HTTPAPI
Internet-Draft
Intended status: Standards Track
Expires: 19 September 2025

R. Polli
Team Digitale, Italian Government
A. Martinez
Red Hat
D. Miller
Microsoft

18 March 2025

RateLimit header fields for HTTP
draft-ietf-httpapi-ratelimit-headers-09

Abstract

This document defines the RateLimit-Policy and RateLimit HTTP header fields for servers to advertise their quota policies and the current service 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 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/.

Copyright Notice

Table of Contents

1. Introduction

Rate limiting of 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 their requests to prevent errors. This document defines a set of standard HTTP header fields to enable rate limiting:

* RateLimit-Policy: a quota policy, defined by the server, that client HTTP requests will consume.

* RateLimit: the currently remaining quota available for a specific policy.

These fields enable establishing complex rate limiting policies, including using multiple and variable time windows and dynamic quotas, and implementing concurrency limits.

1.1. Goals

The goals of this document are:

Interoperability: Standardize 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.

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 does it mandates any correlation between the RateLimit values and the returned status code.

Throttling algorithm: This specification does not mandate a specific throttling algorithm. 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.

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 term "problem type" in this document is to be interpreted as described in [PROBLEM].

2. Terminology

Quota: A quota is an allocation of capacity used by a resource server to limit client requests. That capacity is measured in quota units and may be reallocated at the end of a time window.

Quota Unit: A quota unit is the unit of measurement used to measure the activity of a client.

Quota Partition: A quota partition is a division of a server’s capacity across different clients, users and owned resources.

Time Window: A time window indicates a period of time associated to the allocated quota.

Quota Policy: A quota policy is implemented by the server to regulate the activity within a specified quota partition, quantified in quota units, over a defined time window. This activity is restricted to a predefined limit, known as the quota. Quota policies can be advertised by servers, but they are not required to be, and more than one quota policy can affect a given request from a client to a server.

Service Limit: A service limit is the currently remaining quota from a specific quota policy and, if defined, the remaining time before quota is reallocated.

List: A [STRUCTURED-FIELDS] list of Items

Item: A [STRUCTURED-FIELDS] item with a set of associated parameters

3. RateLimit-Policy Field

The "RateLimit-Policy" response header field is a non-empty List[RFC8941] of Quota Policy Items (Section 3.1). The Item[RFC8941] value MUST be a String[RFC8941].

The field value SHOULD remain consistent over a sequence of HTTP responses. It is this characteristic that differentiates it from the RateLimit (Section 4) field that contains information that MAY change on every request. The "RateLimit-Policy" field enables clients to control their own flow of requests based on policy information provided by the server. Situations where throttling constraints are highly dynamic are better served using the (RateLimit field)[{#ratelimit-field}] that communicates the latest service information a client can react to. Both fields can be communicated by the server when appropriate.

Lists of Quota Policy Items (Section 3.1) can be split over multiple "RateLimit-Policy" fields in the same HTTP response as described in Section 3.1 of [STRUCTURED-FIELDS].

```
RateLimit-Policy: "burst";q=100;w=60,"daily";q=1000;w=86400
```

3.1. Quota Policy Item

A quota policy Item contains an identifier for the policy and a set of Parameters[RFC8941] that contain information about a server’s capacity allocation for the policy.

The following parameters are defined:

q: The REQUIRED "q" parameter indicates the quota allocated by this

policy measured in quota units.

qu: The OPTIONAL "qu" parameter value conveys the quota units associated to the "q" parameter. The default quota unit is "requests".

w: The OPTIONAL "w" parameter value conveys a time window.

pk: The OPTIONAL "pk" parameter value conveys the partition key associated to the corresponding request.

Other parameters are allowed and can be regarded as comments.

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

This field MUST NOT appear in a trailer section.

3.1.1. Quota Parameter

The "q" parameter value MUST be a non-negative Integer. The value indicates the quota allocated for client activity (measured in quota units) for a given quota partition.

