DNSOP Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 8, 2019

R. Bellis
ISC
November 04, 2018

A DNS Resource Record for HTTP
draft-bellis-dnsop-http-record-00

Abstract

This document specifies an "HTTP" resource record type for the DNS to facilitate the lookup of the server hostname of HTTP(s) URIs. It is intended to replace the use of CNAME records for this purpose, and in the process provides a solution for the inability of the DNS to allow a CNAME to be placed at the apex of a domain name.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

It is very common for HTTP(s) URIs to contain a domain name that is not the same as the hostname of the actual server that hosts the content.

This is typically achieved via a CNAME record where the owner name of that record (the "Alias") is the domain name from the URI and the Canonical name field in its RDATA corresponds with the target hostname (although it should be noted that this strictly a violation of the original design semantics of the CNAME record).

It is also impossible to store a CNAME at the apex of a domain name, which causes signficant difficulties if you wish to redirect your domain name without a "www" prefix to a content delivery network (CDN). The only portable solution at the moment is to determine the IP address records of the content host and insert them directly at the apex of the zone, but this is brittle, and prevents the correct operation of typical CDN features.

While there have been previous attempts to promote the use of the SRV record instead of CNAME records, there have been concerns raised about the performance impact of the additional DNS lookup an SRV record would typically require.

To achieve equivalent end-user performance as existing CNAME-based solutions, this document permits recursive resolvers to pre-emptively look up the target of an HTTP Record and return the corresponding records to the client. While this feature is not mandatory it is hoped that support would over time become near ubiquitous.

Also, the presence of the Port field in an SRV record is incompatible with the "Same Origin" security policy enforced by web browsers and in practise the load-balancing / fallback capabilities of the SRV record are not widely used either, and non-DNS based solutions for this are already widely deployed for HTTP traffic.

This document therefore specifies a minimal "HTTP" resource record type for the DNS to facilitate the redirection from the domain name portion of an HTTP(s) URI to the server hostname and thence to A or AAAA records. It is specifically intended to replace the use of CNAME records for this purpose, and in the process provides a solution for the inability of the DNS to allow a CNAME to be placed at the apex of a domain name.

2. Terminology

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.

3. Description

The owner name of an HTTP RR is the domain name portion of an HTTP(s) URI.

The use of underscore label prefixes (e.g. _http._tcp) was considered, but rejected since it prohibits the use of wildcard records which us a valuable technique for offering per-customer domain prefixes without requiring that every prefix be individually provisioned.

3.1. Wire Format

The RDATA of an HTTP RR is a domain name in uncompressed wire format.

3.2. Presentation Format

The RDATA of an HTTP RR is presented as a domain name in standard master file format.

3.3. Server Operation

Recursive resolvers MAY on receiving a request for an HTTP record look up the A and AAAA records for the target (either from cache, or via new iterative queries) and include the results in the Additional Section of the response.

If the recursive resolver is performing DNSSEC resolution but is unable to validate the A or AAAA responses it MUST NOT include them in the response unless the client has specified the +CD (checking disabled) flag.

Where EDNS Client Subnet [RFC7871] is configured on the resolver those A and AAAA lookups MUST be performed as if the client had made those queries directly to the resolver.

3.4. Client Operation

HTTP clients supporting this specification MUST issue parallel DNS requests for the A, AAAA and HTTP records for the domain portion of an http: or https: URI.

If an HTTP record is returned, the client MUST either use the A and AAAA records contained in the Additional Section of the response, or issue further parallel requests for the A and AAAA records corresponding to the domain name in the RDATA of the HTTP record and then use those IP addresses to access the URI.

If the original A and AAAA lookups return IP addresses these MUST only be used if no HTTP record is returned.

<< the above needs more text around timing, happy eyeballs, etc. >>

4. Security Considerations

TBD

5. Implementation status

<< RFC Editor Note: Please remove this entire section prior to publication as an RFC. >>

6. Privacy Considerations

TBD (if any)

7. IANA Considerations

<< a copy of the RFC 6895 IANA RR TYPE application template will appear here >>

8. Acknowledgements

9. References

9.1. Normative References

[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>.

[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>.

9.2. Informative References

[RFC7871] Contavalli, C., van der Gaast, W., Lawrence, D., and W. Kumari, "Client Subnet in DNS Queries", RFC 7871, DOI 10.17487/RFC7871, May 2016, <https://www.rfc-editor.org/info/rfc7871>.

Author’s Address

Ray Bellis
Internet Systems Consortium, Inc.
950 Charter Street
Redwood City CA 94063
USA

Phone: +1 650 423 1200
Email: ray@isc.org

HTTP M. Nottingham
Internet-Draft October 31, 2019
Obsoletes: 3205 (if approved)
Intended status: Best Current Practice
Expires: May 3, 2020

Building Protocols with HTTP
draft-ietf-httpbis-bcp56bis-09

Abstract

HTTP is often used as a substrate for other application protocols (a.k.a. HTTP-based APIs). This document specifies best practices for writing specifications that use HTTP to define new application protocols, especially when they are defined for diverse implementation and broad deployment (e.g., in standards efforts).

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/bcp56bis [3].

Status of This Memo

Table of Contents

1. Introduction

HTTP [I-D.ietf-httpbis-semantics] is often used as a substrate for applications other than Web browsing; this is sometimes referred to as creating "HTTP-based APIs", "REST APIs" or just "HTTP APIs". This is done for a variety of reasons, including:

- familiarity by implementers, specifiers, administrators, developers and users,

- availability of a variety of client, server and proxy implementations,

- ease of use,

- availability of Web browsers,

- reuse of existing mechanisms like authentication and encryption,

- presence of HTTP servers and clients in target deployments, and

- its ability to traverse firewalls.

These protocols are often ad hoc; they are intended for only deployment by one or a few servers, and consumption by a limited set of clients. As a result, a body of practices and tools has arisen around defining HTTP-based APIs that favours these conditions.

However, when such an application has multiple, separate implementations, is deployed on multiple uncoordinated servers, and is consumed by diverse clients - as is often the case for HTTP APIs defined by standards efforts - tools and practices intended for limited deployment can become unsuitable.

This is largely because implementations (both client and server) will implement and evolve at different paces. As a result, such an HTTP-based API will need to more carefully consider how extensibility of the service will be handled and how different deployment requirements will be accommodated.

More generally, application protocols using HTTP face a number of design decisions, including:

- Should it define a new URI scheme? Use new ports?
- Should it use standard HTTP methods and status codes, or define new ones?
- How can the maximum value be extracted from the use of HTTP?
- How does it coexist with other uses of HTTP - especially Web browsing?
- How can interoperability problems and "protocol dead ends" be avoided?

This document contains best current practices for the specification of such applications. Section 2 defines when it applies; Section 3 surveys the properties of HTTP that are important to preserve, and Section 4 conveys best practices for the specifying them.

It is written primarily to guide IETF efforts to define application protocols using HTTP for deployment on the Internet, but might be applicable in other situations. Note that the requirements herein do not necessarily apply to the development of generic HTTP extensions.

1.1. 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.

2. Is HTTP Being Used?

Different applications have different goals when using HTTP. The requirements in this document apply when a specification defines an application that:

- uses the transport port 80 or 443, or

- o uses the URI scheme "http" or "https", or
- o uses an ALPN protocol ID [RFC7301] that generically identifies HTTP (e.g., "http/1.1", "h2", "h2c"), or
- o updates or modifies the IANA registries defined for HTTP.

Additionally, when a specification is using HTTP, all of the requirements of the HTTP protocol suite are in force (including but not limited to [I-D.ietf-httpbis-semantics], [I-D.ietf-httpbis-cache], [I-D.ietf-httpbis-messaging], and [RFC7540]).

Note that this document is intended to apply to applications, not generic extensions to HTTP, which follow the requirements in the relevant specification. Furthermore, it is intended for applications defined by IETF specifications, although other standards organisations are encouraged to adhere to its requirements.

2.1. Non-HTTP Protocols

A specification might not use HTTP according to the criteria above and still define an application that relies upon HTTP in some manner. For example, an application might wish to avoid re-specifying parts of the message format, but change other aspects of the protocol's operation; or, it might want to use a different set of methods.

Doing so brings more freedom to modify protocol operations, but loses at least a portion of the benefits outlined above, as most HTTP implementations won't be easily adaptable to these changes, and as the protocol diverges from HTTP, the benefit of mindshare will be lost.

Such specifications MUST NOT use HTTP's URI schemes, transport ports, ALPN protocol IDs or IANA registries; rather, they are encouraged to establish their own.

3. What's Important About HTTP

This section examines the facets of the protocol that are important to consider when using HTTP to define an application protocol.

3.1. Generic Semantics

Much of the value of HTTP is in its generic semantics - that is, the protocol elements defined by HTTP are potentially applicable to every resource, not specific to a particular context. Application-specific semantics are best expressed in the payload; often in the body, but also in header fields.

This generic/application-specific split allows a HTTP message to be handled by software (e.g., HTTP servers, intermediaries, client implementations, and caches) without understanding the specific application. It also allows people to leverage their knowledge of HTTP semantics without special-casing them for a particular application.

Therefore, applications that use HTTP MUST NOT re-define, refine or overlay the semantics of generic protocol elements such as methods, status codes or existing header fields. Instead, they should focus their specifications on protocol elements that are specific to that application; namely their HTTP resources.

For example, when writing a specification, it's often tempting to specify exactly how HTTP is to be implemented, supported and used.

However, this can easily lead to an unintended profile of HTTP's behaviour. For example, it's common to see specifications with language like this:

A 'POST' request MUST result in a '201 Created' response.

This forms an expectation in the client that the response will always be "201 Created", when in fact there are a number of reasons why the status code might differ in a real deployment; for example, there might be a proxy that requires authentication, or a server-side error, or a redirection. If the client does not anticipate this, the application's deployment is brittle.

See Section 4.2 for more details.

3.2. Links

Another common practice is assuming that the HTTP server's name space (or a portion thereof) is exclusively for the use of a single application. This effectively overlays special, application-specific semantics onto that space, precludes other applications from using it.

As explained in [RFC7320], such "squatting" on a part of the URL space by a standard usurps the server's authority over its own resources, can cause deployment issues, and is therefore bad practice in standards.

Instead of statically defining URI components like paths, it is RECOMMENDED that applications using HTTP define links in payloads, to allow flexibility in deployment.

Using runtime links in this fashion has a number of other benefits - especially when an application is to have multiple implementations and/or deployments (as is often the case for those that are standardised).

For example, navigating with a link allows a request to be routed to a different server without the overhead of a redirection, thereby supporting deployment across machines well.

It also becomes possible to "mix and match" different applications on the same server, and offers a natural mechanism for extensibility, versioning and capability management, since the document containing the links can also contain information about their targets.

Using links also offers a form of cache invalidation that’s seen on the Web; when a resource’s state changes, the application can change its link to it so that a fresh copy is always fetched.

3.3. Rich Functionality

HTTP offers a number of features to applications, such as:

- Message framing
- Multiplexing (in HTTP/2)
- Integration with TLS
- Support for intermediaries (proxies, gateways, Content Delivery Networks)
- Client authentication
- Content negotiation for format, language, and other features
- Caching for server scalability, latency and bandwidth reduction, and reliability
- Granularity of access control (through use of a rich space of URLs)
- Partial content to selectively request part of a response

- o The ability to interact with the application easily using a Web browser

Applications that use HTTP are encouraged to utilise the various features that the protocol offers, so that their users receive the maximum benefit from it, and to allow it to be deployed in a variety of situations. This document does not require specific features to be used, since the appropriate design tradeoffs are highly specific to a given situation. However, following the practices in Section 4 is a good starting point.

4. Best Practices for Specifying the Use of HTTP

This section contains best practices for specifying the use of HTTP by applications, including practices for specific HTTP protocol elements.

4.1. Specifying the Use of HTTP

When specifying the use of HTTP, an application should use [I-D.ietf-httpbis-semantics] as the primary reference; it is not necessary to reference all of the specifications in the HTTP suite unless there are specific reasons to do so (e.g., a particular feature is called out).

Because it is a hop-by-hop protocol, a HTTP connection can be handled by implementations that are not controlled by the application; for example, proxies, CDNs, firewalls and so on. Requiring a particular version of HTTP makes it difficult to use in these situations, and harms interoperability for little reason (since HTTP's semantics are stable between protocol versions). Therefore, it is RECOMMENDED that applications using HTTP not specify a minimum version of HTTP to be used.

However, if an application's deployment would benefit from the use of a particular version of HTTP (for example, HTTP/2's multiplexing), this ought be noted.

Applications using HTTP MUST NOT specify a maximum version, to preserve the protocol's ability to evolve.

When specifying examples of protocol interactions, applications should document both the request and response messages, with full headers, preferably in HTTP/1.1 format. For example:

```
GET /thing HTTP/1.1
Host: example.com
Accept: application/things+json
User-Agent: Foo/1.0


HTTP/1.1 200 OK
Content-Type: application/things+json
Content-Length: 500
Server: Bar/2.2

[payload here]
```

4.2. Specifying Server Behaviour

The most effective way to specify an application's server-side HTTP behaviours is in terms of the following protocol elements:

- o Media types [RFC6838], often based upon a format convention such as JSON [RFC8259],
- o HTTP header fields, as per Section 4.7, and
- o The behaviour of resources, as identified by link relations [RFC8288].

By composing these protocol elements, an application can define a set of resources, identified by link relations, that implement specified behaviours, including:

- o retrieval of their state using GET, in one or more formats identified by media type;
- o resource creation or update using POST or PUT, with an appropriately identified request body format;
- o data processing using POST and identified request and response body format(s); and
- o Resource deletion using DELETE.

For example, an application might specify:

Resources linked to with the "example-widget" link relation type are Widgets. The state of a Widget can be fetched in the "application/example-widget+json" format, and can be updated by PUT to the same link. Widget resources can be deleted.

The "Example-Count" response header field on Widget representations indicates how many Widgets are held by the sender.

The "application/example-widget+json" format is a JSON [RFC8259] format representing the state of a Widget. It contains links to related information in the link indicated by the Link header field value with the "example-other-info" link relation type.

Applications can also specify the use of URI Templates [RFC6570] to allow clients to generate URLs based upon runtime data.

4.3. Specifying Client Behaviour

In general, applications using HTTP ought to align their expectations for client behaviour as closely as possible with that of Web browsers, to avoid interoperability issues when they are used.

One way to do this is to define it in terms of [FETCH], since that is the abstraction that browsers use for HTTP.

Some client behaviours (e.g., automatic redirect handling) and extensions (e.g., Cookies) are not required by HTTP, but nevertheless have become very common. If their use is not explicitly specified by applications using HTTP, there may be confusion and interoperability problems. In particular:

- o Redirect handling - Applications need to specify how redirects are expected to be handled; see Section 4.6.1.

- o Cookies - Applications using HTTP should explicitly reference the Cookie specification [I-D.ietf-httpbis-rfc6265bis] if they are required.

- o Certificates - Applications using HTTP should specify that TLS certificates are to be checked according to [RFC2818] when HTTPS is used.

Applications using HTTP MUST NOT require HTTP features that are usually negotiated to be supported by clients. For example, requiring that clients support responses with a certain content-coding ([I-D.ietf-httpbis-semantics], Section 6.2.2) instead of negotiating for it ([I-D.ietf-httpbis-semantics], Section 8.4.4) means that otherwise conformant clients cannot interoperate with the application. Applications can encourage the implementation of such features, though.

4.4. Specifying URLs

In HTTP, the server resources that clients interact with are identified with URLs [RFC3986]. As [RFC7320] explains, parts of the URL are designed to be under the control of the owner (also known as the "authority") of that server, to give them the flexibility in deployment.

This means that in most cases, specifications for applications that use HTTP won't contain its URLs; while it is common practice for a specification of a single-deployment API to specify the path prefix "/app/v1" (for example), doing so in an IETF specification is inappropriate.

Therefore, the specification writer needs some mechanism to allow clients to discovery an application's URLs. Additionally, they need to specify what URL scheme(s) the application should be used with, and whether to use a dedicated port, or reuse HTTP's port(s).

4.4.1. Discovering an Application's URLs

Generally, a client will begin interacting with a given application server by requesting an initial document that contains information about that particular deployment, potentially including links to other relevant resources. Doing so assures that the deployment is as flexible as possible (potentially spanning multiple servers), allows evolution, and also gives the application the opportunity to tailor the 'discovery document' to the client.

There are a few common patterns for discovering that initial URL.

The most straightforward mechanism for URL discovery is to configure the client with (or otherwise convey to it) a full URL. This might be done in a configuration document, in DNS or mDNS, or through another discovery mechanism.

However, if the client only knows the server's hostname and the identity of the application, there needs to be some way to derive the initial URL from that information.

Applications MUST NOT define a fixed prefix for its URL paths; for reasons explained in [RFC7320], this is bad practice.

Instead, a specification for such an application can use one of the following strategies:

- o Register a Well-Known URI [I-D.nottingham-rfc5785bis] as an entry point for that application. This provides a fixed path on every potential server that will not collide with other applications.

- o Enable the server authority to convey a URL Template [RFC6570] or similar mechanism for generating a URL for an entry point. For example, this might be done in a DNS RR, a configuration document, or other artefact.

Once the discovery document is located, it can be fetched, cached for later reuse (if allowed by its metadata), and used to locate other resources that are relevant to the application, using full URIs or URL Templates.

In some cases, an application may not wish to use such a discovery document; for example, when communication is very brief, or when the latency concerns of doing so precludes the use of a discovery document. These situations can be addressed by placing all of the application's resources under a well-known location.

4.4.2. Considering URI Schemes

Applications that use HTTP will typically employ the "http" and/or "https" URI schemes. "https" is RECOMMENDED to provide authentication, integrity and confidentiality, as well as mitigate pervasive monitoring attacks [RFC7258].

However, application-specific schemes can also be defined. When defining an URI scheme for an application using HTTP, there are a number of tradeoffs and caveats to keep in mind:

- o Unmodified Web browsers will not support the new scheme. While it is possible to register new URI schemes with Web browsers (e.g. registerProtocolHandler() in [HTML], as well as several proprietary approaches), support for these mechanisms is not shared by all browsers, and their capabilities vary.

- o Existing non-browser clients, intermediaries, servers and associated software will not recognise the new scheme. For example, a client library might fail to dispatch the request; a cache might refuse to store the response, and a proxy might fail to forward the request.

- o Because URLs occur in HTTP artefacts commonly, often being generated automatically (e.g., in the "Location" response header), it can be difficult to assure that the new scheme is used consistently.

- o The resources identified by the new scheme will still be available using "http" and/or "https" URLs. Those URLs can "leak" into use, which can present security and operability issues. For example, using a new scheme to assure that requests don't get sent to a "normal" Web site is likely to fail.

- o Features that rely upon the URL's origin [RFC6454], such as the Web's same-origin policy, will be impacted by a change of scheme.

- o HTTP-specific features such as cookies [I-D.ietf-httpbis-rfc6265bis], authentication [I-D.ietf-httpbis-semantics], caching [I-D.ietf-httpbis-cache], HSTS [RFC6797], and CORS [FETCH] might or might not work correctly, depending on how they are defined and implemented. Generally, they are designed and implemented with an assumption that the URL will always be "http" or "https".

- o Web features that require a secure context [SECCTXT] will likely treat a new scheme as insecure.

See [RFC7595] for more information about minting new URI schemes.

4.4.3. Transport Ports

Applications can use the applicable default port (80 for HTTP, 443 for HTTPS), or they can be deployed upon other ports. This decision can be made at deployment time, or might be encouraged by the application's specification (e.g., by registering a port for that application).

If a non-default port is used, it needs to be reflected in the authority of all URLs for that resource; the only mechanism for changing a default port is changing the scheme (see Section 4.4.2).

Using a port other than the default has privacy implications (i.e., the protocol can now be distinguished from other traffic), as well as operability concerns (as some networks might block or otherwise interfere with it). Privacy implications should be documented in Security Considerations.

See [RFC7605] for further guidance.

4.5. Using HTTP Methods

Applications that use HTTP MUST confine themselves to using registered HTTP methods such as GET, POST, PUT, DELETE, and PATCH.

New HTTP methods are rare; they are required to be registered in the HTTP Method Registry with IETF Review (see [I-D.ietf-httpbis-semantics]), and are also required to be generic. That means that they need to be potentially applicable to all resources, not just those of one application.

While historically some applications (e.g., [RFC4791]) have defined non-generic methods, [I-D.ietf-httpbis-semantics] now forbids this.

When authors believe that a new method is required, they are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application's specification.

4.5.1. GET

GET is the most common and useful HTTP method; its retrieval semantics allow caching, side-effect free linking and underlies many of the benefits of using HTTP.

A common use of GET is to perform queries, often using the query component of the URL; this is a familiar pattern from Web browsing, and the results can be cached, improving efficiency of an often expensive process.

In some cases, however, GET might be unwieldy for expressing queries, because of the limited syntax of the URI; in particular, if binary data forms part of the query terms, it needs to be encoded to conform to URI syntax.

While this is not an issue for short queries, it can become one for larger query terms, or ones which need to sustain a high rate of requests. Additionally, some HTTP implementations limit the size of URLs they support - although modern HTTP software has much more generous limits than previously (typically, considerably more than 8000 octets, as required by [I-D.ietf-httpbis-semantics].

In these cases, an application using HTTP might consider using POST to express queries in the request body; doing so avoids encoding overhead and URL length limits in implementations. However, in doing so it should be noted that the benefits of GET such as caching and linking to query results are lost. Therefore, applications using HTTP that feel a need to allow POST queries ought consider allowing both methods.

Applications should not change their state or have other side effects that might be significant to the client, since implementations can and do retry HTTP GET requests that fail. Note that this does not include logging and similar functions; see [I-D.ietf-httpbis-semantics], Section 7.2.1.

Finally, note that while HTTP allows GET requests to have a body syntactically, this is done only to allow parsers to be generic; as per [I-D.ietf-httpbis-semantics], Section 7.3.1, a body on a GET has no meaning, and will be either ignored or rejected by generic HTTP software.

4.5.2. OPTIONS

The OPTIONS method was defined for metadata retrieval, and is used both by WebDAV [RFC4918] and CORS [FETCH]. Because HTTP-based APIs often need to retrieve metadata about resources, it is often considered for their use.

However, OPTIONS does have significant limitations:

- It isn't possible to link to the metadata with a simple URL, because OPTIONS is not the default GET method.

- OPTIONS responses are not cacheable, because HTTP caches operate on representations of the resource (i.e., GET and HEAD). If OPTIONS responses are cached separately, their interaction with HTTP cache expiry, secondary keys and other mechanisms needs to be considered.

- OPTIONS is "chatty" - always separating metadata out into a separate request increases the number of requests needed to interact with the application.

- Implementation support for OPTIONS is not universal; some servers do not expose the ability to respond to OPTIONS requests without significant effort.

Instead of OPTIONS, one of these alternative approaches might be more appropriate:

- For server-wide metadata, create a well-known URI [I-D.nottingham-rfc5785bis], or using an already existing one if it's appropriate (e.g., HostMeta [RFC6415]).

- For metadata about a specific resource, create a separate resource and link to it using a Link response header or a link serialised into the representation's body. See [RFC8288]. Note that the Link header is available on HEAD responses, which is useful if the client wants to discover a resource's capabilities before they interact with it.

