-quota=4, remaining: 1, status=
GET /books?author=Eco           ; request-quota=4, remaining: 0, status=
```

2.3. Quota policy

This specification allows describing a quota policy with the following syntax:

```
quota-policy = request-quota; "w" "=" time-window
               *( OWS ";" OWS quota-comment)
quota-comment = token "=" (token / quoted-string)
```

quota-policy parameters like w and quota-comment tokens MUST NOT occur multiple times within the same quota-policy.

An example policy of 100 quota-units per minute.

```
100;w=60
```

Two examples of providing further details via custom parameters in quota-comments.

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

3. Header Specifications

The following RateLimit response fields are defined

3.1. RateLimit-Limit

The RateLimit-Limit response field indicates the request-quota associated to the client in the current time-window.

If the client exceeds that limit, it MAY not be served.

The header value is

```
RateLimit-Limit = expiring-limit [, 1#quota-policy ]
expiring-limit = request-quota
```

The expiring-limit value MUST be set to the request-quota that is closer to reach its limit.

The quota-policy is defined in Section 2.3, and its values are informative.

```
RateLimit-Limit: 100
```

A time-window associated to expiring-limit can be communicated via an optional quota-policy value, like shown in the following example

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

If the expiring-limit is not associated to a time-window, the time-window MUST either be:

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

- communicated out-of-band (eg. in the documentation).

Policies using multiple quota limits MAY be returned using multiple quota-policy items, like shown in the following two examples:

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

This header MUST NOT occur multiple times and can be sent in a trailer section.

3.2. RateLimit-Remaining

The RateLimit-Remaining response field indicates the remaining quota-units defined in Section 2.2 associated to the client.

The header value is

```
RateLimit-Remaining = quota-units
```

This header MUST NOT occur multiple times and can be sent in a trailer section.

Clients MUST NOT assume that a positive RateLimit-Remaining value is a guarantee of being served.

A low RateLimit-Remaining value is like a yellow traffic-light: the red light may arrive suddenly.

One example of RateLimit-Remaining use is below.

RateLimit-Remaining: 50

3.3. RateLimit-Reset

The RateLimit-Reset response field indicates either

- the number of seconds until the quota resets.

The header value is

RateLimit-Reset = delta-seconds

The delta-seconds format is used because:

- it does not rely on clock synchronization and is resilient to clock adjustment and clock skew between client and server (see [SEMANTICS] Section 4.1.1.1);

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

This header MUST NOT occur multiple times and can be sent in a trailer section.

An example of RateLimit-Reset use is below.

RateLimit-Reset: 50

The client MUST NOT assume that all its request-quota will be restored after the moment referenced by RateLimit-Reset. The server MAY arbitrarily alter the RateLimit-Reset value between subsequent requests eg. in case of resource saturation or to implement sliding window policies.

4. Providing RateLimit headers

A server MAY use one or more RateLimit response fields defined in this document to communicate its quota policies.

The returned values refers to the metrics used to evaluate if the current request respects the quota policy and MAY not apply to subsequent requests.

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. Server metrics may be subject to other conditions like the one shown in the example from Section 2.2.

A server MAY return RateLimit response fields independently of the response status code. This includes throttled responses.

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

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

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

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. Nonetheless servers MAY decide to send the RateLimit fields in a trailer section.

5. Intermediaries

This section documents the considerations advised in Section 15.3.3 of [SEMANTICS].

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 fields' values in such a way as to communicate a more permissive quota-policy; this includes removing the RateLimit fields.

An intermediary MAY alter the RateLimit 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 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.

6. Caching

As is the ordinary case for HTTP caching ([RFC7234]), a response with RateLimit fields might be cached and re-used for subsequent requests. A cached RateLimit response, does not modify quota counters but could contain stale information. Clients interested in determining the freshness of the RateLimit fields could rely on fields such as Date and on the window value of a quota-policy.

7. Receiving RateLimit headers

A client MUST process the received RateLimit headers.

A client MUST validate the values received in the RateLimit headers before using them and check if there are significant discrepancies with the expected ones. This includes a RateLimit-Reset moment too far in the future or a request-quota too high.

Malformed RateLimit headers MAY be ignored.

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

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

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

The quota-policy values and comments provided in RateLimit-Limit are informative and MAY be ignored.

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

8. Examples

8.1. Unparameterized responses

8.1.1. Throttling informations in responses

The client exhausted its request-quota for the next 50 seconds. The time-window is communicated out-of-band or inferred by the header values.

Request:

```
GET /items/123
```

Response:

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

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

8.1.2. Use in conjunction with custom headers

The server uses two custom headers, 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-* headers to inform the client that:

- *it has only 100 quota-units left;
- *the window will reset in 10 hours.

Request:

```
GET /items/123
```

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"}
```

8.1.3. Use for limiting concurrency

Throttling headers 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.

After 2 seconds the client consumed 40 quota-units

Request:

```
GET /items/123
```

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
```

Response:

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

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

8.1.4. Use in throttled responses

A client exhausted its quota and the server throttles the request sending the Retry-After response header field.

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

The 429 Too Many Requests HTTP status code is just used as an example.

Request:

```
GET /items/123
```

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"
}
```

8.2. Parameterized responses

8.2.1. 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
```

Response:

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

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

8.2.2. Dynamic limits with parameterized windows

The policy conveyed by RateLimit-Limit 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 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 RateLimit-Limit: this specification does not mandate any relation between the field values contained in subsequent responses.

Request:

```
GET /items/123
```

Response:

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

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

8.2.3. 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
```

Response:

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

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

8.3. 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 the Retry-After header, with the advantage that the server can now specify the policy's nominal limit and window that will apply after the reset, ie. 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
```

Response:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/json
Retry-After: 20
RateLimit-Limit: 15, 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 the Retry-After header 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.

8.3.1. Missing Remaining informations

The server does not expose RateLimit-Remaining values, but 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.

Request:

```
GET /items/123
```

Response:

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

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

Request:

```
GET /items/123
```

Response:

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

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

8.3.2. Use with multiple windows

This is a standardized way of describing the policy detailed in Section 8.1.2:

- 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 headers 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
```

Response:

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

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

9. Security Considerations

9.1. Throttling does not prevent clients from issuing requests

This specification does not prevent clients to make over-quota requests.