3.1.2. Quota Unit Parameter

The "qu" parameter value conveys the quota units applicable to the quota (Section 3.1.1). The value MUST be a String. Allowed values are listed in the RateLimit Quota Units registry (Section 10.3). This specification defines three quota units:

requests: This value indicates the quota is based on the number of requests processed by the resource server. Whether a specific request actually consumes a quota unit is implementation-specific.

content-bytes: This value indicates the quota is based on the number of content bytes processed by the resource server.

concurrent-requests: This value indicates the quota is based on the number of concurrent requests processed by the resource server.

3.1.3. Window Parameter

The "w" parameter value conveys a time window applicable to the quota (Section 3.1.1). The time window MUST be a non-negative, non-zero, Integer value expressing an interval in seconds, similar to the "delay-seconds" rule defined in Section 10.2.3 of [HTTP]. Sub-second precision is not supported.

3.1.4. Partition Key Parameter

The "pk" parameter value conveys the partition key associated to the request. The value MUST be a Byte Sequence. Servers MAY use the partition key to divide server capacity across different clients and resources. Quotas are allocated per partition key.

3.2. RateLimit Policy Field Examples

This field MAY convey the time window associated with the quota, as shown in this example:

```
RateLimit-Policy: "default";q=100;w=10
```

These examples show multiple policies being returned:

```
RateLimit-Policy: "permin";q=50;w=60,"perhr";q=1000;w=3600
```

The following example shows a policy with a partition key:

```
RateLimit-Policy: "peruser";q=100;w=60;pk=:cHsdsRa894==:
```

The following example shows a policy with a partition key and a quota unit:

```
RateLimit-Policy: "peruser";q=65535;qu="content-bytes";w=10;pk=:sdfjLJUOUH==:
```

4. RateLimit Field

A server uses the "RateLimit" response header field to communicate the current service limit for a quota policy for a particular partition key.

The field is expressed as a List[RFC8941] of Service Limit Items (Section 4.1).

Lists of Service Limit Items can be split over multiple "RateLimit" fields in the same HTTP response as described in Section 3.1 of [STRUCTURED-FIELDS].

```
RateLimit: "default";r=50;t=30
```

4.1. Service Limit Item

Each service limit Item[RFC8941] identifies the quota policy (Section 3.1) associated with the request and contains Parameters[RFC8941] with information about the current service limit.

The following parameters are defined in this specification:

r: This REQUIRED parameter value conveys the remaining quota units for the identified policy (Section 4.1.1).

t: This OPTIONAL parameter value conveys the time window reset time for the identified policy (Section 4.1.2).

pk: The OPTIONAL "pk" parameter value conveys the partition key associated to the corresponding request.

This field cannot appear in a trailer section. Other parameters are allowed and can be regarded as comments.

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

4.1.1. Remaining Parameter

The "r" parameter indicates the remaining quota units for the identified policy (Section 4.1.1).

It is a non-negative Integer expressed in quota units. Clients MUST NOT assume that a positive remaining value is a guarantee that further requests will be served. When the remaining parameter value is low, it indicates that the server may soon throttle the client (see Section 6).

4.1.2. Reset Parameter

The "t" parameter indicates the number of seconds until the quota associated with the quota policy resets.

It 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.

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

4.1.3. Partition Key Parameter

The "pk" parameter value conveys the partition key associated to the request. The value MUST be a Byte Sequence. Servers MAY use the partition key to divide server capacity across different clients and resources. Quotas are allocated per partition key.

4.2. RateLimit Field Examples

This example shows a RateLimit field with a remaining quota of 50 units and a time window reset in 30 seconds:

```
RateLimit: "default";r=50;t=30
```

This example shows a remaining quota of 999 requests for a partition key that has no time window reset:

```
RateLimit: "default";r=999;pk=:dHJpYWwxMjEzMjM=:
```

This example shows a 300MB remaining quota for an application in the next 60 seconds:

```
RateLimit: "default";r=300000000;t=60;pk=:QXBwLTk5OQ==:
```

5. Problem Types

5.1. Quota Exceeded

This section defines the "https://iana.org/assignments/http-problem-types#quota-exceeded" problem type. A server MAY use this problem type if it wants to communicate to the client that the requests sent by the client exceed one or more Quota Policies. This problem type defines the extension member "violated-policies" as an array of strings, whose value is the names of policies where the quota was exceeded.

```
HTTP/1.1 429 Bad Request
Content-Type: application/problem+json

{
  "type": "https://iana.org/assignments/http-problem-types#quota-exceeded",
  "title": "Request cannot be satisifed as assigned quota has been exceeded",
  "violated-policies": ["daily","bandwidth"]
}
```