4.6. Using HTTP Status Codes

HTTP status codes convey semantics both for the benefit of generic HTTP components - such as caches, intermediaries, and clients - and applications themselves. However, applications can encounter a number of pitfalls in their use.

First, status codes are often generated by intermediaries, as well as server and client implementations. This can happen, for example, when network errors are encountered, a captive portal is present, when an implementation is overloaded, or it thinks it is under attack. As a result, if an application assigns specific semantics to one of these status codes, a client can be misled about its state, because the status code was generated by a generic component, not the application itself.

Furthermore, mapping application errors to individual HTTP status codes one-to-one often leads to a situation where the finite space of applicable HTTP status codes is exhausted. This, in turn, leads to a number of bad practices - including minting new, application-specific status codes, or using existing status codes even though the link between their semantics and the application’s is tenuous at best.

Instead, applications using HTTP should define their errors to use the most applicable status code, making generous use of the general status codes (200, 400 and 500) when in doubt. Importantly, they should not specify a one-to-one relationship between status codes and application errors, thereby avoiding the exhaustion issue outlined above.

To distinguish between multiple error conditions that are mapped to the same status code, and to avoid the misattribution issue outlined above, applications using HTTP should convey finer-grained error information in the response’s message body and/or header fields. [RFC7807] provides one way to do so.

Because the set of registered HTTP status codes can expand, applications using HTTP should explicitly point out that clients ought to be able to handle all applicable status codes gracefully (i.e., falling back to the generic "n00" semantics of a given status code; e.g., "499" can be safely handled as "400" by clients that don’t recognise it). This is preferable to creating a "laundry list" of potential status codes, since such a list won’t be complete in the foreseeable future.

Applications using HTTP MUST NOT re-specify the semantics of HTTP status codes, even if it is only by copying their definition. It is RECOMMENDED they require specific reason phrases to be used; the reason phrase has no function in HTTP, is not guaranteed to be preserved by implementations, and is not carried at all in the HTTP/2 [RFC7540] message format.

Applications MUST only use registered HTTP status codes. As with methods, new HTTP status codes are rare, and required (by [I-D.ietf-httpbis-semantics]) to be registered with IETF Review. Similarly, HTTP status codes are generic; they are required (by [I-D.ietf-httpbis-semantics]) to be potentially applicable to all resources, not just to those of one application.

When authors believe that a new status code is required, they are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application's specification.

4.6.1. Redirection

The 3xx series of status codes specified in Section 9.4 [I-D.ietf-httpbis-semantics] direct the user agent to another resource to satisfy the request. The most common of these are 301, 302, 307 and 308, all of which use the Location response header field to indicate where the client should send the request to.

There are two ways that this group of status codes differ:

- Whether they are permanent or temporary. Permanent redirects can be used to update links stored in the client (e.g., bookmarks), whereas temporary ones can not. Note that this has no effect on HTTP caching; it is completely separate.

- Whether they allow the redirected request to change the request method from POST to GET. Web browsers generally do change POST to GET for 301 and 302; therefore, 308 and 307 were created to allow redirection without changing the method.

This table summarises their relationships:

	Permanent	Temporary
Allows changing the request method from POST to GET	301	302
Does not allow changing the request method	308	307

As noted in [I-D.ietf-httpbis-semantics], a user agent is allowed to automatically follow a 3xx redirect that has a Location response header field, even if they don't understand the semantics of the specific status code. However, they aren't required to do so; therefore, if an application using HTTP desires redirects to be automatically followed, it needs to explicitly specify the circumstances when this is required.

Applications using HTTP are encouraged to specify that 301 and 302 responses change the subsequent request method from POST (but no other method) to GET, to be compatible with browsers.

Generally, when a redirected request is made, its header fields are copied from the original request's. However, they can be modified by various mechanisms; e.g., sent Authorization ([I-D.ietf-httpbis-semantics]) and Cookie ([I-D.ietf-httpbis-rfc6265bis]) headers will change if the origin (and sometimes path) of the request changes. An application using HTTP should specify if any request headers that it defines need to be modified or removed upon a redirect; however, this behaviour cannot be relied upon, since a generic client (like a browser) will be unaware of such requirements.

4.7. Specifying HTTP Header Fields

Applications often define new HTTP header fields. Typically, using HTTP header fields is appropriate in a few different situations:

- o Their content is useful to intermediaries (who often wish to avoid parsing the body), and/or
- o Their content is useful to generic HTTP software (e.g., clients, servers), and/or
- o It is not possible to include their content in the message body (usually because a format does not allow it).

When the conditions above are not met, it is usually better to convey application-specific information in other places; e.g., the message body or the URL query string.

New header fields MUST be registered, as per [I-D.ietf-httpbis-semantics].

See [I-D.ietf-httpbis-semantics], Section 4.1.3 for guidelines to consider when minting new header fields. [I-D.ietf-httpbis-header-structure] provides a common structure for new header fields, and avoids many issues in their parsing and handling; it is RECOMMENDED that new header fields use it.

It is RECOMMENDED that header field names be short (even when HTTP/2 header compression is in effect, there is an overhead) but appropriately specific. In particular, if a header field is specific to an application, an identifier for that application can form a prefix to the header field name, separated by a "-".

For example, if the "example" application needs to create three headers, they might be called "example-foo", "example-bar" and "example-baz". Note that the primary motivation here is to avoid consuming more generic header names, not to reserve a portion of the namespace for the application; see [RFC6648] for related considerations.

The semantics of existing HTTP header fields MUST NOT be re-defined without updating their registration or defining an extension to them (if allowed). For example, an application using HTTP cannot specify that the "Location" header has a special meaning in a certain context.

See Section 4.9 for the interaction between headers and HTTP caching; in particular, request headers that are used to "select" a response have impact there, and need to be carefully considered.

See Section 4.10 for considerations regarding header fields that carry application state (e.g., Cookie).

4.8. Defining Message Payloads

There are many potential formats for payloads; for example, JSON [RFC8259], XML [XML], and CBOR [RFC7049]. Best practices for their use are out of scope for this document.

Applications should register distinct media types for each format they define; this makes it possible to identify them unambiguously and negotiate for their use. See [RFC6838] for more information.

4.9. Leveraging HTTP Caching

HTTP caching [I-D.ietf-httpbis-cache] is one of the primary benefits of using HTTP for applications; it provides scalability, reduces latency and improves reliability. Furthermore, HTTP caches are readily available in browsers and other clients, networks as forward and reverse proxies, Content Delivery Networks and as part of server software.

Even when an application using HTTP isn't designed to take advantage of caching, it needs to consider how caches will handle its responses, to preserve correct behaviour when one is interposed (whether in the network, server, client, or intervening infrastructure).

4.9.1. Freshness

Assigning even a short freshness lifetime (Section 4.2 of [I-D.ietf-httpbis-cache]) - e.g., 5 seconds - allows a response to be reused to satisfy multiple clients, and/or a single client making the same request repeatedly. In general, if it is safe to reuse something, consider assigning a freshness lifetime.

The most common method for specifying freshness is the max-age response directive (Section 5.2.2.8 of [I-D.ietf-httpbis-cache]). The Expires header (Section 5.3 of [I-D.ietf-httpbis-cache]) can also be used, but it is not necessary; all modern cache implementations support Cache-Control, and specifying freshness as a delta is usually more convenient and less error-prone.

In some situations, responses without explicit cache freshness directives will be stored and served using a heuristic freshness lifetime; see [I-D.ietf-httpbis-cache], Section 4.2.2. As the heuristic is not under control of the application, it is generally preferable to set an explicit freshness lifetime, or make the response explicitly uncacheable.

If caching of a response is not desired, the appropriate response directive is "Cache-Control: no-store". This only need be sent in situations where the response might be cached; see [I-D.ietf-httpbis-cache], Section 3. Note that "Cache-Control: no-cache" allows a response to be stored, just not reused by a cache without validation; it does not prevent caching (despite its name).

For example, this response cannot be stored or reused by a cache:

```
HTTP/1.1 200 OK
Content-Type: application/example+xml
Cache-Control: no-store

[content]
```

4.9.2. Stale Responses

Authors should understand that stale responses (e.g., with "Cache-Control: max-age=0") can be reused by caches when disconnected from the origin server; this can be useful for handling network issues.

If doing so is not suitable for a given response, the origin should use "Cache-Control: must-revalidate". See [I-D.ietf-httpbis-cache], Section 4.2.4, and also [RFC5861] for additional controls over stale content.

Stale responses can be refreshed by assigning a validator, saving both transfer bandwidth and latency for large responses; see [I-D.ietf-httpbis-semantics].

4.9.3. Caching and Application Semantics

When an application has a need to express a lifetime that's separate from the freshness lifetime, this should be conveyed separately, either in the response's body or in a separate header field. When this happens, the relationship between HTTP caching and that lifetime need to be carefully considered, since the response will be used as long as it is considered fresh.

In particular, application authors need to consider how responses that are not freshly obtained from the origin server should be handled; if they have a concept like a validity period, this will need to be calculated considering the age of the response (see Section 4.2.3 of [I-D.ietf-httpbis-cache]).

One way to address this is to explicitly specify that all responses be fresh upon use.

4.9.4. Varying Content Based Upon the Request

If an application uses a request header field to change the response's headers or body, authors should point out that this has implications for caching; in general, such resources need to either make their responses uncacheable (e.g., with the "no-store" cache-control directive defined in [I-D.ietf-httpbis-cache], Section 5.2.2.3) or send the Vary response header ([I-D.ietf-httpbis-semantics], Section 10.1.4) on all responses from that resource (including the "default" response).

For example, this response:

```
HTTP/1.1 200 OK
Content-Type: application/example+xml
Cache-Control: max-age=60
ETag: "sa0f8wf20fs0f"
Vary: Accept-Encoding

[content]
```

can be stored for 60 seconds by both private and shared caches, can be revalidated with If-None-Match, and varies on the Accept-Encoding request header field.

4.10. Handling Application State

Applications can use stateful cookies [I-D.ietf-httpbis-rfc6265bis] to identify a client and/or store client-specific data to contextualise requests.

When used, it is important to carefully specify the scoping and use of cookies; if the application exposes sensitive data or capabilities (e.g., by acting as an ambient authority), exploits are possible. Mitigations include using a request-specific token to assure the intent of the client.

Applications MUST NOT make assumptions about the relationship between separate requests on a single transport connection; doing so breaks many of the assumptions of HTTP as a stateless protocol, and will cause problems in interoperability, security, operability and evolution.

4.11. Client Authentication

Applications can use HTTP authentication [I-D.ietf-httpbis-semantics] to identify clients. The Basic authentication scheme [RFC7617] MUST NOT be used unless the underlying transport is authenticated, integrity-protected and confidential (e.g., as provided the "HTTPS" URI scheme, or another using TLS). The Digest scheme [RFC7616] MUST NOT be used unless the underlying transport is similarly secure, or the chosen hash algorithm is not "MD5".

With HTTPS, clients might also be authenticated using certificates [RFC5246].

When used, it is important to carefully specify the scoping and use of authentication; if the application exposes sensitive data or capabilities (e.g., by acting as an ambient authority), exploits are possible. Mitigations include using a request-specific token to assure the intent of the client.

4.12. Co-Existing with Web Browsing

Even if there is not an intent for an application to be used with a Web browser, its resources will remain available to browsers and other HTTP clients.

This means that all such applications that use HTTP need to consider how browsers will interact with them, particularly regarding security.

For example, if an application's state can be changed using a POST request, a Web browser can easily be coaxed into cross-site request forgery (CSRF) from arbitrary Web sites.

Or, if content returned from the application's resources is under control of an attacker (for example, part of the request is reflected in the response, or the response contains external information that might be under the control of the attacker), a cross-site scripting (XSS) attack is possible, whereby an attacker can inject code into the browser and access data and capabilities on that origin.

This is only a small sample of the kinds of issues that applications using HTTP must consider. Generally, the best approach is to consider the application actually as a Web application, and to follow best practices for their secure development.

A complete enumeration of such practices is out of scope for this document, but some considerations include:

- Using an application-specific media type in the Content-Type header, and requiring clients to fail if it is not used.

- Using X-Content-Type-Options: nosniff [FETCH] to assure that content under attacker control can't be coaxed into a form that is interpreted as active content by a Web browser.

- Using Content-Security-Policy [CSP] to constrain the capabilities of active content (such as HTML [HTML]), thereby mitigating Cross-Site Scripting attacks.

- Using Referrer-Policy [REFERRER-POLICY] to prevent sensitive data in URLs from being leaked in the Referer request header.

- Using the 'HttpOnly' flag on Cookies to assure that cookies are not exposed to browser scripting languages [I-D.ietf-httpbis-rfc6265bis].

- Avoiding use of compression on any sensitive information (e.g., authentication tokens, passwords), as the scripting environment offered by Web browsers allows an attacker to repeatedly probe the compression space; if the attacker has access to the path of the communication, they can use this capability to recover that information.

Depending on how they are intended to be deployed, specifications for applications using HTTP might require the use of these mechanisms in specific ways, or might merely point them out in Security Considerations.

An example of a HTTP response from an application that does not intend for its content to be treated as active by browsers might look like this:

```
HTTP/1.1 200 OK
Content-Type: application/example+json
X-Content-Type-Options: nosniff
Content-Security-Policy: default-src 'none'
Cache-Control: max-age=3600
Referrer-Policy: no-referrer

[content]
```

If an application has browser compatibility as a goal, client interaction ought to be defined in terms of [FETCH], since that is the abstraction that browsers use for HTTP; it enforces many of these best practices.

4.13. Maintaining Application Boundaries

Because the origin [RFC6454] is how many HTTP capabilities are scoped, applications also need to consider how deployments might interact with other applications (including Web browsing) on the same origin.

For example, if Cookies [I-D.ietf-httpbis-rfc6265bis] are used to carry application state, they will be sent with all requests to the origin by default, unless scoped by path, and the application might receive cookies from other applications on the origin. This can lead to security issues, as well as collision in cookie names.

One solution to these issues is to require a dedicated hostname for the application, so that it has a unique origin. However, it is often desirable to allow multiple applications to be deployed on a single hostname; doing so provides the most deployment flexibility and enables them to be "mixed" together (See [RFC7320] for details). Therefore, applications using HTTP should strive to allow multiple applications on an origin.

To enable this, when specifying the use of Cookies, HTTP authentication realms [I-D.ietf-httpbis-semantics], or other origin-wide HTTP mechanisms, applications using HTTP should not mandate the use of a particular name, but instead let deployments configure them.

Consideration should be given to scoping them to part of the origin, using their specified mechanisms for doing so.

Modern Web browsers constrain the ability of content from one origin to access resources from another, to avoid leaking private information. As a result, applications that wish to expose cross-origin data to browsers will need to implement the CORS protocol; see [FETCH].

4.14. Using Server Push

HTTP/2 adds the ability for servers to "push" request/response pairs to clients in [RFC7540], Section 8.2. While server push seems like a natural fit for many common application semantics (e.g., "fanout" and publish/subscribe), a few caveats should be noted:

- Server push is hop-by-hop; that is, it is not automatically forwarded by intermediaries. As a result, it might not work easily (or at all) with proxies, reverse proxies, and Content Delivery Networks.

- Server push can have negative performance impact on HTTP when used incorrectly; in particular, if there is contention with resources that have actually been requested by the client.

- Server push is implemented differently in different clients, especially regarding interaction with HTTP caching, and capabilities might vary.

- APIs for server push are currently unavailable in some implementations, and vary widely in others. In particular, there is no current browser API for it.

- Server push is not supported in HTTP/1.1 or HTTP/1.0.

- Server push does not form part of the "core" semantics of HTTP, and therefore might not be supported by future versions of the protocol.

Applications wishing to optimise cases where the client can perform work related to requests before the full response is available (e.g., fetching links for things likely to be contained within) might benefit from using the 103 (Early Hints) status code; see [RFC8297].

Applications using server push directly need to enforce the requirements regarding authority in [RFC7540], Section 8.2, to avoid cross-origin push attacks.

4.15. Allowing Versioning and Evolution

It's often necessary to introduce new features into application protocols, and change existing ones.

In HTTP, backwards-incompatible changes are possible using a number of mechanisms:

- o Using a distinct link relation type [RFC8288] to identify a URL for a resource that implements the new functionality.

- o Using a distinct media type [RFC6838] to identify formats that enable the new functionality.

- o Using a distinct HTTP header field to implement new functionality outside the message body.

5. IANA Considerations

This document has no requirements for IANA.

6. Security Considerations

Section 4.10 discusses the impact of using stateful mechanisms in the protocol as ambient authority, and suggests a mitigation.

Section 4.4.2 requires support for 'https' URLs, and discourages the use of 'http' URLs, to provide authentication, integrity and confidentiality, as well as mitigate pervasive monitoring attacks.

Section 4.12 highlights the implications of Web browsers' capabilities on applications that use HTTP.

Section 4.13 discusses the issues that arise when applications are deployed on the same origin as Web sites (and other applications).

Section 4.14 highlights risks of using HTTP/2 server push in a manner other than specified.

Applications that use HTTP in a manner that involves modification of implementations - for example, requiring support for a new URI scheme, or a non-standard method - risk having those implementations "fork" from their parent HTTP implementations, with the possible result that they do not benefit from patches and other security improvements incorporated upstream.

6.1. Privacy Considerations

HTTP clients can expose a variety of information to servers. Besides information that's explicitly sent as part of an application's operation (for example, names and other user-entered data), and "on the wire" (which is one of the reasons https is recommended in Section 4.4.2), other information can be gathered through less obvious means - often by connecting activities of a user over time.

This includes session information, tracking the client through fingerprinting, and mobile code.

Session information includes things like the IP address of the client, TLS session tickets, Cookies, ETags stored in the client's cache, and other stateful mechanisms. Applications are advised to avoid using session mechanisms unless they are unavoidable or necessary for operation, in which case these risks needs to be documented. When they are used, implementations should be encouraged to allow clearing such state.

Fingerprinting uses unique aspects of a client's messages and behaviours to connect disparate requests and connections. For example, the User-Agent request header conveys specific information about the implementation; the Accept-Language request header conveys the users' preferred language. In combination, a number of these markers can be used to uniquely identify a client, impacting its control over its data. As a result, applications are advised to specify that clients should only emit the information they need to function in requests.

Finally, if an application exposes the ability to run mobile code, great care needs to be taken, since any ability to observe its environment can be used as an opportunity to both fingerprint the client and to obtain and manipulate private data (including session information). For example, access to high-resolution timers (even indirectly) can be used to profile the underlying hardware, creating a unique identifier for the system. Applications are advised to avoid allowing the use of mobile code where possible; when it cannot be avoided, the resulting system's security properties need be carefully scrutinised.

7. References

7.1. Normative References

[I-D.ietf-httpbis-cache]
Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", draft-ietf-httpbis-cache-05 (work in progress), July 2019.

[I-D.ietf-httpbis-messaging]
Fielding, R., Nottingham, M., and J. Reschke, "HTTP/1.1 Messaging", draft-ietf-httpbis-messaging-05 (work in progress), July 2019.

[I-D.ietf-httpbis-semantics]
Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", draft-ietf-httpbis-semantics-05 (work in progress), July 2019.

[I-D.nottingham-rfc5785bis]
Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", draft-nottingham-rfc5785bis-11 (work in progress), April 2019.

[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>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

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

[RFC6648] Saint-Andre, P., Crocker, D., and M. Nottingham, "Deprecating the "X-" Prefix and Similar Constructs in Application Protocols", BCP 178, RFC 6648, DOI 10.17487/RFC6648, June 2012, <https://www.rfc-editor.org/info/rfc6648>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7320] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 7320, DOI 10.17487/RFC7320, July 2014, <https://www.rfc-editor.org/info/rfc7320>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[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>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

7.2. Informative References

[CSP] West, M., "Content Security Policy Level 3", World Wide Web Consortium WD WD-CSP3-20160913, September 2016, <https://www.w3.org/TR/2016/WD-CSP3-20160913>.

[FETCH] WHATWG, "Fetch - Living Standard", n.d., <https://fetch.spec.whatwg.org>.

[HTML] WHATWG, "HTML - Living Standard", n.d., <https://html.spec.whatwg.org>.

[I-D.ietf-httpbis-header-structure] Nottingham, M. and P. Kamp, "Structured Headers for HTTP", draft-ietf-httpbis-header-structure-13 (work in progress), August 2019.

[I-D.ietf-httpbis-rfc6265bis] Barth, A. and M. West, "Cookies: HTTP State Management Mechanism", draft-ietf-httpbis-rfc6265bis-03 (work in progress), April 2019.

[REFERRER-POLICY]
Eisinger, J. and E. Stark, "Referrer Policy", World Wide Web Consortium CR CR-referrer-policy-20170126, January 2017, <https://www.w3.org/TR/2017/CR-referrer-policy-20170126>.

[RFC3205] Moore, K., "On the use of HTTP as a Substrate", BCP 56, RFC 3205, DOI 10.17487/RFC3205, February 2002, <https://www.rfc-editor.org/info/rfc3205>.

[RFC4791] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <https://www.rfc-editor.org/info/rfc4791>.

[RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, DOI 10.17487/RFC4918, June 2007, <https://www.rfc-editor.org/info/rfc4918>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale Content", RFC 5861, DOI 10.17487/RFC5861, May 2010, <https://www.rfc-editor.org/info/rfc5861>.

[RFC6415] Hammer-Lahav, E., Ed. and B. Cook, "Web Host Metadata", RFC 6415, DOI 10.17487/RFC6415, October 2011, <https://www.rfc-editor.org/info/rfc6415>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <https://www.rfc-editor.org/info/rfc6570>.

[RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <https://www.rfc-editor.org/info/rfc6797>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <https://www.rfc-editor.org/info/rfc7258>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/info/rfc7595>.

[RFC7605] Touch, J., "Recommendations on Using Assigned Transport Port Numbers", BCP 165, RFC 7605, DOI 10.17487/RFC7605, August 2015, <https://www.rfc-editor.org/info/rfc7605>.

[RFC7616] Shekh-Yusef, R., Ed., Ahrens, D., and S. Bremer, "HTTP Digest Access Authentication", RFC 7616, DOI 10.17487/RFC7616, September 2015, <https://www.rfc-editor.org/info/rfc7616>.

[RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <https://www.rfc-editor.org/info/rfc7617>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/info/rfc7807>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8297] Oku, K., "An HTTP Status Code for Indicating Hints", RFC 8297, DOI 10.17487/RFC8297, December 2017, <https://www.rfc-editor.org/info/rfc8297>.

[SECCTXT] West, M., "Secure Contexts", World Wide Web Consortium CR CR-secure-contexts-20160915, September 2016, <https://www.w3.org/TR/2016/CR-secure-contexts-20160915>.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, M., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, November 2008, <http://www.w3.org/TR/2008/REC-xml-20081126>.

7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

[2] http://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/bcp56bis

Appendix A. Changes from RFC 3205

[RFC3205] captured the Best Current Practice in the early 2000's, based on the concerns facing protocol designers at the time. Use of HTTP has changed considerably since then, and as a result this document is substantially different. As a result, the changes are too numerous to list individually.