Servers should always implement mechanisms to prevent resource exhaustion.

9.2. Information disclosure

Servers should not disclose operational capacity informations that can be used to saturate its resources.

While this specification does not mandate whether non 2xx responses consume quota, if 401 and 403 responses count on quota a malicious client could probe the endpoint to get traffic informations of another user.

As intermediaries might retransmit requests and consume quota-units without prior knowledge of the User Agent, RateLimit headers might reveal the existence of an intermediary to the User Agent.

9.3. Remaining quota-units are not granted requests

RateLimit-* headers convey hints from the server to the clients in order to avoid being throttled out.

Clients MUST NOT consider the quota-units returned in RateLimit-Remaining as a service level agreement.

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

9.4. Reliability of RateLimit-Reset

Consider that request-quota may not be restored after the moment referenced by RateLimit-Reset, and the RateLimit-Reset value should not be considered fixed nor constant.

Subsequent requests may return an higher RateLimit-Reset value to limit concurrency or implement dynamic or adaptive throttling policies.

9.5. Resource exhaustion

When returning RateLimit-Reset 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 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 adding some jitter to the field-value.

9.6. Denial of Service

RateLimit fields may assume unexpected values by chance or purpose. For example, an excessively high RateLimit-Remaining 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 an high RateLimit-Reset value could inhibit clients to contact the server.

Clients MUST validate the received values to mitigate those risks.

10. IANA Considerations

10.1. RateLimit-Limit Field Registration

This section registers the RateLimit-Limit field in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" registry ([SEMANTICS]).

Field name: RateLimit-Limit

Status: permanent

Specification document(s): Section 3.1 of this document

10.2. RateLimit-Remaining Field Registration

This section registers the RateLimit-Remaining field in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" registry ([SEMANTICS]).

Field name: RateLimit-Remaining

Status: permanent

Specification document(s): Section 3.2 of this document

10.3. RateLimit-Reset Field Registration

This section registers the RateLimit-Reset field in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" registry ([SEMANTICS]).

Field name: RateLimit-Reset

Status: permanent

Specification document(s): Section 3.3 of this document

11. References

11.1. Normative References

[CACHING] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[MESSAGING] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[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/info/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/info/rfc5234>.

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

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/info/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/info/rfc8174>.

[SEMANTICS] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

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

11.2. Informative References

[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/info/rfc3339>.

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

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

Appendix A. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

Appendix B. 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 implementors: Aapo Talvensaari, Nathan Friedly and Sanyam Dogra.

Appendix C. RateLimit headers currently used on the web

RFC EDITOR PLEASE DELETE THIS SECTION.

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
  - -a datetime, either IMF-fixdate [SEMANTICS] 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 value related to the ratio between the current and the maximum throughput. Eg.

```
RateLimit-Limit: 12, 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, 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.

Appendix D. FAQ

1. Why defining standard headers for throttling?

   To simplify enforcement of throttling policies.

2. Can I use RateLimit-* 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.

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

   After a discussion on a similar thread we will probably add a new "RateLimit-Scope" header to this spec.

   I'm open to suggestions: comment on this issue

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

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

   delta-seconds as defined in [CACHING] section 1.2.1 clarifies some parsing rules too.

   Timestamps require a clock synchronization protocol (see [SEMANTICS] section 4.1.1.1). This may be problematic (eg. 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 (eg. 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 (eg. 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 imply any service guarantee for my future requests to be served?

   No. The returned values were used to decide whether to serve or not *the current request* and do not imply any guarantee that future requests will be successful.

   Instead they help to understand when future requests will probably be throttled. A low value for RateLimit-Remaining should be interpreted as a yellow traffic-light for either the number of requests issued in the time-window or the request throughput.

9. Is the quota-policy definition Section 2.3 too complex?

   You can always return the simplest form of the 3 headers

```
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 header:

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

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

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

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 headers is easily parseable, even by implementors procesing responses using technologies like dynamic interpreter with limited syntax.

Using a single header 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": ￼ ￼ - it is just *one* possible saturation cause. Once you go that path ￼ you will expose other infrastructural details (bandwidth, CPU, .. see Section 9.2) ￼ and complicate client compliance; ￼ - it is an infrastructural detail defined in terms of server and network ￼ rather than the consumed service. This specification protects the services first, and then the infrastructures through client cooperation (see Section 9.1). ￼ ￼ RateLimit headers enable sending *on the same connection* different limit values ￼ on each response, depending on the policy scope (eg. per-user, per-custom-key, ..) ￼

2. Can intermediaries alter RateLimit fields?

   Generally, they should not because it might result in unserviced requests. There are reasonable use cases for intermediaries mangling RateLimit 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.

Authors' Addresses

Roberto Polli
Team Digitale, Italian Government

Email: robipolli@gmail.com

Alejandro Martinez Ruiz
Red Hat

Email: amr@redhat.com