5.2. Temporary Reduced Capacity

This section defines the "https://iana.org/assignments/http-problem-types#temporary-reduced-capacity" problem type. A server MAY use this problem type if it wants to communicate to the client that the requests sent by the client exceed cannot currently be satisfied due to a temporary reduction in capacity due to service limitations. The server MAY chose to include a RateLimit-Policy field indicating the new temporarily lower quota. This problem type defines the extension member "violated-policies" as an array of strings, whose value is the names of policies where the quota was exceeded.

```
HTTP/1.1 503 Server Unavailable
Content-Type: application/problem+json

{
  "type": "https://iana.org/assignments/http-problem-types#temporary-reduced-capacity"
,
  "title": "Request cannot be satisifed due to temporary server capacity constraints",
  "violated-policies": ["hourly"]
}
```

5.3. Abnormal Usage Detected

This section defines the "https://iana.org/assignments/http-problem-types#abnormal-usage-detected" problem type. A server MAY use this problem type to communicate to the client that it has detected a pattern of requests that suggest unintentional or malicous behaviour on the part of the client. This problem type defines the extension member "violated-policies" as an array of strings, whose value is the names of policies where the quota was exceeded.

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/problem+json

{
  "type": "https://iana.org/assignments/http-problem-types#abnormal-usage-detected",
  "title": "Request not satisifed due to detection of abnormal request pattern",
  "violated-policies": ["hourly"]
}
```

6. Server Behavior

A server MAY return RateLimit header fields independently of the response status code. This includes 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 remaining parameter 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 remaining parameter value is 0.

```
HTTP/1.1 301 Moved Permanently
Location: /foo/123
RateLimit: "problemPolicy";r=0;t=10
```

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

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 the remaining and the reset parameter values (see Section 8.5); 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.

6.1. Generating Partition Keys

Servers MAY choose to return partition keys that distinguish between quota allocated to different consumers or different resources. There are a wide range of strategies for partitioning server capacity, including per user, per application, per HTTP method, per resource, or some combination of those values. The server SHOULD document how the partition key is generated so that clients can predict the key value for a future request and determine if there is sufficient quota remaining to execute the request. Servers should avoid returning partition keys that contain sensitive information. Servers SHOULD only use information that is present in the request to generate the partition key.

6.2. 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 close to exhaustion.

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].

7. 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 requests will succeed.

For example, a successful response with the following fields:

```
RateLimit: "default";r=1;t=7
```

does not guarantee that the next request will be successful. Servers' behavior may be subject to other conditions.

A client is responsible for ensuring that RateLimit header field values returned cause reasonable client behavior with respect to throughput and latency (see Section 8.5 and Section 8.5.1).

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 remaining parameter before the time window expressed in the reset parameter.

The value of the reset parameter 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 reset parameter 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 and Retry-After fields, the Retry-After field MUST take precedence and the reset parameter MAY be ignored.

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

* slowing down or pre-emptively back-off their request rate when approaching quota limits;

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

7.1. Consuming Partition Keys

Partition keys are useful for a client if it is likely that single client will make requests that consume different quota allocations. E.g. a client making requests on behalf of different users or for different resources that have independent quota allocations.

If a server documents the partition key generation algorithm, clients MAY generate a partition key for a future request. Using this key, and comparing to the key returned by the server, the client can determine if there is sufficient quota remaining to execute the request.

For cases where the partition key generation algorithm of a server is unknown, clients MAY use heuristics to guess if a future request will be successful based on its similarity to previous requests.

7.2. 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 9) provide further guidance on intermediaries.

7.3. 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]).

8. Security Considerations

8.1. 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.

8.2. 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.

Where partition keys contain identifying information, either of the client application or the user, servers should be aware of the potential for impersonation and apply the appropriate security mechanisms.

8.3. 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 returned in the remaining parameter (Section 4.1.1) 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.

8.4. Reliability of the reset parameter

Consider that quota might not be restored after the moment referenced by the reset parameter (Section 4.1.2), and the reset parameter value may not be constant.

Subsequent requests might return a higher reset parameter value to limit concurrency or implement dynamic or adaptive throttling policies.

8.5. Resource exhaustion

When returning reset values, servers 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 reset parameter accordingly