Author's Address

Mark Nottingham

Email: mnot@mnot.net
URI: https://www.mnot.net/

HTTP Working Group K. Oku
Internet-Draft Fastly
Intended status: Experimental Y. Weiss
Expires: January 3, 2019 Akamai
July 2, 2018

Cache Digests for HTTP/2
draft-ietf-httpbis-cache-digest-05

Abstract

This specification defines a HTTP/2 frame type to allow clients to inform the server of their cache’s contents. Servers can then use this to inform their choices of what to push to clients.

Note to Readers

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

Working Group information can be found at http://httpwg.github.io/ ; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/cache-digest .

Status of This Memo

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Table of Contents

1. Introduction

HTTP/2 [RFC7540] allows a server to "push" synthetic request/response pairs into a client's cache optimistically. While there is strong interest in using this facility to improve perceived Web browsing performance, it is sometimes counterproductive because the client might already have cached the "pushed" response.

When this is the case, the bandwidth used to "push" the response is effectively wasted, and represents opportunity cost, because it could be used by other, more relevant responses. HTTP/2 allows a stream to be cancelled by a client using a RST_STREAM frame in this situation, but there is still at least one round trip of potentially wasted capacity even then.

This specification defines a HTTP/2 frame type to allow clients to inform the server of their freshly cached contents using a Cuckoo-filter [Cuckoo] based digest. Servers can then use this to inform their choices of what to push to clients.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. The CACHE_DIGEST Frame

The CACHE_DIGEST frame type is 0xd (decimal 13).

```
+-------------------------------+-------------------------------+
|         Origin-Len (16)       | Origin? (\*)                ...
+-------------------------------+-------------------------------+
|                   Digest-Value? (\*)                        ...
+---------------------------------------------------------------+
```

The CACHE_DIGEST frame payload has the following fields:

Origin-Len: An unsigned, 16-bit integer indicating the length, in octets, of the Origin field.

Origin: A sequence of characters containing the ASCII serialization of an origin ([RFC6454], Section 6.2) that the Digest-Value applies to.

Digest-Value: A sequence of octets containing the digest as computed in Section 2.1.1 and Section 2.1.2.

The CACHE_DIGEST frame defines the following flags:

- *RESET* (0x1): When set, indicates that any and all cache digests for the applicable origin held by the recipient MUST be considered invalid.

- *COMPLETE* (0x2): When set, indicates that the currently valid set of cache digests held by the server constitutes a complete representation of the cache's state regarding that origin.

2.1. Client Behavior

A CACHE_DIGEST frame MUST be sent from a client to a server on stream 0, and conveys a digest of the contents of the client's cache for the indicated origin.

In typical use, a client will send one or more CACHE_DIGESTs immediately after the first request on a connection for a given origin, on the same stream, because there is usually a short period of inactivity then, and servers can benefit most when they understand the state of the cache before they begin pushing associated assets (e.g., CSS, JavaScript and images). Clients MAY send CACHE_DIGEST at other times.

If the cache's state is cleared, lost, or the client otherwise wishes the server to stop using previously sent CACHE_DIGESTs, it can send a CACHE_DIGEST with the RESET flag set.

When generating CACHE_DIGEST, a client MUST NOT include stale-cached responses or responses whose URLs do not share origins [RFC6454] with the indicated origin. Clients MUST NOT send CACHE_DIGEST frames on connections that are not authoritative (as defined in [RFC7540], 10.1) for the indicated origin.

When the CACHE_DIGEST frames sent represent the complete set of stored responses, the last such frame SHOULD have a COMPLETE flag set, to indicate to the server that it has all relevant state. Note that for the purposes of COMPLETE, responses cached since the beginning of the connection or the last RESET flag on a CACHE_DIGEST frame need not be included.

CACHE_DIGEST has no defined meaning when sent from servers, and SHOULD be ignored by clients.

2.1.1. Creating a digest

Given the following inputs:

- "P", an integer smaller than 256, that indicates the probability of a false positive that is acceptable, expressed as "1/2**P".

- "N", an integer that represents the number of entries - a prime number smaller than 2**32

1. Let "f" be the number of bits per fingerprint, calculated as "P + 3"

2. Let "b" be the bucket size, defined as 4.

3. Let "allocated" be the closest power of 2 that is larger than "N".

4. Let "bytes" be "f"*"allocated"*"b"/8 rounded up to the nearest integer

5. Add 5 to "bytes"

6. Allocate memory of "bytes" and set it to zero. Assign it to "digest-value".

7. Set the first byte to "P"

8. Set the second till fifth bytes to "N" in big endian form

9. Return the "digest-value".

Note: "allocated" is necessary due to the nature of the way Cuckoo filters are creating the secondary hash, by XORing the initial hash and the fingerprint's hash. The XOR operation means that secondary hash can pick an entry beyond the initial number of entries, up to the next power of 2. In order to avoid issues there, we allocate the table appropriately. For increased space efficiency, it is recommended that implementations pick a number of entries that's close to the next power of 2.

2.1.2. Adding a URL to the Digest-Value

Given the following inputs:

- "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]
- "maxcount" - max number of cuckoo hops
- "digest-value"

1. Let "f" be the value of the first byte of "digest-value".

2. Let "b" be the bucket size, defined as 4.

3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.

4. Let "key" be the return value of Section 2.1.5 with "URL" as input.

5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.

6. Let "dest_fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.

7. Let "h2" be the return value of Section 2.1.7 with "h1", "dest_fingerprint" and "N" as inputs.

8. Let "h" be either "h1" or "h2", picked in random.

9. While "maxcount" is larger than zero:

    1. Let "position_start" be 40 + "h" * "f" * "b".

    2. Let "position_end" be "position_start" + "f" * "b".

    3. While "position_start" < "position_end":

        1. Let "bits" be "f" bits from "digest_value" starting at "position_start".

        2. If "bits" is all zeros, set "bits" to "dest_fingerprint" and terminate these steps.

        3. Add "f" to "position_start".

    4. Let "e" be a random number from 0 to "b".

    5. Subtract "f" * ("b" - "e") from "position_start".

    6. Let "bits" be "f" bits from "digest_value" starting at "position_start".

    7. Let "fingerprint" be the value of bits, read as big endian.

    8. Set "bits" to "dest_fingerprint".

    9. Set "dest_fingerprint" to "fingerprint".

    10. Let "h" be Section 2.1.7 with "h", "dest_fingerprint" and "N" as inputs.

    11. Subtract 1 from "maxcount".

10. Subtract "f" from "position_start".

11. Let "fingerprint" be the "f" bits starting at "position_start".

12. Let "h1" be "h"

13. Subtract 1 from "maxcount".

14. If "maxcount" is zero, return an error.

15. Go to step 7.

2.1.3. Removing a URL to the Digest-Value

Given the following inputs:

- "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]
- "digest-value"

1. Let "f" be the value of the first byte of "digest-value".
2. Let "b" be the bucket size, defined as 4.
3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.
4. Let "key" be the return value of Section 2.1.5 with "URL" as input.
5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.
6. Let "fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.
7. Let "h2" be the return value of Section 2.1.7 with "h1", "fingerprint" and "N" as inputs.
8. Let "hashes" be an array containing "h1" and "h2".
9. For each "h" in "hashes":
    1. Let "position_start" be 40 + "h" * "f" * "b".
    2. Let "position_end" be "position_start" + "f" * "b".

3. While "position_start" < "position_end":

   1. Let "bits" be "f" bits from "digest_value" starting at "position_start".

   2. If "bits" is "fingerprint", set "bits" to all zeros and terminate these steps.

   3. Add "f" to "position_start".

2.1.4. Computing a fingerprint value

Given the following inputs:

- "key", an array of characters

- "f", an integer indicating the number of output bits

1. Let "hash-value" be the SHA-256 message digest [RFC6234] of "key", expressed as an integer.

2. Let "h" be the number of bits in "hash-value"

3. Let "fingerprint-value" be 0

4. While "fingerprint-value" is 0 and "h" > "f":

   1. Let "fingerprint-value" be the "f" least significant bits of "hash-value".

   2. Let "hash-value" be the "h"-"f" most significant bits of "hash-value".

   3. Subtract "f" from "h".

5. If "fingerprint-value" is 0, let "fingerprint-value" be 1.

6. Return "fingerprint-value".

Note: Step 5 is to handle the extremely unlikely case where a SHA-256 digest of "key" is all zeros. The implications of it means that there's an infitisimaly larger probability of getting a "fingerprint-value" of 1 compared to all other values. This is not a problem for any practical purpose.

2.1.5. Computing the key

Given the following inputs:

- "URL", an array of characters

1. Let "key" be "URL" converted to an ASCII string by percent-encoding as appropriate [RFC3986].

2. Return "key"

2.1.6. Computing a Hash Value

Given the following inputs:

- "key", an array of characters.

- "N", an integer

"hash-value" can be computed using the following algorithm:

1. Let "hash-value" be the SHA-256 message digest [RFC6234] of "key", truncated to 32 bits, expressed as an integer.

2. Return "hash-value" modulo N.

2.1.7. Computing an Alternative Hash Value

Given the following inputs:

- "hash1", an integer indicating the previous hash.

- "fingerprint", an integer indicating the fingerprint value.

- "N", an integer indicating the number of entries in the digest.

1. Let "fingerprint-string" be the value of "fingerprint" in base 10, expressed as a string.

2. Let "hash2" be the return value of Section 2.1.6 with "fingerprint-string" and "N" as inputs, XORed with "hash1".

3. Return "hash2".

2.2. Server Behavior

In typical use, a server will query (as per Section 2.2.1) the CACHE_DIGESTs received on a given connection to inform what it pushes to that client;

- o If a given URL has a match in a current CACHE_DIGEST, a complete response need not be pushed; The server MAY push a 304 response for that resource, indicating the client that it hasn't changed.

- o If a given URL has no match in any current CACHE_DIGEST, the client does not have a cached copy, and a complete response can be pushed.

Servers MAY use all CACHE_DIGESTs received for a given origin as current, as long as they do not have the RESET flag set; a CACHE_DIGEST frame with the RESET flag set MUST clear any previously stored CACHE_DIGESTs for its origin. Servers MUST treat an empty Digest-Value with a RESET flag set as effectively clearing all stored digests for that origin.

Clients are not likely to send updates to CACHE_DIGEST over the lifetime of a connection; it is expected that servers will separately track what cacheable responses have been sent previously on the same connection, using that knowledge in conjunction with that provided by CACHE_DIGEST.

Servers MUST ignore CACHE_DIGEST frames sent on a stream other than 0.

2.2.1. Querying the Digest for a Value

Given the following inputs:

- o "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234].

- o "digest-value", an array of bits.

1. Let "f" be the value of the first byte of "digest-value".

2. Let "b" be the bucket size, defined as 4.

3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.

4. Let "key" be the return value of Section 2.1.5 with "URL" as input.

5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.

6. Let "fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.

7. Let "h2" be the return value of Section 2.1.7 with "h1", "fingerprint" and "N" as inputs.

8. Let "hashes" be an array containing "h1" and "h2".

9. For each "h" in "hashes":

    1. Let "position_start" be 40 + "h" * "f" * "b".

    2. Let "position_end" be "position_start" + "f" * "b".

    3. While "position_start" < "position_end":

        1. Let "bits" be "f" bits from "digest_value" starting at "position_start".

        2. If "bits" is "fingerprint", return true

        3. Add "f" to "position_start".

10. Return false.

3. The SETTINGS_SENDING_CACHE_DIGEST SETTINGS Parameter

A Client SHOULD notify its support for CACHE_DIGEST frames by sending the SETTINGS_SENDING_CACHE_DIGEST (0xXXX) SETTINGS parameter.

The value of the parameter is a bit-field of which the following bits are defined:

DIGEST_PENDING (0x1): When set it indicates that the client has a digest to send, and the server may choose to wait for a digest in order to make server push decisions.

Rest of the bits MUST be ignored and MUST be left unset when sending.

The initial value of the parameter is zero (0x0) meaning that the client has no digest to send the server.

4. The SETTINGS_ACCEPT_CACHE_DIGEST SETTINGS Parameter

A server can notify its support for CACHE_DIGEST frame by sending the SETTINGS_ACCEPT_CACHE_DIGEST (0x7) SETTINGS parameter. If the server is tempted to making optimizations based on CACHE_DIGEST frames, it SHOULD send the SETTINGS parameter immediately after the connection is established.

The value of the parameter is a bit-field of which the following bits are defined:

ACCEPT (0x1): When set, it indicates that the server is willing to make use of a digest of cached responses.

Rest of the bits MUST be ignored and MUST be left unset when sending.

The initial value of the parameter is zero (0x0) meaning that the server is not interested in seeing a CACHE_DIGEST frame.

Some underlying transports allow the server's first flight of application data to reach the client at around the same time when the client sends it's first flight data. When such transport (e.g., TLS 1.3 [I-D.ietf-tls-tls13] in full-handshake mode) is used, a client can postpone sending the CACHE_DIGEST frame until it receives a SETTINGS_ACCEPT_CACHE_DIGEST settings value.

When the underlying transport does not have such property (e.g., TLS 1.3 in 0-RTT mode), a client can reuse the settings value found in previous connections to that origin [RFC6454] to make assumptions.

5. IANA Considerations

This document registers the following entry in the Permanent Message Headers Registry, as per [RFC3864]:

- Header field name: Cache-Digest
- Applicable protocol: http
- Status: experimental
- Author/Change controller: IESG
- Specification document(s): [this document]

This document registers the following entry in the HTTP/2 Frame Type Registry, as per [RFC7540]:

- o Frame Type: CACHE_DIGEST
- o Code: 0xd
- o Specification: [this document]

This document registers the following entry in the HTTP/2 Settings Registry, as per [RFC7540]:

- o Code: 0x7
- o Name: SETTINGS_ACCEPT_CACHE_DIGEST
- o Initial Value: 0x0
- o Reference: [this document]

6. Security Considerations

The contents of a User Agent’s cache can be used to re-identify or "fingerprint" the user over time, even when other identifiers (e.g., Cookies [RFC6265]) are cleared.

CACHE_DIGEST allows such cache-based fingerprinting to become passive, since it allows the server to discover the state of the client’s cache without any visible change in server behaviour.

As a result, clients MUST mitigate for this threat when the user attempts to remove identifiers (e.g., "clearing cookies"). This could be achieved in a number of ways; for example: by clearing the cache, by changing one or both of N and P, or by adding new, synthetic entries to the digest to change its contents.

TODO: discuss how effective the suggested mitigations actually would be.

Additionally, User Agents SHOULD NOT send CACHE_DIGEST when in "privacy mode."

7. References

7.1. Normative References

[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>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

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

[RFC7230] 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>.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <https://www.rfc-editor.org/info/rfc7232>.

[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>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

7.2. Informative References

[Cuckoo] "Cuckoo Filter: Practically Better Than Bloom", n.d., <https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf>.

[Fetch] "Fetch Standard", n.d., <https://fetch.spec.whatwg.org/>.

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc3864>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[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>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[Service-Workers] Russell, A., Song, J., Archibald, J., and M. Kruisselbrink, "Service Workers 1", W3C Working Draft WD-service-workers-1-20161011, October 2016, <https://www.w3.org/TR/2016/WD-service-workers-1-20161011/>.

Appendix A. Encoding the CACHE_DIGEST frame as an HTTP Header

On some web browsers that support Service Workers [Service-Workers] but not Cache Digests (yet), it is possible to achieve the benefit of using Cache Digests by emulating the frame using HTTP Headers.

For the sake of interoperability with such clients, this appendix defines how a CACHE_DIGEST frame can be encoded as an HTTP header named "Cache-Digest".

The definition uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with the list rule extension defined in [RFC7230], Section 7.

```
Cache-Digest  = 1#digest-entity
digest-entity = digest-value *(OWS ";" OWS digest-flag)
digest-value  = <Digest-Value encoded using base64url>
digest-flag   = token
```

A Cache-Digest request header is defined as a list construct of cache-digest-entities. Each cache-digest-entity corresponds to a CACHE_DIGEST frame.

Digest-Value is encoded using base64url [RFC4648], Section 5. Flags that are set are encoded as digest-flags by their names that are compared case-insensitively.

Origin is omitted in the header form. The value is implied from the value of the ":authority" pseudo header. Client MUST only send Cache-Digest headers containing digests that belong to the origin specified by the HTTP request.

The example below contains a digest of one resource and has only the "COMPLETE" flag set.

```
Cache-Digest: AfdA; complete
```

Clients MUST associate Cache-Digest headers to every HTTP request, since Fetch [Fetch] - the HTTP API supported by Service Workers - does not define the order in which the issued requests will be sent to the server nor guarantees that all the requests will be transmitted using a single HTTP/2 connection.

Also, due to the fact that any header that is supplied to Fetch is required to be end-to-end, there is an ambiguity in what a Cache-Digest header respresents when a request is transmitted through a proxy. The header may represent the cache state of a client or that of a proxy, depending on how the proxy handles the header.

Appendix B. Changes

B.1. Since draft-ietf-httpbis-cache-digest-04

- o Remove ETag from the digest key calculations.
- o Add SETTINGS_ prefix to parameter names.

B.2. Since draft-ietf-httpbis-cache-digest-03

- o Yoav becomes an author; Mark steps down.

B.3. Since draft-ietf-httpbis-cache-digest-02

- o Switch to Cuckoo Filter.

B.4. Since draft-ietf-httpbis-cache-digest-01

- o Added definition of the Cache-Digest header.
- o Introduce ACCEPT_CACHE_DIGEST SETTINGS parameter.

- o Change intended status from Standard to Experimental.

B.5. Since draft-ietf-httpbis-cache-digest-00

- o Make the scope of a digest frame explicit and shift to stream 0.

Appendix C. Acknowledgements

+{:numbered="false"}

Thanks to Stefan Eissing for his suggestions.

Authors' Addresses

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Yoav Weiss
Akamai

Email: yoav@yoav.ws
URI: https://blog.yoav.ws/

HTTP Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 8, 2019

S. Ludin
Akamai Technologies
M. Nottingham
Fastly
N. Sullivan
Cloudflare
February 04, 2019

CDN Loop Detection
draft-ietf-httpbis-cdn-loop-02

Abstract

This document defines the CDN-Loop request header field for HTTP. CDN-Loop addresses an operational need that occurs when an HTTP request is intentionally forwarded between Content Delivery Networks (CDNs), but is then accidentally or maliciously re-routed back into the original CDN causing a non-terminating loop. The new header field can be used to identify the error and terminate the loop.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

In modern deployments of HTTP servers, it is common to interpose Content Delivery Networks (CDNs) in front of origin servers to improve end-user perceived latency, reduce operational costs, and improve scalability and reliability of services.

Often, more than one CDN is in use by a given origin. This happens for a variety of reasons, such as cost savings, arranging for failover should one CDN have issues, or to directly compare their services.

As a result, it is not unknown for forwarding CDNs to be configured in a "loop" accidentally; because routing is achieved through a combination of DNS and forwarding rules, and site configurations are sometimes complex and managed by several parties.

When this happens, it is difficult to debug. Additionally, it sometimes isn't accidental; loops between multiple CDNs can be used as an attack vector (e.g., see [loop-attack]), especially if one CDN unintentionally strips the loop detection headers of another.

This specification defines the CDN-Loop HTTP request header field to help detect such attacks and accidents among implementing forwarding CDNs, by disallowing its modification by their customers.

1.1. Relationship to Via

HTTP defines the Via header field in [RFC7230], Section 5.7.1 for "tracking message forwards, avoiding request loops, and identifying the protocol capabilities of senders along the request/response chain."

In theory, Via could be used to identify these loops. However, in practice it is not used in this fashion, because some HTTP servers use Via for other purposes - in particular, some implementations disable some HTTP/1.1 features when the Via header is present.

1.2. Conventions and Definitions

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 specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [RFC7230], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Additionally, it uses the token, OWS, uri-host and port rules from [RFC7230] and the parameter rule from [RFC7231].

2. The CDN-Loop Request Header Field

The CDN-Loop request header field is intended to help a Content Delivery Network identify when an incoming request has already passed through that CDN's servers, to detect loops.

```
CDN-Loop  = #cdn-info
cdn-info  = cdn-id *( OWS ";" OWS parameter )
cdn-id    = ( uri-host [ ":" port ] ) / pseudonym
pseudonym = token
```

The cdn-id identifies the CDN using either a hostname under its control or a pseudonym. Hostnames are preferred, to help avoid accidental collisions. If a pseudonym is used, unintentional collisions are more likely, and therefore values should be carefully chosen to prevent them; for example, using a well-known value (such as the recognized name of the CDN in question), or a generated value with enough entropy to make collisions unlikely (such as a UUID [RFC4122]).

Optionally, cdn-info can have semicolon-separated key/value parameters, to accommodate additional information for the CDN's use.

Conforming Content Delivery Networks SHOULD add a cdn-info to this header field in all requests they generate or forward (creating the header field if necessary).

As with all HTTP header fields defined using the "#" rule, the CDN-Loop header field can be added to by comma-separating values, or by creating a new header field with the desired value.

For example:

```
GET /image.jpg HTTP/1.1
Host: cdn-customer.example
User-Agent: ExampleBrowser/5
CDN-Loop: foo123.foocdn.example, barcdn.example; trace="abcdef"
CDN-Loop: AnotherCDN; abc=123; def="456"
```

Note that the pseudonym syntax does not allow whitespace, DQUOTE or any of the characters "(),/:;<=>?@[]{}". See [RFC7230], Section 3.2.6. Likewise, note the rules for when parameter values need to be quoted in [RFC7231], Section 3.1.1.

The effectiveness of this mechanism relies on all intermediaries preserving the header field, since removing (or allowing it to be removed, e.g., by customer configuration) would prevent downstream CDNs from using it to detect looping. In general, unknown header fields are not removed by intermediaries, but there may be need to add CDN-Loop to an implementation's list of header fields that are not to be removed under any circumstances. The header field SHOULD NOT be used for other purposes.

3. Security Considerations

The threat model that the CDN-Loop header field addresses is a customer who is attacking a service provider by configuring a forwarding loop by accident or malice. For it to function, CDNs cannot allow customers to modify or remove it in their configuration (see Section 2).

Note that a CDN that allows customers to remove or modify the CDN-Loop header field (i.e., they do not implement this specification) remains an attack vector against both implementing and non-implementing CDNs.

A CDN's use of the CDN-Loop header field might expose its presence. For example, if CDN A is configured to forward its requests to CDN B for a given origin, CDN B's presence can be revealed if it behaves differently based upon the presence of the CDN-Loop header field.

The CDN-Loop header field can be generated by any client, and therefore its contents cannot be trusted. CDNs who modify their behaviour based upon its contents should assure that this does not become an attack vector (e.g., for Denial-of-Service).

It is possible to sign the contents of the header field (either by putting the signature directly into the field's content, or using another header field), but such use is not defined (or required) by this specification.

Depending on how it is used, CDN-Loop can expose information about the internal configuration of the CDN; for example, the number of hops inside the CDN, and the hostnames of nodes.

4. IANA Considerations

This document registers the "CDN-Loop" header field in the Permanent Message Header Field Names registry.

- o Header Field Name: CDN-Loop
- o Protocol: http
- o Status: standard
- o Reference: (this document)

5. References

5.1. Normative References

[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>.

[RFC7230] 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>.

[RFC7231] 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>.

[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>.