```
Date: Tue, 15 Nov 1994 18:00:00 GMT
RateLimit: "daily";r=1;t=36400
```

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 reset 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-Policy: "somepolicy";q=10000;w=1000
RateLimit: "somepolicy";r=10000;t=10
```

A client implementing a simple ratio between remaining parameter and reset parameter could infer an average throughput of 1000 quota units per second, while the quota parameter 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 remaining parameter value or a higher reset parameter value. Moreover, complementing large time window quota policies with a short time window one mitigates those risks.

8.5.1. Denial of Service

RateLimit header fields may contain unexpected values by chance or on purpose. For example, an excessively high remaining parameter 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 reset parameter 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 reset parameter 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).

9. 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.

10. IANA Considerations

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

10.1. Update HTTP Field Name Registry

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

Field Name	Status	Specification
RateLimit	permanent	Section 4 of RFC nnnn
RateLimit-Policy	permanent	Section 3 of RFC nnnn

Table 1

10.2. Update HTTP Problem Type registry

IANA is asked to register the following entries in the "HTTP Problem Types" registry at https://www.iana.org/assignments/http-problem-types.

10.2.1. Registration of "quota-exceeded" Problem Type

Type URI: https://iana.org/assignments/http-problem-types#quota-exceeded

Title: Quota Exceeded

Recommended HTTP status code: 429

Reference: Section 5.1 of this document

10.2.2. Registration of "temporary-reduced-capacity" Problem Type

Type URI: https://iana.org/assignments/http-problem-types#temporary-reduced-capacity

Title: Temporary Reduced Capacity

Recommended HTTP status code: 503

Reference: Section 5.2 of this document

10.2.3. Registration of "abnormal-usage-detected" Problem Type

Type URI: https://iana.org/assignments/http-problem-types#abnormal-usage-detected

Title: Abnormal Usage Detected

Recommended HTTP status code: 429

Reference: Section 5.3 of this document

10.3. RateLimit quota unit registry

This specification establishes the registry "Hypertext Transfer Protocol (HTTP) RateLimit Quota Units" registry to be located at https://www.iana.org/assignments/http-ratelimit-quota-units. 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).

The registry has the following initial content:

Quota Unit	Reference	Notes
request	RFC nnnn	
content-bytes	RFC nnnn	
concurrent-requests	RFC nnnn	

Table 2

10.3.1. Registration Template

The registration template for the RateLimit Quota Units registry is as follows:

* Quota Unit: The name of the quota unit.

* Reference: A reference to the document that specifies the quota unit.

* Notes: Any additional notes about the quota unit.

11. References

11.1. 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>.

[PROBLEM] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/rfc/rfc9457>.

[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>.

[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>.

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

[STRUCTURED-FIELDS] Nottingham, M. and P. 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>.

11.2. 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, etc.) 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. Responses without defining policies

Some servers may not expose the policy limits in the RateLimit-Policy header field. Clients can still use the RateLimit header field to throttle their requests.

B.1.1. Throttling information in responses

The client exhausted its quota for the next 50 seconds. The limit and time-window is communicated out-of-band.

Request:

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

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit: "default";r=0;t=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 remaining parameter 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: "default";r=0;t=48

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

B.1.2. Multiple policies in response

The server uses two different policies to limit the client’s requests:

* 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 in the daily quota and the window will reset in 10 hours;

The server MAY choose to omit returning the hourly policy as it uses the same quota units as the daily policy and the daily policy is the one that is closest to being exhausted.

Request:

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

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit: "dayLimit";r=100;t=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 limit and remaining parameter 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-Policy: "basic";q=100;w=60
RateLimit: "basic";r=60;t=58

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

At the subsequent request - due to resource exhaustion - the server advertises only r=20.