5.2. Informative References

[loop-attack] Chen, J., Jiang, J., Zheng, X., Duan, H., Liang, J., Li, K., Wan, T., and V. Paxson, "Forwarding-Loop Attacks in Content Delivery Networks", DOI 10.14722/ndss.2016.23442, February 2016, <http://www.icir.org/vern/papers/cdn-loops.NDSS16.pdf>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

Authors' Addresses

Stephen Ludin
Akamai Technologies

Email: sludin@akamai.com

Mark Nottingham
Fastly

Email: mnot@fastly.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

HTTP Working Group   I. Grigorik
Internet-Draft   Y. Weiss
Intended status: Experimental   Google
Expires: January 4, 2021   July 3, 2020

HTTP Client Hints
draft-ietf-httpbis-client-hints-15

Abstract

HTTP defines proactive content negotiation to allow servers to select the appropriate response for a given request, based upon the user agent's characteristics, as expressed in request headers. In practice, user agents are often unwilling to send those request headers, because it is not clear whether they will be used, and sending them impacts both performance and privacy.

This document defines an Accept-CH response header that servers can use to advertise their use of request headers for proactive content negotiation, along with a set of guidelines for the creation of such headers, colloquially known as "Client Hints."

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/client-hints [3].

Status of This Memo

This Internet-Draft will expire on January 4, 2021.

Copyright Notice

Table of Contents

1. Introduction

There are thousands of different devices accessing the web, each with different device capabilities and preference information. These device capabilities include hardware and software characteristics, as well as dynamic user and user agent preferences. Historically, applications that wanted the server to optimize content delivery and user experience based on such capabilities had to rely on passive identification (e.g., by matching the User-Agent header field (Section 5.5.3 of [RFC7231]) against an established database of user agent signatures), use HTTP cookies [RFC6265] and URL parameters, or use some combination of these and similar mechanisms to enable ad hoc content negotiation.

Such techniques are expensive to set up and maintain, and are not portable across both applications and servers. They also make it hard for both user agent and server to understand which data are required and is in use during the negotiation:

- o User agent detection cannot reliably identify all static variables, cannot infer dynamic user agent preferences, requires an external device database, is not cache friendly, and is reliant on a passive fingerprinting surface.
- o Cookie-based approaches are not portable across applications and servers, impose additional client-side latency by requiring JavaScript execution, and are not cache friendly.
- o URL parameters, similar to cookie-based approaches, suffer from lack of portability, and are hard to deploy due to a requirement to encode content negotiation data inside of the URL of each resource.

Proactive content negotiation (Section 3.4.1 of [RFC7231]) offers an alternative approach; user agents use specified, well-defined request headers to advertise their capabilities and characteristics, so that servers can select (or formulate) an appropriate response based on those request headers (or on other, implicit characteristics).

However, traditional proactive content negotiation techniques often mean that user agents send these request headers prolifically. This causes performance concerns (because it creates "bloat" in requests), as well as privacy issues; passively providing such information allows servers to silently fingerprint the user.

This document defines Client Hints, a framework that enables servers to opt-in to specific proactive content negotiation features, adapting their content accordingly, as well as guidelines for content negotiation mechanisms that use the framework. This document also defines a new response header, Accept-CH, that allows an origin server to explicitly ask that user agents send these headers in requests.

Client Hints mitigate performance concerns by assuring that user agents will only send the request headers when they're actually going to be used, and privacy concerns of passive fingerprinting by requiring explicit opt-in and disclosure of required headers by the server through the use of the Accept-CH response header, turning passive fingerprinting vectors into active ones.

The document does not define specific usages of Client Hints. Such usages need to be defined in their respective specifications.

One example of such usage is the User Agent Client Hints [UA-CH].

1.1. 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 Backus-Naur Form (ABNF) notation of [RFC5234].

2. Client Hint Request Header Fields

A Client Hint request header field is a HTTP header field that is used by HTTP user agents to indicate data that can be used by the server to select an appropriate response. Each one conveys user agent preferences that the server can use to adapt and optimize the response.

2.1. Sending Client Hints

User agents choose what Client Hints to send in a request based on their default settings, user configuration, and server preferences expressed in "Accept-CH". The user agent and server can use an opt-in mechanism outlined below to negotiate which header fields need to be sent to allow for efficient content adaption, and optionally use additional mechanisms (e.g., as outlined in [CLIENT-HINTS-INFRASTRUCTURE]) to negotiate delegation policies that control access of third parties to those same header fields. User agents SHOULD require an opt-in to send any hints that are not listed in the low-entropy hint table at [CLIENT-HINTS-INFRASTRUCTURE].

Implementers need to be aware of the fingerprinting implications when implementing support for Client Hints, and follow the considerations outlined in the Security Considerations (Section 4) section of this document.

2.2. Server Processing of Client Hints

When presented with a request that contains one or more Client Hint header fields, servers can optimize the response based upon the information in them. When doing so, and if the resource is cacheable, the server MUST also generate a Vary response header field (Section 7.1.4 of [RFC7231]) to indicate which hints can affect the selected response and whether the selected response is appropriate for a later request.

Servers MUST ignore hints they do not understand nor support. There is no mechanism for servers to indicate to user agents that hints were ignored.

Furthermore, the server can generate additional response header fields (as specified by the hint or hints in use) that convey related values to aid client processing.

3. Advertising Server Support

Servers can advertise support for Client Hints using the mechanism described below.

3.1. The Accept-CH Response Header Field

The Accept-CH response header field indicates server support for the hints indicated in its value. Servers wishing to receive user agent information through Client Hints SHOULD add Accept-CH response header to their responses as early as possible.

Accept-CH is a Structured Header [I-D.ietf-httpbis-header-structure]. Its value MUST be an sf-list (Section 3.1 of [I-D.ietf-httpbis-header-structure]) whose members are tokens (Section 3.3.4 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

```
Accept-CH = sf-list
```

For example:

```
Accept-CH: Sec-CH-Example, Sec-CH-Example-2
```

When a user agent receives an HTTP response containing "Accept-CH", that indicates that the origin opts-in to receive the indicated request header fields for subsequent same-origin requests. The opt-in MUST be ignored if delivered over non-secure transport (using a scheme different from HTTPS). It SHOULD be persisted and bound to the origin to enable delivery of Client Hints on subsequent requests to the server's origin, for the duration of the user's session (as defined by the user agent). An opt-in overrides previous persisted opt-in values and SHOULD be persisted in its stead.

Based on the Accept-CH example above, which is received in response to a user agent navigating to "https://site.example", and delivered over a secure transport, persisted Accept-CH preferences will be bound to "https://site.example". It will then use it for navigations to e.g., "https://site.example/foobar.html", but not to e.g., "https://foobar.site.example/". It will similarly use the preference for any same-origin resource requests (e.g., to "https://site.example/image.jpg") initiated by the page constructed from the navigation's response, but not to cross-origin resource requests (e.g., "https://thirdparty.example/resource.js"). This preference will not extend to resource requests initiated to "https://site.example" from other origins (e.g., from navigations to "https://other.example/").

3.2. Interaction with Caches

When selecting a response based on one or more Client Hints, and if the resource is cacheable, the server needs to generate a Vary response header field ([RFC7234]) to indicate which hints can affect the selected response and whether the selected response is appropriate for a later request.

```
Vary: Sec-CH-Example
```

The above example indicates that the cache key needs to include the Sec-CH-Example header field.

```
Vary: Sec-CH-Example, Sec-CH-Example-2
```

The above example indicates that the cache key needs to include the Sec-CH-Example and Sec-CH-Example-2 header fields.

4. Security Considerations

4.1. Information Exposure

Request header fields used in features relying on this document expose information about the user's environment to enable privacy-preserving proactive content negotiation, and avoid exposing passive fingerprinting vectors. However, implementers need to bear in mind that in the worst case, uncontrolled and unmonitored active fingerprinting is not better than passive fingerprinting. In order to provide user privacy benefits, user agents need to apply further policies that prevent abuse of the information exposed by features using Client Hints.

The information exposed by features might reveal new information about the user and implementers ought to consider the following considerations, recommendations, and best practices.

The underlying assumption is that exposing information about the user as a request header is equivalent (from a security perspective) to exposing this information by other means. (For example, if the request's origin can access that information using JavaScript APIs, and transmit it to its servers).

Because Client Hints is an explicit opt-in mechanism, that means that servers that want access to information about the user's environment need to actively ask for it, enabling clients and privacy researchers to keep track of which origins collect that data, and potentially act upon it. The header-based opt-in means that removal of passive fingerprinting vectors is possible, such as the User-Agent string (enabling active access to that information through User-Agent Client Hints ([UA-CH]) or otherwise expose information already available through script (e.g., the Save-Data Client Hint [4]), without increasing the passive fingerprinting surface. User agents supporting Client Hints features which send certain information to opted-in servers SHOULD avoid sending the equivalent information passively.

Therefore, features relying on this document to define Client Hint headers MUST NOT provide new information that is otherwise not made available to the application by the user agent, such as existing request headers, HTML, CSS, or JavaScript.

Such features need to take into account the following aspects of the information exposed:

o Entropy - Exposing highly granular data can be used to help identify users across multiple requests to different origins.

Reducing the set of header field values that can be expressed, or restricting them to an enumerated range where the advertised value is close to but is not an exact representation of the current value, can improve privacy and reduce risk of linkability by ensuring that the same value is sent by multiple users.

- o Sensitivity - The feature SHOULD NOT expose user-sensitive information. To that end, information available to the application, but gated behind specific user actions (e.g., a permission prompt or user activation) SHOULD NOT be exposed as a Client Hint.
- o Change over time - The feature SHOULD NOT expose user information that changes over time, unless the state change itself is also exposed (e.g., through JavaScript callbacks).

Different features will be positioned in different points in the space between low-entropy, non-sensitive and static information (e.g., user agent information), and high-entropy, sensitive and dynamic information (e.g., geolocation). User agents need to consider the value provided by a particular feature vs these considerations, and may wish to have different policies regarding that tradeoff on a per-feature or other fine-grained basis.

Implementers ought to consider both user- and server- controlled mechanisms and policies to control which Client Hints header fields are advertised:

- o Implementers SHOULD restrict delivery of some or all Client Hints header fields to the opt-in origin only, unless the opt-in origin has explicitly delegated permission to another origin to request Client Hints header fields.
- o Implementers considering providing user choice mechanisms that allow users to balance privacy concerns against bandwidth limitations need to also consider that explaining to users the privacy implications involved, such as the risks of passive fingerprinting, may be challenging or even impractical.
- o Implementations specific to certain use cases or threat models MAY avoid transmitting some or all of Client Hints header fields. For example, avoid transmission of header fields that can carry higher risks of linkability.

User agents MUST clear persisted opt-in preferences when any one of site data, browsing history, browsing cache, cookies, or similar, are cleared.

4.2. Deployment and Security Risks

Deployment of new request headers requires several considerations:

- o Potential conflicts due to existing use of header field name
- o Properties of the data communicated in header field value

Authors of new Client Hints are advised to carefully consider whether they need to be able to be added by client-side content (e.g., scripts), or whether they need to be exclusively set by the user agent. In the latter case, the Sec- prefix on the header field name has the effect of preventing scripts and other application content from setting them in user agents. Using the "Sec-" prefix signals to servers that the user agent - and not application content - generated the values. See [FETCH] for more information.

By convention, request headers that are Client Hints are encouraged to use a CH- prefix, to make them easier to identify as using this framework; for example, CH-Foo or, with a "Sec-" prefix, Sec-CH-Foo. Doing so makes them easier to identify programmatically (e.g., for stripping unrecognised hints from requests by privacy filters).

A Client Hints request header negotiated using the Accept-CH opt-in mechanism MUST have a field name that matches sf-token (Section 3.3.4 of [I-D.ietf-httpbis-header-structure]).

4.3. Abuse Detection

A user agent that tracks access to active fingerprinting information SHOULD consider emission of Client Hints headers similarly to the way it would consider access to the equivalent API.

Research into abuse of Client Hints might look at how HTTP responses to requests that contain Client Hints differ from those with different values, and from those without. This might be used to reveal which Client Hints are in use, allowing researchers to further analyze that use.

5. Cost of Sending Hints

Sending Client Hints to the server incurs an increase in request byte size. Some of this increase can be mitigated by HTTP header compression schemes, but each new hint sent will still lead to some increased bandwidth usage. Servers SHOULD take that into account when opting in to receive Client Hints, and SHOULD NOT opt-in to receive hints unless they are to be used for content adaptation purposes.

Due to request byte size increase, features relying on this document to define Client Hints MAY consider restricting sending those hints to certain request destinations [FETCH], where they are more likely to be useful.

6. IANA Considerations

Features relying on this document are expected to register added request header fields in the Permanent Message Header Fields registry ([RFC3864]).

This document defines the "Accept-CH" HTTP response header field, and registers it in the same registry.

6.1. Accept-CH

- Header field name: Accept-CH
- Applicable protocol: HTTP
- Status: experimental
- Author/Change controller: IETF
- Specification document(s): Section 3.1 of this document
- Related information: for Client Hints

7. References

7.1. Normative References

[CLIENT-HINTS-INFRASTRUCTURE]
Weiss, Y., "Client Hints Infrastructure", n.d., <https://wicg.github.io/client-hints-infrastructure/>.

[I-D.ietf-httpbis-header-structure]
Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", draft-ietf-httpbis-header-structure-19 (work in progress), June 2020.

[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>.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc3864>.

[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>.

[RFC7231] 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>.

[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>.

[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>.

7.2. Informative References

[FETCH] van Kesteren, A., "Fetch", n.d., <https://fetch.spec.whatwg.org/>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[UA-CH] West, M. and Y. Weiss, "User Agent Client Hints", n.d., <https://wicg.github.io/ua-client-hints/>.

7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

[2] http://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/client-hints

[4] https://wicg.github.io/savedata/#save-data-request-header-field

Appendix A. Changes

A.1. Since -00

- Issue 168 (make Save-Data extensible) updated ABNF.
- Issue 163 (CH review feedback) editorial feedback from httpwg list.

- Issue 153 (NetInfo API citation) added normative reference.

A.2. Since -01

- Issue 200: Moved Key reference to informative.
- Issue 215: Extended passive fingerprinting and mitigation considerations.
- Changed document status to experimental.

A.3. Since -02

- Issue 239: Updated reference to CR-css-values-3
- Issue 240: Updated reference for Network Information API
- Issue 241: Consistency in IANA considerations
- Issue 250: Clarified Accept-CH

A.4. Since -03

- Issue 284: Extended guidance for Accept-CH
- Issue 308: Editorial cleanup
- Issue 306: Define Accept-CH-Lifetime

A.5. Since -04

- Issue 361: Removed Downlink
- Issue 361: Moved Key to appendix, plus other editorial feedback

A.6. Since -05

- Issue 372: Scoped CH opt-in and delivery to secure transports
- Issue 373: Bind CH opt-in to origin

A.7. Since -06

- Issue 524: Save-Data is now defined by NetInfo spec, dropping
- PR 775: Removed specific features to be defined in other specifications

A.8. Since -07

- Issue 761: Clarified that the defined headers are response headers.
- Issue 730: Replaced Key reference with Variants.
- Issue 700: Replaced ABNF with structured headers.
- PR 878: Removed Accept-CH-Lifetime based on feedback at IETF 105

A.9. Since -08

- PR 985: Describe the bytesize cost of hints.
- PR 776: Add Sec- and CH- prefix considerations.
- PR 1001: Clear CH persistence when cookies are cleared.

A.10. Since -09

- PR 1064: Fix merge issues with "cost of sending hints".

A.11. Since -10

- PR 1072: LC feedback from Julian Reschke.
- PR 1080: Improve list style.
- PR 1082: Remove section mentioning Variants.
- PR 1097: Editorial feedback from mnot.
- PR 1131: Remove unused references.
- PR 1132: Remove nested list.

A.12. Since -11

- PR 1134: Re-insert back section.

A.13. Since -12

- PR 1160: AD review.

A.14. Since -13

- PR 1171: Genart review.

A.15. Since -14

- PR 1220: AD review.

Acknowledgements

Thanks to Mark Nottingham, Julian Reschke, Chris Bentzel, Ben Greenstein, Tarun Bansal, Roy Fielding, Vasiliy Faronov, Ted Hardie, Jonas Sicking, Martin Thomson, and numerous other members of the IETF HTTP Working Group for invaluable help and feedback.

Authors' Addresses

Ilya Grigorik
Google

Email: ilya@igvita.com
URI: https://www.igvita.com/

Yoav Weiss
Google

Email: yoav@yoav.ws
URI: https://blog.yoav.ws/

HTTP   M. Nottingham
Internet-Draft   Fastly
Intended status: Standards Track   P-H. Kamp
Expires: December 5, 2020   The Varnish Cache Project
                                                            June 3, 2020

Structured Field Values for HTTP
draft-ietf-httpbis-header-structure-19

Abstract

This document describes a set of data types and associated algorithms that are intended to make it easier and safer to define and handle HTTP header and trailer fields, known as "Structured Fields", "Structured Headers", or "Structured Trailers". It is intended for use by specifications of new HTTP fields that wish to use a common syntax that is more restrictive than traditional HTTP field values.

Note to Readers

RFC EDITOR: please remove this section before publication

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at https://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/header-structure [3].

Tests for implementations are collected at https://github.com/httpwg/structured-field-tests [4].

Implementations are tracked at https://github.com/httpwg/wiki/wiki/Structured-Headers [5].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Specifying the syntax of new HTTP header (and trailer) fields is an onerous task; even with the guidance in Section 8.3.1 of [RFC7231], there are many decisions - and pitfalls - for a prospective HTTP field author.

Once a field is defined, bespoke parsers and serializers often need to be written, because each field value has slightly different handling of what looks like common syntax.

This document introduces a set of common data structures for use in definitions of new HTTP field values to address these problems. In particular, it defines a generic, abstract model for them, along with a concrete serialization for expressing that model in HTTP [RFC7230] header and trailer fields.

A HTTP field that is defined as a "Structured Header" or "Structured Trailer" (if the field can be either, it is a "Structured Field") uses the types defined in this specification to define its syntax and basic handling rules, thereby simplifying both its definition by specification writers and handling by implementations.

Additionally, future versions of HTTP can define alternative serializations of the abstract model of these structures, allowing fields that use that model to be transmitted more efficiently without being redefined.

Note that it is not a goal of this document to redefine the syntax of existing HTTP fields; the mechanisms described herein are only intended to be used with fields that explicitly opt into them.

Section 2 describes how to specify a Structured Field.

Section 3 defines a number of abstract data types that can be used in Structured Fields.

Those abstract types can be serialized into and parsed from HTTP field values using the algorithms described in Section 4.

1.1. Intentionally Strict Processing

This specification intentionally defines strict parsing and serialization behaviors using step-by-step algorithms; the only error handling defined is to fail the operation altogether.

It is designed to encourage faithful implementation and therefore good interoperability. Therefore, an implementation that tried to be helpful by being more tolerant of input would make interoperability worse, since that would create pressure on other implementations to implement similar (but likely subtly different) workarounds.

In other words, strict processing is an intentional feature of this specification; it allows non-conformant input to be discovered and corrected by the producer early, and avoids both interoperability and security issues that might otherwise result.

Note that as a result of this strictness, if a field is appended to by multiple parties (e.g., intermediaries, or different components in the sender), an error in one party's value is likely to cause the entire field value to fail parsing.

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 algorithms to specify parsing and serialization behaviors, and the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] to illustrate expected syntax in HTTP header fields. In doing so, it uses the VCHAR, SP, DIGIT, ALPHA and DQUOTE rules from [RFC5234]. It also includes the tchar and OWS rules from [RFC7230].

When parsing from HTTP fields, implementations MUST have behavior that is indistinguishable from following the algorithms. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence.

For serialization to HTTP fields, the ABNF illustrates their expected wire representations, and the algorithms define the recommended way to produce them. Implementations MAY vary from the specified behavior so long as the output is still correctly handled by the parsing algorithm.

2. Defining New Structured Fields

To specify a HTTP field as a Structured Field, its authors needs to:

- o Normatively reference this specification. Recipients and generators of the field need to know that the requirements of this document are in effect.

- o Identify whether the field is a Structured Header (i.e., it can only be used in the header section - the common case), a Structured Trailer (only in the trailer section), or a Structured Field (both).

- o Specify the type of the field value; either List (Section 3.1), Dictionary (Section 3.2), or Item (Section 3.3).

- o Define the semantics of the field value.

- o Specify any additional constraints upon the field value, as well as the consequences when those constraints are violated.

Typically, this means that a field definition will specify the top-level type - List, Dictionary or Item - and then define its allowable types, and constraints upon them. For example, a header defined as a List might have all Integer members, or a mix of types; a header defined as an Item might allow only Strings, and additionally only strings beginning with the letter "Q", or strings in lowercase. Likewise, Inner Lists (Section 3.1.1) are only valid when a field definition explicitly allows them.

When parsing fails, the entire field is ignored (see Section 4.2); in most situations, violating field-specific constraints should have the same effect. Thus, if a header is defined as an Item and required to be an Integer, but a String is received, the field will by default be ignored. If the field requires different error handling, this should be explicitly specified.

Both Items and Inner Lists allow parameters as an extensibility mechanism; this means that values can later be extended to accommodate more information, if need be. To preserve forward compatibility, field specifications are discouraged from defining the presence of an unrecognized Parameter as an error condition.

To further assure that this extensibility is available in the future, and to encourage consumers to use a complete parser implementation, a field definition can specify that "grease" Parameters be added by senders. A specification could stipulate that all Parameters that fit a defined pattern are reserved for this use and then encourage them to be sent on some portion of requests. This helps to discourage recipients from writing a parser that does not account for Parameters.

Specifications that use Dictionaries can also allow for forward compatibility by requiring that the presence of - as well as value and type associated with - unknown members be ignored. Later specifications can then add additional members, specifying constraints on them as appropriate.

An extension to a structured field can then require that an entire field value be ignored by a recipient that understands the extension if constraints on the value it defines are not met.

A field definition cannot relax the requirements of this specification because doing so would preclude handling by generic software; they can only add additional constraints (for example, on the numeric range of Integers and Decimals, the format of Strings and Tokens, the types allowed in a Dictionary's values, or the number of Items in a List). Likewise, field definitions can only use this specification for the entire field value, not a portion thereof.

This specification defines minimums for the length or number of various structures supported by implementations. It does not specify maximum sizes in most cases, but authors should be aware that HTTP implementations do impose various limits on the size of individual fields, the total number of fields, and/or the size of the entire header or trailer section.

Specifications can refer to a field name as a "structured header name", "structured trailer name" or "structured field name" as appropriate. Likewise, they can refer its field value as a "structured header value", "structured trailer value" or "structured field value" as necessary. Field definitions are encouraged to use the ABNF rules beginning with "sf-" defined in this specification; other rules in this specification are not intended for their use.

For example, a fictitious Foo-Example header field might be specified as:

```
--8<--
42. Foo-Example Header

The Foo-Example HTTP header field conveys information about how
much Foo the message has.

Foo-Example is a Item Structured Header [RFCxxxx]. Its value MUST be
an Integer (Section Y.Y of [RFCxxxx]). Its ABNF is:

  Foo-Example = sf-integer

Its value indicates the amount of Foo in the message, and MUST
be between 0 and 10, inclusive; other values MUST cause
the entire header field to be ignored.

The following parameters are defined:
* A Parameter whose name is "foourl", and whose value is a String
  (Section Y.Y of [RFCxxxx]), conveying the Foo URL
  for the message. See below for processing requirements.

"foourl" contains a URI-reference (Section 4.1 of [RFC3986]). If
its value is not a valid URI-reference, the entire header field
MUST be ignored. If its value is a relative reference (Section 4.2
of [RFC3986]), it MUST be resolved (Section 5 of [RFC3986]) before
being used.

For example:

  Foo-Example: 2; foourl="https://foo.example.com/"
-->8--
```

3. Structured Data Types

This section defines the abstract types for Structured Fields. The ABNF provided represents the on-wire format in HTTP field values.

In summary:

- There are three top-level types that a HTTP field can be defined as: Lists, Dictionaries, and Items.

- Lists and Dictionaries are containers; their members can be Items or Inner Lists (which are themselves arrays of Items).

- Both Items and Inner Lists can be parameterized with key/value pairs.

3.1. Lists

Lists are arrays of zero or more members, each of which can be an Item (Section 3.3) or an Inner List (Section 3.1.1), both of which can be Parameterized (Section 3.1.2).

The ABNF for Lists in HTTP fields is:

```
sf-list       = list-member *( OWS "," OWS list-member )
list-member   = sf-item / inner-list
```

Each member is separated by a comma and optional whitespace. For example, a field whose value is defined as a List of Strings could look like:

```
Example-StrList: "foo", "bar", "It was the best of times."
```

An empty List is denoted by not serializing the field at all. This implies that fields defined as Lists have a default empty value.

Note that Lists can have their members split across multiple lines inside a header or trailer section, as per Section 3.2.2 of [RFC7230]; for example, the following are equivalent:

```
Example-Hdr: foo, bar
```

and

```
Example-Hdr: foo
Example-Hdr: bar
```

However, individual members of a List cannot be safely split between across lines; see Section 4.2 for details.

Parsers MUST support Lists containing at least 1024 members. Field specifications can constrain the types and cardinality of individual List values as they require.

3.1.1. Inner Lists

An Inner List is an array of zero or more Items (Section 3.3). Both the individual Items and the Inner List itself can be Parameterized (Section 3.1.2).

The ABNF for Inner Lists is:

```
inner-list    = "(" *SP [ sf-item *( 1*SP sf-item ) *SP ] ")"
                parameters
```

Inner Lists are denoted by surrounding parenthesis, and have their values delimited by one or more spaces. A field whose value is defined as a List of Inner Lists of Strings could look like:

```
Example-StrListList: ("foo" "bar"), ("baz"), ("bat" "one"), ()
```

Note that the last member in this example is an empty Inner List.

A header field whose value is defined as a List of Inner Lists with Parameters at both levels could look like:

```
Example-ListListParam: ("foo"; a=1;b=2);lvl=5, ("bar" "baz");lvl=1
```

Parsers MUST support Inner Lists containing at least 256 members. Field specifications can constrain the types and cardinality of individual Inner List members as they require.

3.1.2. Parameters

Parameters are an ordered map of key-value pairs that are associated with an Item (Section 3.3) or Inner List (Section 3.1.1). The keys are unique within the scope the Parameters they occur within, and the values are bare items (i.e., they themselves cannot be parameterized; see Section 3.3).

The ABNF for Parameters is:

```
parameters    = *( ";" *SP parameter )
parameter     = param-name [ "=" param-value ]
param-name    = key
key           = ( lcalpha / "*" )
                *( lcalpha / DIGIT / "_" / "-" / "." / "*" )
lcalpha       = %x61-7A ; a-z
param-value   = bare-item
```

Note that Parameters are ordered as serialized, and Parameter keys cannot contain uppercase letters. A parameter is separated from its Item or Inner List and other parameters by a semicolon. For example:

```
Example-ParamList: abc;a=1;b=2; cde_456, (ghi;jk=4 l);q="9";r=w
```

Parameters whose value is Boolean (see Section 3.3.6) true MUST omit that value when serialized. For example, the "a" parameter here is true, while the "b" parameter is false:

```
Example-Int: 1; a; b=?0
```

Note that this requirement is only on serialization; parsers are still required to correctly handle the true value when it appears in a parameter.

Parsers MUST support at least 256 parameters on an Item or Inner List, and support parameter keys with at least 64 characters. Field specifications can constrain the order of individual Parameters, as well as their values' types as required.

3.2. Dictionaries

Dictionaries are ordered maps of name-value pairs, where the names are short textual strings and the values are Items (Section 3.3) or arrays of Items, both of which can be Parameterized (Section 3.1.2). There can be zero or more members, and their names are unique in the scope of the Dictionary they occur within.

Implementations MUST provide access to Dictionaries both by index and by name. Specifications MAY use either means of accessing the members.

The ABNF for Dictionaries is:

```
sf-dictionary  = dict-member *( OWS "," OWS dict-member )
dict-member    = member-name [ "=" member-value ]
member-name    = key
member-value   = sf-item / inner-list
```

Members are ordered as serialized, and separated by a comma with optional whitespace. Member names cannot contain uppercase characters. Names and values are separated by "=" (without whitespace). For example:

```
Example-Dict: en="Applepie", da=:w4ZibGV0w6ZydGU=:
```

Note that in this example, the final "=" is due to the inclusion of a Byte Sequence; see Section 3.3.5.

Members whose value is Boolean (see Section 3.3.6) true MUST omit that value when serialized. For example, here both "b" and "c" are true:

```
Example-Dict: a=?0, b, c; foo=bar
```

Note that this requirement is only on serialization; parsers are still required to correctly handle the true Boolean value when it appears in Dictionary values.

A Dictionary with a member whose value is an Inner List of Tokens:

```
Example-DictList: rating=1.5, feelings=(joy sadness)
```

A Dictionary with a mix of Items and Inner Lists, some with Parameters:

```
Example-MixDict: a=(1 2), b=3, c=4;aa=bb, d=(5 6);valid
```

As with lists, an empty Dictionary is represented by omitting the entire field. This implies that fields defined as Dictionaries have a default empty value.

Typically, a field specification will define the semantics of Dictionaries by specifying the allowed type(s) for individual members by their names, as well as whether their presence is required or optional. Recipients MUST ignore names that are undefined or unknown, unless the field's specification specifically disallows them.

Note that Dictionaries can have their members split across multiple lines inside a header or trailer section; for example, the following are equivalent:

```
Example-Hdr: foo=1, bar=2
```

and

```
Example-Hdr: foo=1
Example-Hdr: bar=2
```

However, individual members of a Dictionary cannot be safely split between lines; see Section 4.2 for details.

Parsers MUST support Dictionaries containing at least 1024 name/value pairs, and names with at least 64 characters. Field specifications can constrain the order of individual Dictionary members, as well as their values' types as required.

3.3. Items

An Item can be a Integer (Section 3.3.1), Decimal (Section 3.3.2), String (Section 3.3.3), Token (Section 3.3.4), Byte Sequence (Section 3.3.5), or Boolean (Section 3.3.6). It can have associated Parameters (Section 3.1.2).

The ABNF for Items is:

```
sf-item   = bare-item parameters
bare-item = sf-integer / sf-decimal / sf-string / sf-token
            / sf-binary / sf-boolean
```

For example, a header field that is defined to be an Item that is an Integer might look like:

```
Example-IntItemHeader: 5
```

or with Parameters:

```
Example-IntItem: 5; foo=bar
```

3.3.1. Integers

Integers have a range of -999,999,999,999,999 to 999,999,999,999,999 inclusive (i.e., up to fifteen digits, signed), for IEEE 754 compatibility ([IEEE754]).

The ABNF for Integers is:

```
sf-integer = ["-"] 1*15DIGIT
```

For example:

```
Example-Integer: 42
```

Integers larger than 15 digits can be supported in a variety of ways; for example, by using a String (Section 3.3.3), Byte Sequence (Section 3.3.5), or a parameter on an Integer that acts as a scaling factor.

While it is possible to serialise Integers with leading zeros (e.g., "0002", "-01") and signed zero ("-0"), these distinctions may not be preserved by implementations.

Note that commas in Integers are used in this section’s prose only for readability; they are not valid in the wire format.

3.3.2. Decimals

Decimals are numbers with an integer and a fractional component. The integer component has at most 12 digits; the fractional component has at most three digits.

The ABNF for decimals is:

```
sf-decimal  = ["-"] 1*12DIGIT "." 1*3DIGIT
```

For example, a header whose value is defined as a Decimal could look like:

```
Example-Decimal: 4.5
```

While it is possible to serialise Decimals with leading zeros (e.g., "0002.5", "-01.334"), trailing zeros (e.g., "5.230", "-0.40"), and signed zero (e.g., "-0.0"), these distinctions may not be preserved by implementations.

Note that the serialisation algorithm (Section 4.1.5) rounds input with more than three digits of precision in the fractional component. If an alternative rounding strategy is desired, this should be specified by the header definition to occur before serialisation.

3.3.3. Strings

Strings are zero or more printable ASCII [RFC0020] characters (i.e., the range %x20 to %x7E). Note that this excludes tabs, newlines, carriage returns, etc.

The ABNF for Strings is:

```
sf-string = DQUOTE *chr DQUOTE
chr       = unescaped / escaped
unescaped = %x20-21 / %x23-5B / %x5D-7E
escaped   = "\" ( DQUOTE / "\" )
```

Strings are delimited with double quotes, using a backslash ("\") to escape double quotes and backslashes. For example:

```
Example-String: "hello world"
```

Note that Strings only use DQUOTE as a delimiter; single quotes do not delimit Strings. Furthermore, only DQUOTE and "\" can be escaped; other characters after "\" MUST cause parsing to fail.

Unicode is not directly supported in Strings, because it causes a number of interoperability issues, and - with few exceptions - field values do not require it.

When it is necessary for a field value to convey non-ASCII content, a Byte Sequence (Section 3.3.5) can be specified, along with a character encoding (preferably [UTF-8]).

Parsers MUST support Strings (after any decoding) with at least 1024 characters.

3.3.4. Tokens

Tokens are short textual words; their abstract model is identical to their expression in the HTTP field value serialization.

The ABNF for Tokens is:

```
sf-token = ( ALPHA / "*" ) *( tchar / ":" / "/" )
```

For example:

```
Example-Token: foo123/456
```

Parsers MUST support Tokens with at least 512 characters.

Note that Token allows the same characters as the "token" ABNF rule defined in [RFC7230], with the exceptions that the first character is required to be either ALPHA or "*", and ":" and "/" are also allowed in subsequent characters.

3.3.5. Byte Sequences

Byte Sequences can be conveyed in Structured Fields.

The ABNF for a Byte Sequence is:

```
sf-binary = ":" *(base64) ":"
base64    = ALPHA / DIGIT / "+" / "/" / "="
```

A Byte Sequence is delimited with colons and encoded using base64 ([RFC4648], Section 4). For example:

```
Example-Binary: :cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==:
```

Parsers MUST support Byte Sequences with at least 16384 octets after decoding.

3.3.6. Booleans

Boolean values can be conveyed in Structured Fields.

The ABNF for a Boolean is:

```
sf-boolean = "?" boolean
boolean    = "0" / "1"
```

A Boolean is indicated with a leading "?" character followed by a "1" for a true value or "0" for false. For example:

Example-Bool: ?1

Note that in Dictionary (Section 3.2) and Parameter (Section 3.1.2) values, Boolean true is indicated by omitting the value.

4. Working With Structured Fields in HTTP

This section defines how to serialize and parse Structured Fields in textual HTTP field values and other encodings compatible with them (e.g., in HTTP/2 [RFC7540] before compression with HPACK [RFC7541]).

4.1. Serializing Structured Fields

Given a structure defined in this specification, return an ASCII string suitable for use in a HTTP field value.

1. If the structure is a Dictionary or List and its value is empty (i.e., it has no members), do not serialize the field at all (i.e., omit both the field-name and field-value).

2. If the structure is a List, let output_string be the result of running Serializing a List (Section 4.1.1) with the structure.

3. Else if the structure is a Dictionary, let output_string be the result of running Serializing a Dictionary (Section 4.1.2) with the structure.

4. Else if the structure is an Item, let output_string be the result of running Serializing an Item (Section 4.1.3) with the structure.

5. Else, fail serialization.

6. Return output_string converted into an array of bytes, using ASCII encoding [RFC0020].

4.1.1. Serializing a List

Given an array of (member_value, parameters) tuples as input_list, return an ASCII string suitable for use in a HTTP field value.

1. Let output be an empty string.

2. For each (member_value, parameters) of input_list:

    1. If member_value is an array, append the result of running Serializing an Inner List (Section 4.1.1.1) with (member_value, parameters) to output.

       2. Otherwise, append the result of running Serializing an Item (Section 4.1.3) with (member_value, parameters) to output.

       3. If more member_values remain in input_list:

           1. Append "," to output.

           2. Append a single SP to output.

   3. Return output.

4.1.1.1. Serializing an Inner List

Given an array of (member_value, parameters) tuples as inner_list, and parameters as list_parameters, return an ASCII string suitable for use in a HTTP field value.

1. Let output be the string "(".

2. For each (member_value, parameters) of inner_list:

    1. Append the result of running Serializing an Item (Section 4.1.3) with (member_value, parameters) to output.

    2. If more values remain in inner_list, append a single SP to output.

3. Append ")" to output.

4. Append the result of running Serializing Parameters (Section 4.1.1.2) with list_parameters to output.

5. Return output.

4.1.1.2. Serializing Parameters

Given an ordered Dictionary as input_parameters (each member having a param_name and a param_value), return an ASCII string suitable for use in a HTTP field value.

1. Let output be an empty string.

2. For each param_name with a value of param_value in input_parameters:

    1. Append ";" to output.

    2. Append the result of running Serializing a Key (Section 4.1.1.3) with param_name to output.

    3. If param_value is not Boolean true:

        1. Append "=" to output.

        2. Append the result of running Serializing a bare Item (Section 4.1.3.1) with param_value to output.

3. Return output.

4.1.1.3. Serializing a Key

Given a key as input_key, return an ASCII string suitable for use in a HTTP field value.

1. Convert input_key into a sequence of ASCII characters; if conversion fails, fail serialization.

2. If input_key contains characters not in lcalpha, DIGIT, "_", "-", ".", or "*" fail serialization.

3. If the first character of input_key is not lcalpha or "*", fail serialization.

4. Let output be an empty string.

5. Append input_key to output.

6. Return output.

4.1.2. Serializing a Dictionary

Given an ordered Dictionary as input_dictionary (each member having a member_name and a tuple value of (member_value, parameters)), return an ASCII string suitable for use in a HTTP field value.

1. Let output be an empty string.

2. For each member_name with a value of (member_value, parameters) in input_dictionary:

    1. Append the result of running Serializing a Key (Section 4.1.1.3) with member's member_name to output.

    2. If member_value is Boolean true:

1. Append the result of running Serializing Parameters (Section 4.1.1.2) with parameters to output.

3. Otherwise:

    1. Append "=" to output.

    2. If member_value is an array, append the result of running Serializing an Inner List (Section 4.1.1.1) with (member_value, parameters) to output.

    3. Otherwise, append the result of running Serializing an Item (Section 4.1.3) with (member_value, parameters) to output.

4. If more members remain in input_dictionary:

    1. Append "," to output.

    2. Append a single SP to output.

3. Return output.

4.1.3. Serializing an Item

Given an Item as bare_item and Parameters as item_parameters, return an ASCII string suitable for use in a HTTP field value.

1. Let output be an empty string.

2. Append the result of running Serializing a Bare Item Section 4.1.3.1 with bare_item to output.

3. Append the result of running Serializing Parameters Section 4.1.1.2 with item_parameters to output.

4. Return output.

4.1.3.1. Serializing a Bare Item

Given an Item as input_item, return an ASCII string suitable for use in a HTTP field value.

1. If input_item is an Integer, return the result of running Serializing an Integer (Section 4.1.4) with input_item.

2. If input_item is a Decimal, return the result of running Serializing a Decimal (Section 4.1.5) with input_item.

3. If input_item is a String, return the result of running Serializing a String (Section 4.1.6) with input_item.

4. If input_item is a Token, return the result of running Serializing a Token (Section 4.1.7) with input_item.

5. If input_item is a Boolean, return the result of running Serializing a Boolean (Section 4.1.9) with input_item.

6. If input_item is a Byte Sequence, return the result of running Serializing a Byte Sequence (Section 4.1.8) with input_item.

7. Otherwise, fail serialization.

4.1.4. Serializing an Integer

Given an Integer as input_integer, return an ASCII string suitable for use in a HTTP field value.

1. If input_integer is not an integer in the range of -999,999,999,999,999 to 999,999,999,999,999 inclusive, fail serialization.

2. Let output be an empty string.

3. If input_integer is less than (but not equal to) 0, append "-" to output.

4. Append input_integer's numeric value represented in base 10 using only decimal digits to output.

5. Return output.

4.1.5. Serializing a Decimal

Given a decimal number as input_decimal, return an ASCII string suitable for use in a HTTP field value.

1. If input_decimal is not a decimal number, fail serialization.

2. If input_decimal has more than three significant digits to the right of the decimal point, round it to three decimal places, rounding the final digit to the nearest value, or to the even value if it is equidistant.

3. If input_decimal has more than 12 significant digits to the left of the decimal point after rounding, fail serialization.

4. Let output be an empty string.

5. If input_decimal is less than (but not equal to) 0, append "-" to output.

6. Append input_decimal's integer component represented in base 10 (using only decimal digits) to output; if it is zero, append "0".

7. Append "." to output.

8. If input_decimal's fractional component is zero, append "0" to output.

9. Otherwise, append the significant digits of input_decimal's fractional component represented in base 10 (using only decimal digits) to output.

10. Return output.

4.1.6. Serializing a String

Given a String as input_string, return an ASCII string suitable for use in a HTTP field value.

1. Convert input_string into a sequence of ASCII characters; if conversion fails, fail serialization.

2. If input_string contains characters in the range %x00-1f or %x7f (i.e., not in VCHAR or SP), fail serialization.

3. Let output be the string DQUOTE.

4. For each character char in input_string:

    1. If char is "\" or DQUOTE:

        1. Append "\" to output.

    2. Append char to output.

5. Append DQUOTE to output.

6. Return output.

4.1.7. Serializing a Token

Given a Token as input_token, return an ASCII string suitable for use in a HTTP field value.

1. Convert input_token into a sequence of ASCII characters; if conversion fails, fail serialization.

2. If the first character of input_token is not ALPHA or "*", or the remaining portion contains a character not in tchar, ":" or "/", fail serialization.

3. Let output be an empty string.

4. Append input_token to output.

5. Return output.

4.1.8. Serializing a Byte Sequence

Given a Byte Sequence as input_bytes, return an ASCII string suitable for use in a HTTP field value.

1. If input_bytes is not a sequence of bytes, fail serialization.

2. Let output be an empty string.

3. Append ":" to output.

4. Append the result of base64-encoding input_bytes as per [RFC4648], Section 4, taking account of the requirements below.

5. Append ":" to output.

6. Return output.

The encoded data is required to be padded with "=", as per [RFC4648], Section 3.2.

Likewise, encoded data SHOULD have pad bits set to zero, as per [RFC4648], Section 3.5, unless it is not possible to do so due to implementation constraints.

4.1.9. Serializing a Boolean

Given a Boolean as input_boolean, return an ASCII string suitable for use in a HTTP field value.

1. If input_boolean is not a boolean, fail serialization.

2. Let output be an empty string.

3. Append "?" to output.

4. If input_boolean is true, append "1" to output.

5. If input_boolean is false, append "0" to output.

6. Return output.

4.2. Parsing Structured Fields

When a receiving implementation parses HTTP fields that are known to be Structured Fields, it is important that care be taken, as there are a number of edge cases that can cause interoperability or even security problems. This section specifies the algorithm for doing so.

Given an array of bytes input_bytes that represents the chosen field's field-value (which is empty if that field is not present), and field_type (one of "dictionary", "list", or "item"), return the parsed header value.

1. Convert input_bytes into an ASCII string input_string; if conversion fails, fail parsing.

2. Discard any leading SP characters from input_string.

3. If field_type is "list", let output be the result of running Parsing a List (Section 4.2.1) with input_string.

4. If field_type is "dictionary", let output be the result of running Parsing a Dictionary (Section 4.2.2) with input_string.

5. If field_type is "item", let output be the result of running Parsing an Item (Section 4.2.3) with input_string.

6. Discard any leading SP characters from input_string.

7. If input_string is not empty, fail parsing.

8. Otherwise, return output.

When generating input_bytes, parsers MUST combine all field lines in the same section (header or trailer) that case-insensitively match the field name into one comma-separated field-value, as per

[RFC7230], Section 3.2.2; this assures that the entire field value is processed correctly.

For Lists and Dictionaries, this has the effect of correctly concatenating all of the field's lines, as long as individual members of the top-level data structure are not split across multiple header instances. The parsing algorithms for both types allow tab characters, since these might be used to combine field lines by some implementations.

Strings split across multiple field lines will have unpredictable results, because comma(s) and whitespace inserted upon combination will become part of the string output by the parser. Since concatenation might be done by an upstream intermediary, the results are not under the control of the serializer or the parser, even when they are both under the control of the same party.

Tokens, Integers, Decimals and Byte Sequences cannot be split across multiple field lines because the inserted commas will cause parsing to fail.

Parsers MAY fail when processing a field value spread across multiple field lines, when one of those lines does not parse as that field. For example, a parsing handling an Example-String field that's defined as a sf-string is allowed to fail when processing this field section:

```
Example-String: "foo
Example-String: bar"
```

If parsing fails - including when calling another algorithm - the entire field value MUST be ignored (i.e., treated as if the field were not present in the section). This is intentionally strict, to improve interoperability and safety, and specifications referencing this document are not allowed to loosen this requirement.

Note that this requirement does not apply to an implementation that is not parsing the field; for example, an intermediary is not required to strip a failing field from a message before forwarding it.

4.2.1. Parsing a List

Given an ASCII string as input_string, return an array of (item_or_inner_list, parameters) tuples. input_string is modified to remove the parsed value.

1. Let members be an empty array.

2. While input_string is not empty:

   1. Append the result of running Parsing an Item or Inner List (Section 4.2.1.1) with input_string to members.

   2. Discard any leading OWS characters from input_string.

   3. If input_string is empty, return members.

   4. Consume the first character of input_string; if it is not ",", fail parsing.

   5. Discard any leading OWS characters from input_string.

   6. If input_string is empty, there is a trailing comma; fail parsing.

3. No structured data has been found; return members (which is empty).

4.2.1.1. Parsing an Item or Inner List

Given an ASCII string as input_string, return the tuple (item_or_inner_list, parameters), where item_or_inner_list can be either a single bare item, or an array of (bare_item, parameters) tuples. input_string is modified to remove the parsed value.

1. If the first character of input_string is "(", return the result of running Parsing an Inner List (Section 4.2.1.2) with input_string.

2. Return the result of running Parsing an Item (Section 4.2.3) with input_string.

4.2.1.2. Parsing an Inner List

Given an ASCII string as input_string, return the tuple (inner_list, parameters), where inner_list is an array of (bare_item, parameters) tuples. input_string is modified to remove the parsed value.