Request:

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

Response:

```
HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Policy: "basic";q=100;w=60
RateLimit: "basic";r=20;t=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 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/problem+json
Date: Mon, 05 Aug 2019 09:27:00 GMT
Retry-After: Mon, 05 Aug 2019 09:27:05 GMT
RateLimit: "default";r=0;t=5

{
"type": "https://iana.org/assignments/http-problem-types#quota-exceeded"
"title": "Too Many Requests",
"status": 429,
"policy-violations": ["default"]
}
```

B.2. Responses with defined policies

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: "fixedwindow";r=99;t=50
RateLimit-Policy: "fixedwindow";q=100;w=60
{"hello": "world"}
```

B.2.2. Dynamic limits with parameterized windows

The policy conveyed by the RateLimit 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 remaining parameter 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 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-Policy: "dynamic";q=100;w=60
RateLimit: "dynamic";r=9;t=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 r=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-Policy: "dynamic";q=15;w=20
RateLimit: "dynamic";r=0;t=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-Policy: "dynamic";q=100;w=60
RateLimit: "dynamic";r=15;t=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 remaining 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 limit and reset parameters.

Request:

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

Response:

HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Policy: quota;q=100;w=1
RateLimit: quota;t=1

{"first": "request"}

Request:

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

Response:

HTTP/1.1 200 Ok
Content-Type: application/json
RateLimit-Policy: quota;q=10
RateLimit: quota;t=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-Policy: "hour";q=1000;w=3600, "day";q=5000;w=86400
RateLimit: "day";r=100;t=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 and enable clients to constraint their requests to avoid being throttled.

2. Can I use RateLimit header fields in throttled responses (e.g. 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 is the partition key necessary?

   Without a partition key, a server can effectively only have one scope (aka partition), which is impractical for most services, or it needs to communicate the scopes out-of-band. This prevents the development of generic connector code that can be used to prevent requests from being throttled. Many APIs rely on API keys, user identity or client identity to allocate quota. As soon as a single client processes requests for more than one partition, the client needs to know the corresponding partition key to properly track requests against allocated quota.

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, e.g. 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, etc.). Moreover timestamps may not be monotonically increasing due to clock adjustment. See Another NTP client failure story (https://community.ntppool.org/t/another-ntp-client-failure-story/1014/)

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 (https://lists.w3.org/Archives/Public/ietf-http-wg/2019JulSep/0202.html)

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

6. 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, etc.). Saturation conditions can be either dynamic or static: all this is out of the scope for the current document.

7. Do a positive value of remaining parameter imply any service guarantee for my future requests to be served?

   No. FAQ integrated in Section 4.1.1.

8. Is the quota-policy definition too complex?

You can always return the simplest form

```
RateLimit:"default";r=50;t=60
```

The policy key clearly connects the current usage status of a policy to the defined limits. So for the following field:

```
RateLimit-Policy: "sliding";q=100;w=60;burst=1000
RateLimit-Policy: "fixed";q=5000;w=3600;burst=0
RateLimit: "sliding";r=50;t=44
```

the value "sliding" identifies the policy being reported.

1. 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.

2. 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 likely be not interoperable. We thus decided to leave w as an informational parameter and only rely on the limit, remaining and reset parameters for defining the throttling behavior.

3. 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 it 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;

There are variants too, where the window is specified in the header field name, e.g.:

* 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 depends on the windowing algorithm. A sliding window policy for example, may result in having a remaining parameter value related to the ratio between the current and the maximum throughput. e.g.

```
RateLimit-Policy: "sliding";q=12;w=1
; using 50% of throughput, that is 6 units/s
RateLimit: "sliding";q=12;r=6;t=1
```

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

```
RateLimit-Policy: "sliding";q=12;w=1
; using 100% of throughput, that is 12 units/s
RateLimit: "sliding";q=12;r=1;t=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, and Julian Reschke.

Changes

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

Since draft-ietf-httpapi-ratelimit-headers-08

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

* Added Problem Types
* Clarified when to use RateLimit-Policy vs RateLimit fields

Since draft-ietf-httpapi-ratelimit-headers-07

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

* Refactored both fields to lists of Items that identify policy and use parameters
* Added quota unit parameter
* Added partition key parameter

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

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

* Split policy informatiom 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

Darrel Miller
Microsoft
Email: darrel@tavis.ca