1. Consume the first character of input_string; if it is not "(", fail parsing.

2. Let inner_list be an empty array.

3. While input_string is not empty:

   1. Discard any leading SP characters from input_string.

2. If the first character of input_string is ")":

    1. Consume the first character of input_string.

    2. Let parameters be the result of running Parsing Parameters (Section 4.2.3.2) with input_string.

    3. Return the tuple (inner_list, parameters).

3. Let item be the result of running Parsing an Item (Section 4.2.3) with input_string.

4. Append item to inner_list.

5. If the first character of input_string is not SP or ")", fail parsing.

4. The end of the inner list was not found; fail parsing.

4.2.2. Parsing a Dictionary

Given an ASCII string as input_string, return an ordered map whose values are (item_or_inner_list, parameters) tuples. input_string is modified to remove the parsed value.

1. Let dictionary be an empty, ordered map.

2. While input_string is not empty:

    1. Let this_key be the result of running Parsing a Key (Section 4.2.3.3) with input_string.

    2. If the first character of input_string is "=":

        1. Consume the first character of input_string.

        2. Let member be the result of running Parsing an Item or Inner List (Section 4.2.1.1) with input_string.

    3. Otherwise:

        1. Let value be Boolean true.

        2. Let parameters be the result of running Parsing Parameters Section 4.2.3.2 with input_string.

        3. Let member be the tuple (value, parameters).

4. Add name this_key with value member to dictionary. If dictionary already contains a name this_key (comparing character-for-character), overwrite its value.

5. Discard any leading OWS characters from input_string.

6. If input_string is empty, return dictionary.

7. Consume the first character of input_string; if it is not ",", fail parsing.

8. Discard any leading OWS characters from input_string.

9. If input_string is empty, there is a trailing comma; fail parsing.

3. No structured data has been found; return dictionary (which is empty).

Note that when duplicate Dictionary keys are encountered, this has the effect of ignoring all but the last instance.

4.2.3. Parsing an Item

Given an ASCII string as input_string, return a (bare_item, parameters) tuple. input_string is modified to remove the parsed value.

1. Let bare_item be the result of running Parsing a Bare Item (Section 4.2.3.1) with input_string.

2. Let parameters be the result of running Parsing Parameters (Section 4.2.3.2) with input_string.

3. Return the tuple (bare_item, parameters).

4.2.3.1. Parsing a Bare Item

Given an ASCII string as input_string, return a bare Item. input_string is modified to remove the parsed value.

1. If the first character of input_string is a "-" or a DIGIT, return the result of running Parsing an Integer or Decimal (Section 4.2.4) with input_string.

2. If the first character of input_string is a DQUOTE, return the result of running Parsing a String (Section 4.2.5) with input_string.

3. If the first character of input_string is ":", return the result of running Parsing a Byte Sequence (Section 4.2.7) with input_string.

4. If the first character of input_string is "?", return the result of running Parsing a Boolean (Section 4.2.8) with input_string.

5. If the first character of input_string is an ALPHA or "*", return the result of running Parsing a Token (Section 4.2.6) with input_string.

6. Otherwise, the item type is unrecognized; fail parsing.

4.2.3.2. Parsing Parameters

Given an ASCII string as input_string, return an ordered map whose values are bare Items. input_string is modified to remove the parsed value.

1. Let parameters be an empty, ordered map.

2. While input_string is not empty:

    1. If the first character of input_string is not ";", exit the loop.

    2. Consume a ";" character from the beginning of input_string.

    3. Discard any leading SP characters from input_string.

    4. let param_name be the result of running Parsing a Key (Section 4.2.3.3) with input_string.

    5. Let param_value be Boolean true.

    6. If the first character of input_string is "=":

        1. Consume the "=" character at the beginning of input_string.

        2. Let param_value be the result of running Parsing a Bare Item (Section 4.2.3.1) with input_string.

    7. Append key param_name with value param_value to parameters. If parameters already contains a name param_name (comparing character-for-character), overwrite its value.

3. Return parameters.

Note that when duplicate Parameter keys are encountered, this has the effect of ignoring all but the last instance.

4.2.3.3. Parsing a Key

Given an ASCII string as input_string, return a key. input_string is modified to remove the parsed value.

1. If the first character of input_string is not lcalpha or "*", fail parsing.

2. Let output_string be an empty string.

3. While input_string is not empty:

    1. If the first character of input_string is not one of lcalpha, DIGIT, "_", "-", ".", or "*", return output_string.

    2. Let char be the result of consuming the first character of input_string.

    3. Append char to output_string.

4. Return output_string.

4.2.4. Parsing an Integer or Decimal

Given an ASCII string as input_string, return an Integer or Decimal. input_string is modified to remove the parsed value.

NOTE: This algorithm parses both Integers (Section 3.3.1) and Decimals (Section 3.3.2), and returns the corresponding structure.

1. Let type be "integer".

2. Let sign be 1.

3. Let input_number be an empty string.

4. If the first character of input_string is "-", consume it and set sign to -1.

5. If input_string is empty, there is an empty integer; fail parsing.

6. If the first character of input_string is not a DIGIT, fail parsing.

7. While input_string is not empty:

    1. Let char be the result of consuming the first character of input_string.

    2. If char is a DIGIT, append it to input_number.

    3. Else, if type is "integer" and char is ".":

        1. If input_number contains more than 12 characters, fail parsing.

        2. Otherwise, append char to input_number and set type to "decimal".

    4. Otherwise, prepend char to input_string, and exit the loop.

    5. If type is "integer" and input_number contains more than 15 characters, fail parsing.

    6. If type is "decimal" and input_number contains more than 16 characters, fail parsing.

8. If type is "integer":

    1. Parse input_number as an integer and let output_number be the product of the result and sign.

    2. If output_number is outside the range -999,999,999,999,999 to 999,999,999,999,999 inclusive, fail parsing.

9. Otherwise:

    1. If the final character of input_number is ".", fail parsing.

    2. If the number of characters after "." in input_number is greater than three, fail parsing.

    3. Parse input_number as a decimal number and let output_number be the product of the result and sign.

10. Return output_number.

4.2.5. Parsing a String

Given an ASCII string as input_string, return an unquoted String. input_string is modified to remove the parsed value.

1. Let output_string be an empty string.

2. If the first character of input_string is not DQUOTE, fail parsing.

3. Discard the first character of input_string.

4. While input_string is not empty:

    1. Let char be the result of consuming the first character of input_string.

    2. If char is a backslash ("\"):

        1. If input_string is now empty, fail parsing.

        2. Let next_char be the result of consuming the first character of input_string.

        3. If next_char is not DQUOTE or "\", fail parsing.

        4. Append next_char to output_string.

    3. Else, if char is DQUOTE, return output_string.

    4. Else, if char is in the range %x00-1f or %x7f (i.e., is not in VCHAR or SP), fail parsing.

    5. Else, append char to output_string.

5. Reached the end of input_string without finding a closing DQUOTE; fail parsing.

4.2.6. Parsing a Token

Given an ASCII string as input_string, return a Token. input_string is modified to remove the parsed value.

1. If the first character of input_string is not ALPHA or "*", fail parsing.

2. Let output_string be an empty string.

3. While input_string is not empty:

    1. If the first character of input_string is not in tchar, ":" or "/", return output_string.

2. Let char be the result of consuming the first character of input_string.

3. Append char to output_string.

4. Return output_string.

4.2.7. Parsing a Byte Sequence

Given an ASCII string as input_string, return a Byte Sequence. input_string is modified to remove the parsed value.

1. If the first character of input_string is not ":", fail parsing.

2. Discard the first character of input_string.

3. If there is not a ":" character before the end of input_string, fail parsing.

4. Let b64_content be the result of consuming content of input_string up to but not including the first instance of the character ":".

5. Consume the ":" character at the beginning of input_string.

6. If b64_content contains a character not included in ALPHA, DIGIT, "+", "/" and "=", fail parsing.

7. Let binary_content be the result of Base 64 Decoding [RFC4648] b64_content, synthesizing padding if necessary (note the requirements about recipient behavior below).

8. Return binary_content.

Because some implementations of base64 do not allow rejection of encoded data that is not properly "=" padded (see [RFC4648], Section 3.2), parsers SHOULD NOT fail when "=" padding is not present, unless they cannot be configured to do so.

Because some implementations of base64 do not allow rejection of encoded data that has non-zero pad bits (see [RFC4648], Section 3.5), parsers SHOULD NOT fail when non-zero pad bits are present, unless they cannot be configured to do so.

This specification does not relax the requirements in [RFC4648], Section 3.1 and 3.3; therefore, parsers MUST fail on characters outside the base64 alphabet, and on line feeds in encoded data.

4.2.8. Parsing a Boolean

Given an ASCII string as input_string, return a Boolean. input_string is modified to remove the parsed value.

1. If the first character of input_string is not "?", fail parsing.

2. Discard the first character of input_string.

3. If the first character of input_string matches "1", discard the first character, and return true.

4. If the first character of input_string matches "0", discard the first character, and return false.

5. No value has matched; fail parsing.

5. IANA Considerations

This document has no actions for IANA.

6. Security Considerations

The size of most types defined by Structured Fields is not limited; as a result, extremely large fields could be an attack vector (e.g., for resource consumption). Most HTTP implementations limit the sizes of individual fields as well as the overall header or trailer section size to mitigate such attacks.

It is possible for parties with the ability to inject new HTTP fields to change the meaning of a Structured Field. In some circumstances, this will cause parsing to fail, but it is not possible to reliably fail in all such circumstances.

7. References

7.1. Normative References

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

[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>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[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>.

[RFC7230] 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>.

[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>.

7.2. Informative References

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754-2019, DOI 10.1109/IEEESTD.2019.8766229, ISBN 978-1-5044-5924-2, July 2019, <https://ieeexplore.ieee.org/document/8766229>.

[RFC7231] 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>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

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

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/std63>.

7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

[2] https://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/header-structure

[4] https://github.com/httpwg/structured-field-tests

[5] https://github.com/httpwg/wiki/wiki/Structured-Headers

[6] https://github.com/httpwg/structured-field-tests

Appendix A. Frequently Asked Questions

A.1. Why not JSON?

Earlier proposals for Structured Fields were based upon JSON [RFC8259]. However, constraining its use to make it suitable for HTTP header fields required senders and recipients to implement specific additional handling.

For example, JSON has specification issues around large numbers and objects with duplicate members. Although advice for avoiding these issues is available (e.g., [RFC7493]), it cannot be relied upon.

Likewise, JSON strings are by default Unicode strings, which have a number of potential interoperability issues (e.g., in comparison). Although implementers can be advised to avoid non-ASCII content where unnecessary, this is difficult to enforce.

Another example is JSON's ability to nest content to arbitrary depths. Since the resulting memory commitment might be unsuitable (e.g., in embedded and other limited server deployments), it's necessary to limit it in some fashion; however, existing JSON implementations have no such limits, and even if a limit is specified, it's likely that some field definition will find a need to violate it.

Because of JSON's broad adoption and implementation, it is difficult to impose such additional constraints across all implementations; some deployments would fail to enforce them, thereby harming interoperability. In short, if it looks like JSON, people will be tempted to use a JSON parser / serializer on field values.

Since a major goal for Structured Fields is to improve interoperability and simplify implementation, these concerns led to a format that requires a dedicated parser and serializer.

Additionally, there were widely shared feelings that JSON doesn't "look right" in HTTP fields.

Appendix B. Implementation Notes

A generic implementation of this specification should expose the top-level serialize (Section 4.1) and parse (Section 4.2) functions. They need not be functions; for example, it could be implemented as an object, with methods for each of the different top-level types.

For interoperability, it's important that generic implementations be complete and follow the algorithms closely; see Section 1.1. To aid this, a common test suite is being maintained by the community at https://github.com/httpwg/structured-field-tests [6].

Implementers should note that Dictionaries and Parameters are order-preserving maps. Some fields may not convey meaning in the ordering of these data types, but it should still be exposed so that applications which need to use it will have it available.

Likewise, implementations should note that it's important to preserve the distinction between Tokens and Strings. While most programming languages have native types that map to the other types well, it may be necessary to create a wrapper "token" object or use a parameter on functions to assure that these types remain separate.

The serialization algorithm is defined in a way that it is not strictly limited to the data types defined in Section 3 in every case. For example, Decimals are designed to take broader input and round to allowed values.

Implementations are allowed to limit the allowed size of different structures, subject to the minimums defined for each type. When a structure exceeds an implementation limit, that structure fails parsing or serialisation.

Appendix C. Changes

RFC Editor: Please remove this section before publication.

C.1. Since draft-ietf-httpbis-header-structure-18

- Use "sf-" prefix for ABNF, not "sh-".
- Fix indentation in Dictionary serialisation (#1164).
- Add example for Token; tweak example field names (#1147).
- Editorial improvements.
- Note that exceeding implementation limits implies failure.
- Talk about specifying order of Dictionary members and Parameters, not cardinality.
- Allow (but don't require) parsers to fail when a single field line isn't valid.
- Note that some aspects of Integers and Decimals are not necessarily preserved.
- Allow Lists and Dictionaries to be delimited by OWS, rather than *SP, to make parsing more robust.

C.2. Since draft-ietf-httpbis-header-structure-17

- Editorial improvements.

C.3. Since draft-ietf-httpbis-header-structure-16

- Editorial improvements.
- Discussion on forwards compatibility.

C.4. Since draft-ietf-httpbis-header-structure-15

- Editorial improvements.
- Use HTTP field terminology more consistently, in line with recent changes to HTTP-core.
- String length requirements apply to decoded strings (#1051).
- Correctly round decimals in serialisation (#1043).
- Clarify input to serialisation algorithms (#1055).
- Omitted True dictionary value can have parameters (#1083).

- Keys can now start with '*' (#1068).

C.5. Since draft-ietf-httpbis-header-structure-14

- Editorial improvements.
- Allow empty dictionary values (#992).
- Change value of omitted parameter value to True (#995).
- Explain more about splitting dictionaries and lists across header instances (#997).
- Disallow HTAB, replace OWS with spaces (#998).
- Change byte sequence delimiters from "*" to ":" (#991).
- Allow tokens to start with "*" (#991).
- Change Floats to fixed-precision Decimals (#982).
- Round the fractional component of decimal, rather than truncating it (#982).
- Handle duplicate dictionary and parameter keys by overwriting their values, rather than failing (#997).
- Allow "." in key (#1027).
- Check first character of key in serialisation (#1037).
- Talk about greasing headers (#1015).

C.6. Since draft-ietf-httpbis-header-structure-13

- Editorial improvements.
- Define "structured header name" and "structured header value" terms (#908).
- Corrected text about valid characters in strings (#931).
- Removed most instances of the word "textual", as it was redundant (#915).
- Allowed parameters on Items and Inner Lists (#907).
- Expand the range of characters in token (#961).

- Disallow OWS before ";" delimiter in parameters (#961).

C.7. Since draft-ietf-httpbis-header-structure-12

- Editorial improvements.
- Reworked float serialisation (#896).
- Don't add a trailing space in inner-list (#904).

C.8. Since draft-ietf-httpbis-header-structure-11

- Allow * in key (#844).
- Constrain floats to six digits of precision (#848).
- Allow dictionary members to have parameters (#842).

C.9. Since draft-ietf-httpbis-header-structure-10

- Update abstract (#799).
- Input and output are now arrays of bytes (#662).
- Implementations need to preserve difference between token and string (#790).
- Allow empty dictionaries and lists (#781).
- Change parameterized lists to have primary items (#797).
- Allow inner lists in both dictionaries and lists; removes lists of lists (#816).
- Subsume Parameterised Lists into Lists (#839).

C.10. Since draft-ietf-httpbis-header-structure-09

- Changed Boolean from T/F to 1/0 (#784).
- Parameters are now ordered maps (#765).
- Clamp integers to 15 digits (#737).

C.11. Since draft-ietf-httpbis-header-structure-08

- Disallow whitespace before items properly (#703).
- Created "key" for use in dictionaries and parameters, rather than relying on identifier (#702). Identifiers have a separate minimum supported size.
- Expanded the range of special characters allowed in identifier to include all of ALPHA, ".", ":", and "%" (#702).
- Use "?" instead of "!" to indicate a Boolean (#719).
- Added "Intentionally Strict Processing" (#684).
- Gave better names for referring specs to use in Parameterised Lists (#720).
- Added Lists of Lists (#721).
- Rename Identifier to Token (#725).
- Add implementation guidance (#727).

C.12. Since draft-ietf-httpbis-header-structure-07

- Make Dictionaries ordered mappings (#659).
- Changed "binary content" to "byte sequence" to align with Infra specification (#671).
- Changed "mapping" to "map" for #671.
- Don’t fail if byte sequences aren’t "=" padded (#658).
- Add Booleans (#683).
- Allow identifiers in items again (#629).
- Disallowed whitespace before items (#703).
- Explain the consequences of splitting a string across multiple headers (#686).

C.13. Since draft-ietf-httpbis-header-structure-06

- Add a FAQ.
- Allow non-zero pad bits.
- Explicitly check for integers that violate constraints.

C.14. Since draft-ietf-httpbis-header-structure-05

- Reorganise specification to separate parsing out.
- Allow referencing specs to use ABNF.
- Define serialisation algorithms.
- Refine relationship between ABNF, parsing and serialisation algorithms.

C.15. Since draft-ietf-httpbis-header-structure-04

- Remove identifiers from item.
- Remove most limits on sizes.
- Refine number parsing.

C.16. Since draft-ietf-httpbis-header-structure-03

- Strengthen language around failure handling.

C.17. Since draft-ietf-httpbis-header-structure-02

- Split Numbers into Integers and Floats.
- Define number parsing.
- Tighten up binary parsing and give it an explicit end delimiter.
- Clarify that mappings are unordered.
- Allow zero-length strings.
- Improve string parsing algorithm.
- Improve limits in algorithms.
- Require parsers to combine header fields before processing.

- o Throw an error on trailing garbage.

C.18. Since draft-ietf-httpbis-header-structure-01

- o Replaced with draft-nottingham-structured-headers.

C.19. Since draft-ietf-httpbis-header-structure-00

- o Added signed 64bit integer type.
- o Drop UTF8, and settle on BCP137 ::EmbeddedUnicodeChar for h1-unicode-string.
- o Change h1_blob delimiter to ":" since "'" is valid t_char

Acknowledgements

Many thanks to Matthew Kerwin for his detailed feedback and careful consideration during the development of this specification.

Thanks also to Ian Clelland, Roy Fielding, Anne van Kesteren, Kazuho Oku, Evert Pot, Julian Reschke, Martin Thomson, Mike West, and Jeffrey Yasskin for their contributions.

Authors' Addresses

Mark Nottingham
Fastly

Email: mnot@mnot.net
URI: https://www.mnot.net/

Poul-Henning Kamp
The Varnish Cache Project

Email: phk@varnish-cache.org

HTTP M. Bishop
Internet-Draft Akamai
Intended status: Standards Track N. Sullivan
Expires: November 15, 2020 Cloudflare
M. Thomson
Mozilla
May 14, 2020

Secondary Certificate Authentication in HTTP/2
draft-ietf-httpbis-http2-secondary-certs-06

Abstract

A use of TLS Exported Authenticators is described which enables HTTP/2 clients and servers to offer additional certificate-based credentials after the connection is established. The means by which these credentials are used with requests is defined.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/secondary-certs [3].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

HTTP clients need to know that the content they receive on a connection comes from the origin that they intended to retrieve it from. The traditional form of server authentication in HTTP has been in the form of a single X.509 certificate provided during the TLS ([RFC5246], [RFC8446]) handshake.

Many existing HTTP [RFC7230] servers also have authentication requirements for the resources they serve. Of the bountiful authentication options available for authenticating HTTP requests, client certificates present a unique challenge for resource-specific authentication requirements because of the interaction with the underlying TLS layer.

TLS 1.2 [RFC5246] supports one server and one client certificate on a connection. These certificates may contain multiple identities, but only one certificate may be provided.

Many HTTP servers host content from several origins. HTTP/2 permits clients to reuse an existing HTTP connection to a server provided that the secondary origin is also in the certificate provided during the TLS handshake. In many cases, servers choose to maintain separate certificates for different origins but still desire the benefits of a shared HTTP connection.

1.1. Server Certificate Authentication

Section 9.1.1 of [RFC7540] describes how connections may be used to make requests from multiple origins as long as the server is authoritative for both. A server is considered authoritative for an origin if DNS resolves the origin to the IP address of the server and (for TLS) if the certificate presented by the server contains the origin in the Subject Alternative Names field.

[RFC7838] enables a step of abstraction from the DNS resolution. If both hosts have provided an Alternative Service at hostnames which resolve to the IP address of the server, they are considered authoritative just as if DNS resolved the origin itself to that address. However, the server's one TLS certificate is still required to contain the name of each origin in question.

[RFC8336] relaxes the requirement to perform the DNS lookup if already connected to a server with an appropriate certificate which claims support for a particular origin.

Servers which host many origins often would prefer to have separate certificates for some sets of origins. This may be for ease of certificate management (the ability to separately revoke or renew them), due to different sources of certificates (a CDN acting on behalf of multiple origins), or other factors which might drive this administrative decision. Clients connecting to such origins cannot currently reuse connections, even if both client and server would prefer to do so.

Because the TLS SNI extension is exchanged in the clear, clients might also prefer to retrieve certificates inside the encrypted context. When this information is sensitive, it might be advantageous to request a general-purpose certificate or anonymous ciphersuite at the TLS layer, while acquiring the "real" certificate in HTTP after the connection is established.

1.2. Client Certificate Authentication

For servers that wish to use client certificates to authenticate users, they might request client authentication during or immediately after the TLS handshake. However, if not all users or resources need certificate-based authentication, a request for a certificate has the unfortunate consequence of triggering the client to seek a certificate, possibly requiring user interaction, network traffic, or other time-consuming activities. During this time, the connection is stalled in many implementations. Such a request can result in a poor experience, particularly when sent to a client that does not expect the request.

The TLS 1.3 CertificateRequest can be used by servers to give clients hints about which certificate to offer. Servers that rely on certificate-based authentication might request different certificates for different resources. Such a server cannot use contextual information about the resource to construct an appropriate TLS CertificateRequest message during the initial handshake.

Consequently, client certificates are requested at connection establishment time only in cases where all clients are expected or required to have a single certificate that is used for all resources. Many other uses for client certificates are reactive, that is, certificates are requested in response to the client making a request.

1.2.1. HTTP/1.1 Using TLS 1.2 and Earlier

In HTTP/1.1, a server that relies on client authentication for a subset of users or resources does not request a certificate when the connection is established. Instead, it only requests a client certificate when a request is made to a resource that requires a certificate. TLS 1.2 [RFC5246] accommodates this by permitting the server to request a new TLS handshake, in which the server will request the client's certificate.

Figure 1 shows the server initiating a TLS-layer renegotiation in response to receiving an HTTP/1.1 request to a protected resource.

```
Client                                      Server
   -- (HTTP) GET /protected -------------------> *1
   <---------------------- (TLS) HelloRequest -- *2
   -- (TLS) ClientHello ----------------------->
   <------------------ (TLS) ServerHello, ... --
   <---------------- (TLS) CertificateRequest -- *3
   -- (TLS) ..., Certificate ------------------> *4
   -- (TLS) Finished ------------------------->
   <-------------------------- (TLS) Finished --
   <--------------------------- (HTTP) 200 OK -- *5
```

Figure 1: HTTP/1.1 reactive certificate authentication with TLS 1.2

In this example, the server receives a request for a protected resource (at *1 on Figure 1). Upon performing an authorization check, the server determines that the request requires authentication using a client certificate and that no such certificate has been provided.

The server initiates TLS renegotiation by sending a TLS HelloRequest (at *2). The client then initiates a TLS handshake. Note that some TLS messages are elided from the figure for the sake of brevity.

The critical messages for this example are the server requesting a certificate with a TLS CertificateRequest (*3); this request might use information about the request or resource. The client then provides a certificate and proof of possession of the private key in Certificate and CertificateVerify messages (*4).

When the handshake completes, the server performs any authorization checks a second time. With the client certificate available, it then authorizes the request and provides a response (*5).

1.2.2. HTTP/1.1 Using TLS 1.3

TLS 1.3 [RFC8446] introduces a new client authentication mechanism that allows for clients to authenticate after the handshake has been completed. For the purposes of authenticating an HTTP request, this is functionally equivalent to renegotiation. Figure 2 shows the simpler exchange this enables.

```
Client                                      Server
   -- (HTTP) GET /protected -------------------->
   <---------------- (TLS) CertificateRequest --
   -- (TLS) Certificate, CertificateVerify,
               Finished ----------------------->
   <--------------------------- (HTTP) 200 OK --
```

Figure 2: HTTP/1.1 reactive certificate authentication with TLS 1.3

TLS 1.3 does not support renegotiation, instead supporting direct client authentication. In contrast to the TLS 1.2 example, in TLS 1.3, a server can simply request a certificate.

1.2.3. HTTP/2

An important part of the HTTP/1.1 exchange is that the client is able to easily identify the request that caused the TLS renegotiation. The client is able to assume that the next unanswered request on the connection is responsible. The HTTP stack in the client is then able to direct the certificate request to the application or component that initiated that request. This ensures that the application has the right contextual information for processing the request.

In HTTP/2, a client can have multiple outstanding requests. Without some sort of correlation information, a client is unable to identify which request caused the server to request a certificate.

Thus, the minimum necessary mechanism to support reactive certificate authentication in HTTP/2 is an identifier that can be use to correlate an HTTP request with a request for a certificate. Since streams are used for individual requests, correlation with a stream is sufficient.

[RFC7540] prohibits renegotiation after any application data has been sent. This completely blocks reactive certificate authentication in HTTP/2 using TLS 1.2. If this restriction were relaxed by an extension or update to HTTP/2, such an identifier could be added to TLS 1.2 by means of an extension to TLS. Unfortunately, many TLS 1.2 implementations do not permit application data to continue during a renegotiation. This is problematic for a multiplexed protocol like HTTP/2.

1.3. HTTP-Layer Certificate Authentication

This draft defines HTTP/2 frames to carry the relevant certificate messages, enabling certificate-based authentication of both clients and servers independent of TLS version. This mechanism can be implemented at the HTTP layer without breaking the existing interface between HTTP and applications above it.

This could be done in a naive manner by replicating the TLS messages as HTTP/2 frames on each stream. However, this would create needless redundancy between streams and require frequent expensive signing operations. Instead, TLS Exported Authenticators [I-D.ietf-tls-exported-authenticator] are exchanged on stream zero and other frames incorporate them to particular requests by reference as needed.

TLS Exported Authenticators are structured messages that can be exported by either party of a TLS connection and validated by the other party. Given an established TLS connection, a request can be constructed which describes the desired certificate and an authenticator message can be constructed proving possession of a certificate and a corresponding private key. Both requests and authenticators can be generated by either the client or the server. Exported Authenticators use the message structures from Sections 4.3.2 and 4.4 of [RFC8446], but different parameters.

Each Authenticator is computed using a Handshake Context and Finished MAC Key derived from the TLS session. The Handshake Context is identical for both parties of the TLS connection, while the Finished MAC Key is dependent on whether the Authenticator is created by the client or the server.

Successfully verified Authenticators result in certificate chains, with verified possession of the corresponding private key, which can be supplied into a collection of available certificates. Likewise, descriptions of desired certificates can be supplied into these collections.

Section 2 describes how the feature is employed, defining means to detect support in peers (Section 2.1), make certificates and requests available (Section 2.2), and indicate when streams are blocked waiting on an appropriate certificate (Section 2.3). Section 3 defines the required frame types, which parallel the TLS 1.3 message exchange. Finally, Section 4 defines new error types which can be used to notify peers when the exchange has not been successful.

1.4. Terminology

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.

2. Discovering Additional Certificates at the HTTP/2 Layer

A certificate chain with proof of possession of the private key corresponding to the end-entity certificate is sent as a sequence of "CERTIFICATE" frames (see Section 3.4) on stream zero. Once the holder of a certificate has sent the chain and proof, this certificate chain is cached by the recipient and available for future use. Clients can proactively indicate the certificate they intend to use on each request using an unsolicited "USE_CERTIFICATE" frame, if desired. The previously-supplied certificates are available for reference without having to resend them.

Otherwise, the server uses a "CERTIFICATE_REQUEST" frame to describe a class of certificates on stream zero, then uses "CERTIFICATE_NEEDED" frames to associate these with individual requests. The client responds with a "USE_CERTIFICATE" frame indicating the certificate which should be used to satisfy the request.

Data sent by each peer is correlated by the ID given in each frame. This ID is unrelated to values used by the other peer, even if each uses the same ID in certain cases. "USE_CERTIFICATE" frames indicate whether they are sent proactively or are in response to a "CERTIFICATE_NEEDED" frame.

2.1. Indicating Support for HTTP-Layer Certificate Authentication

Clients and servers that will accept requests for HTTP-layer certificate authentication indicate this using the HTTP/2 "SETTINGS_HTTP_CLIENT_CERT_AUTH" (0xSETTING-TBD1) and "SETTINGS_HTTP_SERVER_CERT_AUTH" (0xSETTING-TBD2) settings.

The initial value for both settings is 0, indicating that the peer does not support HTTP-layer certificate authentication. If a peer does support HTTP-layer certificate authentication, one or both of the values is non-zero. "SETTINGS_HTTP_CLIENT_CERT_AUTH" indicates that servers can use certificates for client authentication, while "SETTINGS_HTTP_SERVER_CERT_AUTH" indicates that servers are able to offer additional certificates to demonstrate control over other origin hostnames.

In order to ensure that the TLS connection is direct to the server, rather than via a TLS-terminating proxy, each side will separately compute and confirm the value of these settings. The setting values are derived from a TLS exporter (see Section 7.5 of [RFC8446] and [RFC5705] for more details on exporters). Clients MUST NOT use an early exporter during their 0-RTT flight, but MUST send an updated SETTINGS frame using a regular exporter after the TLS handshake completes.

The exporter is constructed with the following input:

- Label:

  * "EXPORTER HTTP CERTIFICATE client" for clients

  * "EXPORTER HTTP CERTIFICATE server" for servers

- Context: Empty

- Length: Eight bytes

The value of the exporter is split into two four-byte values. The first four bytes are used for the "SETTINGS_HTTP_CLIENT_CERT_AUTH" value, while the following four bytes are used for the "SETTINGS_HTTP_SERVER_CERT_AUTH" value.

Each is converted to a setting value as:

Exporter fragment | 0x80000000

That is, the most significant bit will always be set, regardless of the value of the exporter. Each endpoint will compute the expected values from their peer. If the setting is not received, or if the value received is not the expected value, the frames defined in this document SHOULD NOT be sent in the indicated direction.

2.2. Making Certificates or Requests Available

When both peers have advertised support for HTTP-layer certificates in a given direction as in Section 2.1, the indicated endpoint can supply additional certificates into the connection at any time. That is, if both endpoints have sent "SETTINGS_HTTP_SERVER_CERT_AUTH" and validated the value received from the peer, the server may send certificates. If both endpoints have sent "SETTINGS_HTTP_CLIENT_CERT_AUTH" and validated the value received from the peer, the client may send certificates.

Implementations which predict a certificate will be required could supply the certificate before being asked. These certificates are available for reference by future "USE_CERTIFICATE" frames.

Certificates supplied by servers can be considered by clients without further action by the server. A server SHOULD NOT send certificates which do not cover origins which it is prepared to service on the current connection, but MAY use the ORIGIN frame [RFC8336] to indicate that not all covered origins will be served.

Certificates supplied by clients MUST NOT be considered by servers when processing a request unless the client explicitly authorizes their use. Clients MAY send "USE_CERTIFICATE" frame with the "UNSOLICITED" flag set to indicate that an available certificate should be considered on a new request.

```
Client                                      Server
   <------------------ (stream 0) CERTIFICATE --
   ...
   -- (stream N) GET /from-new-origin --------->
   <----------------------- (stream N) 200 OK --
```

Figure 3: Proactive server authentication

```
Client                                      Server
   -- (stream 0) CERTIFICATE ------------------>
   -- (stream 0) USE_CERTIFICATE (S=1) -------->
   -- (stream 0) USE_CERTIFICATE (S=3) -------->
   -- (streams 1,3) GET /protected ------------>
   <-------------------- (streams 1,3) 200 OK --
```

Figure 4: Proactive client authentication

Likewise, a party can supply a "CERTIFICATE_REQUEST" that outlines parameters of a certificate they might request in the future. Upon receipt of a "CERTIFICATE_REQUEST", endpoints SHOULD provide a corresponding certificate in anticipation of a request shortly being blocked. Clients MAY wait for a "CERTIFICATE_NEEDED" frame to assist in associating the certificate request with a particular HTTP transaction.

2.3. Requiring Certificate Authentication

2.3.1. Requiring Additional Server Certificates

As defined in [RFC7540], when a client finds that an https:// origin (or Alternative Service [RFC7838]) to which it needs to make a request has the same IP address as a server to which it is already connected, it MAY check whether the TLS certificate provided contains the new origin as well, and if so, reuse the connection.

If the TLS certificate does not contain the new origin, but the server has claimed support for that origin (with an ORIGIN frame, see [RFC8336]) and advertised support for HTTP-layer server certificates (see Section 2.1), the client MAY send a "CERTIFICATE_REQUEST" frame describing the desired origin. The client then sends a "CERTIFICATE_NEEDED" frame for stream zero referencing the request, indicating that the connection cannot be used for that origin until the certificate is provided.

If the server does not have the desired certificate, it MUST send an Empty Authenticator, as described in Section 5 of [I-D.ietf-tls-exported-authenticator], in a "CERTIFICATE" frame in response to the request, followed by a "USE_CERTIFICATE" frame for stream zero which references the Empty Authenticator.

If a server has not advertised support for HTTP-layer certificates, fails to provide a requested certificate, or provides a certificate which is unacceptable to the client, the client MUST NOT send any requests for resources in that origin on the current connection. The client MAY open a new connection in an effort to reach an authoritative server.

```
Client                                      Server
   <----------------------- (stream 0) ORIGIN --
   -- (stream 0) CERTIFICATE_REQUEST ---------->
   -- (stream 0) CERTIFICATE_NEEDED (S=0) ----->
   <------------------ (stream 0) CERTIFICATE --
   <-------- (stream 0) USE_CERTIFICATE (S=0) --
   -- (stream N) GET /from-new-origin --------->
   <----------------------- (stream N) 200 OK --
```

Figure 5: Client-requested certificate

If a client receives a "PUSH_PROMISE" referencing an origin for which it has not yet received the server’s certificate, this is a stream error on the push stream; see section 8.2 of [RFC7540]. To avoid this, servers MUST supply the associated certificates before pushing resources from a different origin.

2.3.2. Requiring Additional Client Certificates

Likewise, the server sends a "CERTIFICATE_NEEDED" frame for each stream where certificate authentication is required. The client answers with a "USE_CERTIFICATE" frame indicating the certificate to use on that stream. If the request parameters or the responding certificate are not already available, they will need to be sent as described in Section 2.2 as part of this exchange.

```
Client                                      Server
   <---------- (stream 0) CERTIFICATE_REQUEST --
   ...
   -- (stream N) GET /protected --------------->
   <----- (stream 0) CERTIFICATE_NEEDED (S=N) --
   -- (stream 0) CERTIFICATE ------------------>
   -- (stream 0) USE_CERTIFICATE (S=N) -------->
   <----------------------- (stream N) 200 OK --
```

Figure 6: Reactive certificate authentication

If the client does not have the desired certificate, it instead sends an Empty Authenticator, as described in Section 5 of [I-D.ietf-tls-exported-authenticator], in a "CERTIFICATE" frame in response to the request, followed by a "USE_CERTIFICATE" frame which references the Empty Authenticator.

If the client has not advertised support for HTTP-layer certificates, fails to provide a requested certificate, or provides a certificate the server is unable to verify, the server processes the request based solely on the certificate provided during the TLS handshake, if any. This might result in an error response via HTTP, such as a status code 403 (Not Authorized).

3. Certificates Frames for HTTP/2

The "CERTIFICATE_REQUEST" and "CERTIFICATE_NEEDED" frames are correlated by their "Request-ID" field. Subsequent "CERTIFICATE_NEEDED" frames with the same "Request-ID" value MAY be sent for other streams where the sender is expecting a certificate with the same parameters.

The "CERTIFICATE", and "USE_CERTIFICATE" frames are correlated by their "Cert-ID" field. Subsequent "USE_CERTIFICATE" frames with the same "Cert-ID" MAY be sent in response to other "CERTIFICATE_NEEDED" frames and refer to the same certificate.

"CERTIFICATE_NEEDED" and "USE_CERTIFICATE" frames are correlated by the Stream ID they reference. Unsolicited "USE_CERTIFICATE" frames are not responses to "CERTIFICATE_NEEDED" frames; otherwise, each "USE_CERTIFICATE" frame for a stream is considered to respond to a "CERTIFICATE_NEEDED" frame for the same stream in sequence.

```
+---------+           +---------+
| REQUEST |           |   CERT  |
+---------+           +---------+
     |                     |
     | Request-ID          | Cert-ID
     |                     |
     v                     v
+---------+ Stream ID +---------+
| NEEDED  |---------->|   USE   |
+---------+           +---------+
```

Figure 7: Frame correlation

"Request-ID" and "Cert-ID" are independent and sender-local. The use of the same value by the other peer or in the other context does not imply any correlation between these frames. These values MUST be unique per sender for each space over the lifetime of the connection.

3.1. The CERTIFICATE_NEEDED Frame

The "CERTIFICATE_NEEDED" frame (0xFRAME-TBD1) is sent on stream zero to indicate that the HTTP request on the indicated stream is blocked pending certificate authentication. The frame includes stream ID and a request identifier which can be used to correlate the stream with a previous "CERTIFICATE_REQUEST" frame sent on stream zero. The "CERTIFICATE_REQUEST" describes the certificate the sender requires to make progress on the stream in question.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-------------------------------------------------------------+
|R|                        Stream ID (31)                       |
+-+-----------------------------+-------------------------------+
|        Request-ID (16)        |
+-------------------------------+
```

Figure 8: CERTIFICATE_NEEDED frame payload

The "CERTIFICATE_NEEDED" frame contains 6 octets. The first four octets indicate the Stream ID of the affected stream. The following two octets are the authentication request identifier, "Request-ID". A peer that receives a "CERTIFICATE_NEEDED" of any other length MUST treat this as a stream error of type "PROTOCOL_ERROR". Frames with identical request identifiers refer to the same "CERTIFICATE_REQUEST".

A server MAY send multiple "CERTIFICATE_NEEDED" frames for the same stream. If a server requires that a client provide multiple certificates before authorizing a single request, each required certificate MUST be indicated with a separate "CERTIFICATE_NEEDED" frame, each of which MUST have a different request identifier (referencing different "CERTIFICATE_REQUEST" frames describing each required certificate). To reduce the risk of client confusion, servers SHOULD NOT have multiple outstanding "CERTIFICATE_NEEDED" frames for the same stream at any given time.

Clients MUST only send multiple "CERTIFICATE_NEEDED" frames for stream zero. Multiple "CERTIFICATE_NEEDED" frames on any other stream MUST be considered a stream error of type "PROTOCOL_ERROR".

The "CERTIFICATE_NEEDED" frame MUST NOT be sent to a client which has not advertised the "SETTINGS_HTTP_CLIENT_CERT_AUTH", or to a server which has not advertised the "SETTINGS_HTTP_SERVER_CERT_AUTH" setting. An endpoint which receives a "CERTIFICATE_NEEDED" frame but did not advertise support MAY treat this as a connection error of type "CERTIFICATE_WITHOUT_CONSENT".

The "CERTIFICATE_NEEDED" frame MUST NOT reference a stream in the "half-closed (local)" or "closed" states [RFC7540]. A client that receives a "CERTIFICATE_NEEDED" frame for a stream which is not in a valid state SHOULD treat this as a stream error of type "PROTOCOL_ERROR".

3.2. The USE_CERTIFICATE Frame

The "USE_CERTIFICATE" frame (0xFRAME-TBD4) is sent on stream zero to indicate which certificate is being used on a particular request stream.

The "USE_CERTIFICATE" frame defines a single flag:

UNSOLICITED (0x01): Indicates that no "CERTIFICATE_NEEDED" frame has yet been received for this stream.

The payload of the "USE_CERTIFICATE" frame is as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-------------------------------------------------------------+
|R|                        Stream ID (31)                       |
+-+-----------------------------+-------------------------------+
|         [Cert-ID (16)]        |
+-------------------------------+
```

Figure 9: USE_CERTIFICATE frame payload

The first four octets indicate the Stream ID of the affected stream. The following two octets, if present, contain the two-octet "Cert-ID" of the certificate the sender wishes to use. This MUST be the ID of a certificate for which proof of possession has been presented in a "CERTIFICATE" frame. Recipients of a "USE_CERTIFICATE" frame of any other length MUST treat this as a stream error of type "PROTOCOL_ERROR". Frames with identical certificate identifiers refer to the same certificate chain.

A "USE_CERTIFICATE" frame which omits the Cert-ID refers to the certificate provided at the TLS layer, if any. If no certificate was provided at the TLS layer, the stream should be processed with no authentication, likely returning an authentication-related error at the HTTP level (e.g. 403) for servers or routing the request to a new connection for clients.

The "UNSOLICITED" flag MAY be set by clients on the first "USE_CERTIFICATE" frame referring to a given stream. This permits a client to proactively indicate which certificate should be used when processing a new request. When such an unsolicited indication refers to a request that has not yet been received, servers SHOULD cache the indication briefly in anticipation of the request.

Receipt of more than one unsolicited "USE_CERTIFICATE" frame or an unsolicited "USE_CERTIFICATE" frame which is not the first in reference to a given stream MUST be treated as a stream error of type "CERTIFICATE_OVERUSED".

Each "USE_CERTIFICATE" frame which is not marked as unsolicited is considered to respond in order to the "CERTIFICATE_NEEDED" frames for the same stream. If a "USE_CERTIFICATE" frame is received for which a "CERTIFICATE_NEEDED" frame has not been sent, this MUST be treated as a stream error of type "CERTIFICATE_OVERUSED".

Receipt of a "USE_CERTIFICATE" frame with an unknown "Cert-ID" MUST result in a stream error of type "PROTOCOL_ERROR".

The referenced certificate chain needs to conform to the requirements expressed in the "CERTIFICATE_REQUEST" to the best of the sender's ability, or the recipient is likely to reject it as unsuitable despite properly validating the authenticator. If the recipient considers the certificate unsuitable, it MAY at its discretion either return an error at the HTTP semantic layer, or respond with a stream error [RFC7540] on any stream where the certificate is used. Section 4 defines certificate-related error codes which might be applicable.

3.3. The CERTIFICATE_REQUEST Frame

The "CERTIFICATE_REQUEST" frame (id=0xFRAME-TBD2) provides an exported authenticator request message from the TLS layer that specifies a desired certificate. This describes the certificate the sender wishes to have presented.

The "CERTIFICATE_REQUEST" frame SHOULD NOT be sent to a client which has not advertised the "SETTINGS_HTTP_CLIENT_CERT_AUTH", or to a server which has not advertised the "SETTINGS_HTTP_SERVER_CERT_AUTH" setting.

The "CERTIFICATE_REQUEST" frame MUST be sent on stream zero. A "CERTIFICATE_REQUEST" frame received on any other stream MUST be rejected with a stream error of type "PROTOCOL_ERROR".

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------------------------------+-------------------------------+
|        Request-ID (16)        |          Request (?)        ...
+---------------------------------------------------------------+
```

Figure 10: CERTIFICATE_REQUEST frame payload

The frame contains the following fields:

Request-ID: "Request-ID" is a 16-bit opaque identifier used to correlate subsequent certificate-related frames with this request. The identifier MUST be unique in the session for the sender.

Request: An exported authenticator request, generated using the "request" API described in [I-D.ietf-tls-exported-authenticator]. See Section 3.4.1 for more details on the input to this API.

3.3.1. Exported Authenticator Request Characteristics

The Exported Authenticator "request" API defined in [I-D.ietf-tls-exported-authenticator] takes as input a set of desired certificate characteristics and a "certificate_request_context", which needs to be unpredictable. When generating exported authenticators for use with this extension, the "certificate_request_context" MUST contain both the two-octet Request-ID as well as at least 96 bits of additional entropy.

Upon receipt of a "CERTIFICATE_REQUEST" frame, the recipient MUST verify that the first two octets of the authenticator's "certificate_request_context" matches the Request-ID presented in the frame.

The TLS library on the authenticating peer will provide mechanisms to select an appropriate certificate to respond to the transported request. TLS libraries on servers MUST be able to recognize the "server_name" extension ([RFC6066]) at a minimum. Clients MUST always specify the desired origin using this extension, though other extensions MAY also be included.

3.4. The CERTIFICATE Frame

The "CERTIFICATE" frame (id=0xFRAME-TBD3) provides an exported authenticator message from the TLS layer that provides a chain of certificates, associated extensions and proves possession of the private key corresponding to the end-entity certificate.

The "CERTIFICATE" frame defines two flags:

TO_BE_CONTINUED (0x01): Indicates that the exported authenticator spans more than one frame.

UNSOLICITED (0x02): Indicates that the exported authenticator does not contain a Request-ID.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------------------------------+-------------------------------+
|          Cert-ID (16)         |        Request-ID (16)        |
+-------------------------------+-------------------------------+
|                   Authenticator Fragment (*)                ...
+---------------------------------------------------------------+
```

Figure 11: CERTIFICATE frame payload

The frame contains the following fields:

Cert-ID: "Cert-ID" is a 16-bit opaque identifier used to correlate other certificate- related frames with this exported authenticator fragment.

Request-ID: "Request-ID" is an optional 16-bit opaque identifier used to correlate this exported authenticator with the request which triggered it, if any. This field is present only if the "UNSOLICITED" flag is not set.

Authenticator Fragment: A portion of the opaque data returned from the TLS connection exported authenticator "authenticate" API. See Section 3.4.1 for more details on the input to this API.

An exported authenticator is transported in zero or more "CERTIFICATE" frames with the "TO_BE_CONTINUED" flag set, followed by one "CERTIFICATE" frame with the "TO_BE_CONTINUED" flag unset. Each of these frames contains the same "Cert-ID" field, permitting them to be associated with each other. Receipt of any "CERTIFICATE" frame with the same "Cert-ID" following the receipt of a "CERTIFICATE" frame with "TO_BE_CONTINUED" unset MUST be treated as a connection error of type "PROTOCOL_ERROR".

If the "UNSOLICITED" flag is not set, the "CERTIFICATE" frame also contains a Request-ID indicating the certificate request which caused this exported authenticator to be generated. The value of this flag and the contents of the Request-ID field MUST NOT differ between frames with the same Cert-ID.

Upon receiving a complete series of "CERTIFICATE" frames, the receiver may validate the Exported Authenticator value by using the exported authenticator API. This returns either an error indicating that the message was invalid, or the certificate chain and extensions used to create the message.

The "CERTIFICATE" frame MUST be sent on stream zero. A "CERTIFICATE" frame received on any other stream MUST be rejected with a stream error of type "PROTOCOL_ERROR".

3.4.1. Exported Authenticator Characteristics

The Exported Authenticator API defined in [I-D.ietf-tls-exported-authenticator] takes as input a request, a set of certificates, and supporting information about the certificate (OCSP, SCT, etc.). The result is an opaque token which is used when generating the "CERTIFICATE" frame.

Upon receipt of a "CERTIFICATE" frame, an endpoint MUST perform the following steps to validate the token it contains:

- Verify that either the "UNSOLICITED" flag is set (clients only) or that the Request-ID field contains the Request-ID of a previously-sent "CERTIFICATE_REQUEST" frame.

- Using the "get context" API, retrieve the "certificate_request_context" used to generate the authenticator, if any. Verify that the "certificate_request_context" begins with the supplied Request-ID, if any.

- Use the "validate" API to confirm the validity of the authenticator with regard to the generated request (if any).

If the authenticator cannot be validated, this SHOULD be treated as a connection error of type "CERTIFICATE_UNREADABLE".

Once the authenticator is accepted, the endpoint can perform any other checks for the acceptability of the certificate itself. Clients MUST NOT accept any end-entity certificate from an exported authenticator which does not contain the Required Domain extension; see Section 5 and Section 6.1.

4. Indicating Failures During HTTP-Layer Certificate Authentication

Because this draft permits certificates to be exchanged at the HTTP framing layer instead of the TLS layer, several certificate-related errors which are defined at the TLS layer might now occur at the HTTP framing layer.

There are two classes of errors which might be encountered, and they are handled differently.

4.1. Misbehavior

This category of errors could indicate a peer failing to follow restrictions in this document, or might indicate that the connection is not fully secure. These errors are fatal to stream or connection, as appropriate.

CERTIFICATE_OVERUSED (0xERROR-TBD1): More certificates were used on a request than were requested

CERTIFICATE_WITHOUT_CONSENT (0xERROR-TBD2): A CERTIFICATE_NEEDED frame was received by a peer which did not indicate support for this extension.

CERTIFICATE_UNREADABLE (0xERROR-TBD3): An exported authenticator could not be validated.

4.2. Invalid Certificates

Unacceptable certificates (expired, revoked, or insufficient to satisfy the request) are not treated as stream or connection errors. This is typically not an indication of a protocol failure. Servers SHOULD process requests with the indicated certificate, likely resulting in a "4XX"-series status code in the response. Clients SHOULD establish a new connection in an attempt to reach an authoritative server.

5. Required Domain Certificate Extension

The Required Domain extension allows certificates to limit their use with Secondary Certificate Authentication. A client MUST verify that the server has proven ownership of the indicated identity before accepting the limited certificate over Secondary Certificate Authentication.

The identity in this extension is a restriction asserted by the requester of the certificate and is not verified by the CA. Conforming CAs SHOULD mark the requiredDomain extension as non-critical. Conforming CAs MUST require the presence of a CAA record [RFC6844] prior to issuing a certificate with this extension. Because a Required Domain value of "*" has a much higher risk of reuse if compromised, conforming Certificate Authorities are encouraged to require more extensive verification prior to issuing such a certificate.

The required domain is represented as a GeneralName, as specified in Section 4.2.1.6 of [RFC5280]. Unlike the subject field, conforming CAs MUST NOT issue certificates with a requiredDomain extension containing empty GeneralName fields. Clients that encounter such a certificate when processing a certification path MUST consider the certificate invalid.

The wildcard character "_" MAY be used to represent that any previously authenticated identity is acceptable. This character MUST be the entirety of the name if used and MUST have a type of "dNSName". (That is, "_" is acceptable, but "_.com" and "w_.example.com" are not).

```
id-ce-requiredDomain OBJECT IDENTIFIER ::=  { id-ce TBD1 }

RequiredDomain ::= GeneralName
```

6. Security Considerations

This mechanism defines an alternate way to obtain server and client certificates other than in the initial TLS handshake. While the signature of exported authenticator values is expected to be equally secure, it is important to recognize that a vulnerability in this code path is at least equal to a vulnerability in the TLS handshake.

6.1. Impersonation

This mechanism could increase the impact of a key compromise. Rather than needing to subvert DNS or IP routing in order to use a compromised certificate, a malicious server now only needs a client to connect to _some_ HTTPS site under its control in order to present the compromised certificate. Clients SHOULD consult DNS for hostnames presented in secondary certificates if they would have done so for the same hostname if it were present in the primary certificate.

As recommended in [RFC8336], clients opting not to consult DNS ought to employ some alternative means to increase confidence that the certificate is legitimate.

One such means is the Required Domain certificate extension defined in {extension}. Clients MUST require that server certificates presented via this mechanism contain the Required Domain extension and require that a certificate previously accepted on the connection (including the certificate presented in TLS) lists the Required Domain in the Subject field or the Subject Alternative Name extension.

As noted in the Security Considerations of [I-D.ietf-tls-exported-authenticator], it is difficult to formally prove that an endpoint is jointly authoritative over multiple certificates, rather than individually authoritative on each certificate. As a result, clients MUST NOT assume that because one origin was previously colocated with another, those origins will be reachable via the same endpoints in the future. Clients MUST NOT consider previous secondary certificates to be validated after TLS session resumption. However, clients MAY proactively query for previously-presented secondary certificates.

6.2. Fingerprinting

This draft defines a mechanism which could be used to probe servers for origins they support, but opens no new attack versus making repeat TLS connections with different SNI values. Servers SHOULD impose similar denial-of-service mitigations (e.g. request rate limits) to "CERTIFICATE_REQUEST" frames as to new TLS connections.

While the extensions in the "CERTIFICATE_REQUEST" frame permit the sender to enumerate the acceptable Certificate Authorities for the requested certificate, it might not be prudent (either for security or data consumption) to include the full list of trusted Certificate Authorities in every request. Senders, particularly clients, SHOULD send only the extensions that narrowly specify which certificates would be acceptable.

Servers can also learn information about clients using this mechanism. The hostnames a user agent finds interesting and retrieves certificates for might indicate origins the user has previously accessed.

6.3. Denial of Service

Failure to provide a certificate for a stream after receiving "CERTIFICATE_NEEDED" blocks processing, and SHOULD be subject to standard timeouts used to guard against unresponsive peers.

Validating a multitude of signatures can be computationally expensive, while generating an invalid signature is computationally cheap. Implementations will require checks for attacks from this direction. Invalid exported authenticators SHOULD be treated as a session error, to avoid further attacks from the peer, though an implementation MAY instead disable HTTP-layer certificates for the current connection instead.

6.4. Persistence of Service

CNAME records in the DNS are frequently used to delegate authority for an origin to a third-party provider. This delegation can be changed without notice, even to the third-party provider, simply by modifying the CNAME record in question.

After the owner of the domain has redirected traffic elsewhere by changing the CNAME, new connections will not arrive for that origin, but connections which are properly directed to this provider for other origins would continue to claim control of this origin (via ORIGIN frame and Secondary Certificates). This is proper behavior based on the third-party provider's configuration, but would likely not be what is intended by the owner of the origin.

This is not an issue which can be mitigated by the protocol, but something about which third-party providers SHOULD educate their customers before using the features described in this document.

6.5. Confusion About State

Implementations need to be aware of the potential for confusion about the state of a connection. The presence or absence of a validated certificate can change during the processing of a request, potentially multiple times, as "USE_CERTIFICATE" frames are received. A server that uses certificate authentication needs to be prepared to reevaluate the authorization state of a request as the set of certificates changes.

7. IANA Considerations

This draft adds entries in three registries.

The feature negotiation settings is registered in Section 7.1. Four frame types are registered in Section 7.2. Six error codes are registered in Section 7.3.

7.1. HTTP/2 Settings

The SETTINGS_HTTP_CLIENT_CERT_AUTH and SETTINGS_HTTP_SERVER_CERT_AUTH settings are registered in the "HTTP/2 Settings" registry established in [RFC7540].

Name	Code	Initial Value	Specification
HTTP_CLIENT_CERT_AUTH	0xSETTING-TBD1	0	Section 2.1
HTTP_SERVER_CERT_AUTH	0xSETTING-TBD2	0	Section 2.1

7.2. New HTTP/2 Frames

Four new frame types are registered in the "HTTP/2 Frame Types" registry established in [RFC7540]. The entries in the following table are registered by this document.

Frame Type	Code	Specification
CERTIFICATE_NEEDED	0xFRAME-TBD1	Section 3.1
CERTIFICATE_REQUEST	0xFRAME-TBD2	Section 3.3
CERTIFICATE	0xFRAME-TBD3	Section 3.4
USE_CERTIFICATE	0xFRAME-TBD4	Section 3.2

7.3. New HTTP/2 Error Codes

Six new error codes are registered in the "HTTP/2 Error Code" registry established in [RFC7540]. The entries in the following table are registered by this document.

Name	Code	Specification
CERTIFICATE_OVERUSED	0xERROR-TBD1	Section 4
CERTIFICATE_WITHOUT_CONSENT	0xERROR-TBD2	Section 4
CERTIFICATE_UNREADABLE	0xERROR-TBD3	Section 4

8. References

8.1. Normative References

[I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-11 (work in progress), December 2019.

[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>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6844] Hallam-Baker, P. and R. Stradling, "DNS Certification Authority Authorization (CAA) Resource Record", RFC 6844, DOI 10.17487/RFC6844, January 2013, <https://www.rfc-editor.org/info/rfc6844>.

[RFC7230] 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>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[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>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

8.2. Informative References

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[RFC8336] Nottingham, M. and E. Nygren, "The ORIGIN HTTP/2 Frame", RFC 8336, DOI 10.17487/RFC8336, March 2018, <https://www.rfc-editor.org/info/rfc8336>.

8.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

[2] http://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/secondary-certs

Appendix A. Change Log

RFC Editor’s Note: Please remove this section prior to publication of a final version of this document.

A.1. Since draft-ietf-httpbis-http2-secondary-certs-04:

Editorial updates only.

A.2. Since draft-ietf-httpbis-http2-secondary-certs-03:

- "CERTIFICATE_REQUEST" frames contain the Request-ID, which MUST be checked against the "certificate_request_context" of the Exported Authenticator Request

- "CERTIFICATE" frames contain the Request-ID to which they respond, unless the UNSOLICITED flag is set

- The Required Domain extension is defined for certificates, which must be present for certificates presented by servers

A.3. Since draft-ietf-httpbis-http2-secondary-certs-02:

Editorial updates only.

A.4. Since draft-ietf-httpbis-http2-secondary-certs-01:

- Clients can send "CERTIFICATE_NEEDED" for stream 0 rather than speculatively reserving a stream for an origin.

- Use SETTINGS to disable when a TLS-terminating proxy is present (#617,#651)

A.5. Since draft-ietf-httpbis-http2-secondary-certs-00:

- All frames sent on stream zero; replaced "AUTOMATIC_USE" on "CERTIFICATE" with "UNSOLICITED" on "USE_CERTIFICATE". (#482,#566)

- Use Exported Requests from the TLS Exported Authenticators draft; eliminate facilities for expressing certificate requirements in "CERTIFICATE_REQUEST" frame. (#481)

A.6. Since draft-bishop-httpbis-http2-additional-certs-05:

- Adopted as draft-ietf-httpbis-http2-secondary-certs

Acknowledgements

Eric Rescorla pointed out several failings in an earlier revision. Andrei Popov contributed to the TLS considerations.

A substantial portion of Mike's work on this draft was supported by Microsoft during his employment there.

Authors' Addresses

Mike Bishop
Akamai

Email: mbishop@evequefou.be

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

HTTP M. West, Ed.
Internet-Draft Google, Inc
Obsoletes: 6265 (if approved) J. Wilander, Ed.
Intended status: Standards Track Apple, Inc
Expires: October 22, 2020 April 20, 2020

Cookies: HTTP State Management Mechanism
draft-ietf-httpbis-rfc6265bis-06

Abstract

This document defines the HTTP Cookie and Set-Cookie header fields. These header fields can be used by HTTP servers to store state (called cookies) at HTTP user agents, letting the servers maintain a stateful session over the mostly stateless HTTP protocol. Although cookies have many historical infelicities that degrade their security and privacy, the Cookie and Set-Cookie header fields are widely used on the Internet. This document obsoletes RFC 6265.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/6265bis [3].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document defines the HTTP Cookie and Set-Cookie header fields. Using the Set-Cookie header field, an HTTP server can pass name/value pairs and associated metadata (called cookies) to a user agent. When the user agent makes subsequent requests to the server, the user agent uses the metadata and other information to determine whether to return the name/value pairs in the Cookie header.

Although simple on their surface, cookies have a number of complexities. For example, the server indicates a scope for each cookie when sending it to the user agent. The scope indicates the maximum amount of time in which the user agent should return the cookie, the servers to which the user agent should return the cookie, and the URI schemes for which the cookie is applicable.

For historical reasons, cookies contain a number of security and privacy infelicities. For example, a server can indicate that a given cookie is intended for "secure" connections, but the Secure attribute does not provide integrity in the presence of an active network attacker. Similarly, cookies for a given host are shared across all the ports on that host, even though the usual "same-origin policy" used by web browsers isolates content retrieved via different ports.

There are two audiences for this specification: developers of cookie-generating servers and developers of cookie-consuming user agents.

To maximize interoperability with user agents, servers SHOULD limit themselves to the well-behaved profile defined in Section 4 when generating cookies.

User agents MUST implement the more liberal processing rules defined in Section 5, in order to maximize interoperability with existing servers that do not conform to the well-behaved profile defined in Section 4.

This document specifies the syntax and semantics of these headers as they are actually used on the Internet. In particular, this document does not create new syntax or semantics beyond those in use today. The recommendations for cookie generation provided in Section 4 represent a preferred subset of current server behavior, and even the more liberal cookie processing algorithm provided in Section 5 does not recommend all of the syntactic and semantic variations in use today. Where some existing software differs from the recommended protocol in significant ways, the document contains a note explaining the difference.

This document obsoletes [RFC6265].

2. Conventions

2.1. Conformance Criteria

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("MUST", "SHOULD", "MAY", etc.) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant.

2.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

The following core rules are included by reference, as defined in [RFC5234], Appendix B.1: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTLs (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), NUL (null octet), OCTET (any 8-bit sequence of data except NUL), SP (space), HTAB (horizontal tab), CHAR (any [USASCII] character), VCHAR (any visible [USASCII] character), and WSP (whitespace).

The OWS (optional whitespace) and BWS (bad whitespace) rules are defined in Section 3.2.3 of [RFC7230].

2.3. Terminology

The terms "user agent", "client", "server", "proxy", and "origin server" have the same meaning as in the HTTP/1.1 specification ([RFC7230], Section 2).

The request-host is the name of the host, as known by the user agent, to which the user agent is sending an HTTP request or from which it is receiving an HTTP response (i.e., the name of the host to which it sent the corresponding HTTP request).

The term request-uri refers to "request-target" as defined in Section 5.3 of [RFC7230].

Two sequences of octets are said to case-insensitively match each other if and only if they are equivalent under the i;ascii-casemap collation defined in [RFC4790].

The term string means a sequence of non-NUL octets.

The terms "active document", "ancestor browsing context", "browsing context", "dedicated worker", "Document", "WorkerGlobalScope", "sandboxed origin browsing context flag", "parent browsing context", "shared worker", "the worker's Documents", "nested browsing context", and "top-level browsing context" are defined in [HTML].

"Service Workers" are defined in the Service Workers specification [SERVICE-WORKERS].

The term "origin", the mechanism of deriving an origin from a URI, and the "the same" matching algorithm for origins are defined in [RFC6454].

"Safe" HTTP methods include "GET", "HEAD", "OPTIONS", and "TRACE", as defined in Section 4.2.1 of [RFC7231].

A domain's "public suffix" is the portion of a domain that is controlled by a public registry, such as "com", "co.uk", and "pvt.k12.wy.us". A domain's "registrable domain" is the domain's public suffix plus the label to its left. That is, for "https://www.site.example", the public suffix is "example", and the registrable domain is "site.example". Whenever possible, user agents SHOULD use an up-to-date public suffix list, such as the one maintained by the Mozilla project at [PSL].

The term "request", as well as a request's "client", "current url", "method", and "target browsing context", are defined in [FETCH].

3. Overview

This section outlines a way for an origin server to send state information to a user agent and for the user agent to return the state information to the origin server.

To store state, the origin server includes a Set-Cookie header in an HTTP response. In subsequent requests, the user agent returns a Cookie request header to the origin server. The Cookie header contains cookies the user agent received in previous Set-Cookie headers. The origin server is free to ignore the Cookie header or use its contents for an application-defined purpose.

Origin servers MAY send a Set-Cookie response header with any response. User agents MAY ignore Set-Cookie headers contained in responses with 100-level status codes but MUST process Set-Cookie headers contained in other responses (including responses with 400- and 500-level status codes). An origin server can include multiple Set-Cookie header fields in a single response. The presence of a Cookie or a Set-Cookie header field does not preclude HTTP caches from storing and reusing a response.

Origin servers SHOULD NOT fold multiple Set-Cookie header fields into a single header field. The usual mechanism for folding HTTP headers fields (i.e., as defined in Section 3.2.2 of [RFC7230]) might change the semantics of the Set-Cookie header field because the %x2C (",") character is used by Set-Cookie in a way that conflicts with such folding.

3.1. Examples

Using the Set-Cookie header, a server can send the user agent a short string in an HTTP response that the user agent will return in future HTTP requests that are within the scope of the cookie. For example, the server can send the user agent a "session identifier" named SID with the value 31d4d96e407aad42. The user agent then returns the session identifier in subsequent requests.

```
== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42
```

The server can alter the default scope of the cookie using the Path and Domain attributes. For example, the server can instruct the user agent to return the cookie to every path and every subdomain of site.example.

```
== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42; Path=/; Domain=site.example

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42
```

As shown in the next example, the server can store multiple cookies at the user agent. For example, the server can store a session identifier as well as the user's preferred language by returning two Set-Cookie header fields. Notice that the server uses the Secure and HttpOnly attributes to provide additional security protections for the more sensitive session identifier (see Section 4.1.2).

```
== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42; Path=/; Secure; HttpOnly
Set-Cookie: lang=en-US; Path=/; Domain=site.example

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42; lang=en-US
```

Notice that the Cookie header above contains two cookies, one named SID and one named lang. If the server wishes the user agent to persist the cookie over multiple "sessions" (e.g., user agent restarts), the server can specify an expiration date in the Expires attribute. Note that the user agent might delete the cookie before the expiration date if the user agent's cookie store exceeds its quota or if the user manually deletes the server's cookie.

```
== Server -> User Agent ==

Set-Cookie: lang=en-US; Expires=Wed, 09 Jun 2021 10:18:14 GMT

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42; lang=en-US
```

Finally, to remove a cookie, the server returns a Set-Cookie header with an expiration date in the past. The server will be successful in removing the cookie only if the Path and the Domain attribute in the Set-Cookie header match the values used when the cookie was created.

```
== Server -> User Agent ==

Set-Cookie: lang=; Expires=Sun, 06 Nov 1994 08:49:37 GMT

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42
```

4. Server Requirements

This section describes the syntax and semantics of a well-behaved profile of the Cookie and Set-Cookie headers.

4.1. Set-Cookie

The Set-Cookie HTTP response header is used to send cookies from the server to the user agent.

4.1.1. Syntax

Informally, the Set-Cookie response header contains the header name "Set-Cookie" followed by a ":" and a cookie. Each cookie begins with a name-value-pair, followed by zero or more attribute-value pairs. Servers SHOULD NOT send Set-Cookie headers that fail to conform to the following grammar:

```
set-cookie-header = "Set-Cookie:" SP BWS set-cookie-string
set-cookie-string = BWS cookie-pair *( BWS ";" OWS cookie-av )
cookie-pair       = cookie-name BWS "=" BWS cookie-value
cookie-name       = 1*cookie-octet
cookie-value      = *cookie-octet / ( DQUOTE *cookie-octet DQUOTE )
cookie-octet      = %x21 / %x23-2B / %x2D-3A / %x3C-5B / %x5D-7E / %x80-FF
                      ; US-ASCII characters excluding CTLs,
                      ; whitespace DQUOTE, comma, semicolon,
                      ; and backslash

cookie-av         = expires-av / max-age-av / domain-av /
                    path-av / secure-av / httponly-av /
                    samesite-av / extension-av
expires-av        = "Expires" BWS "=" BWS sane-cookie-date
sane-cookie-date  =
    <IMF-fixdate, defined in [RFC7231], Section 7.1.1.1>
max-age-av        = "Max-Age" BWS "=" BWS non-zero-digit *DIGIT
                      ; In practice, both expires-av and max-age-av
                      ; are limited to dates representable by the
                      ; user agent.
non-zero-digit    = %x31-39
                      ; digits 1 through 9
domain-av         = "Domain" BWS "=" BWS domain-value
domain-value      = <subdomain>
                      ; defined in [RFC1034], Section 3.5, as
                      ; enhanced by [RFC1123], Section 2.1
path-av           = "Path" BWS "=" BWS path-value
path-value        = *av-octet
secure-av         = "Secure"
httponly-av       = "HttpOnly"
samesite-av       = "SameSite" BWS "=" BWS samesite-value
samesite-value    = "Strict" / "Lax" / "None"
extension-av      = *av-octet
av-octet          = %x20-3A / %x3C-7E
                      ; any CHAR except CTLs or ";"
```

Note that some of the grammatical terms above reference documents that use different grammatical notations than this document (which uses ABNF from [RFC5234]).

The semantics of the cookie-value are not defined by this document.

To maximize compatibility with user agents, servers that wish to store arbitrary data in a cookie-value SHOULD encode that data, for example, using Base64 [RFC4648].

Per the grammar above, the cookie-value MAY be wrapped in DQUOTE characters. Note that in this case, the initial and trailing DQUOTE characters are not stripped. They are part of the cookie-value, and will be included in Cookie headers sent to the server.

The portions of the set-cookie-string produced by the cookie-av term are known as attributes. To maximize compatibility with user agents, servers SHOULD NOT produce two attributes with the same name in the same set-cookie-string. (See Section 5.4 for how user agents handle this case.)

Servers SHOULD NOT include more than one Set-Cookie header field in the same response with the same cookie-name. (See Section 5.3 for how user agents handle this case.)

If a server sends multiple responses containing Set-Cookie headers concurrently to the user agent (e.g., when communicating with the user agent over multiple sockets), these responses create a "race condition" that can lead to unpredictable behavior.

NOTE: Some existing user agents differ in their interpretation of two-digit years. To avoid compatibility issues, servers SHOULD use the rfc1123-date format, which requires a four-digit year.

NOTE: Some user agents store and process dates in cookies as 32-bit UNIX time_t values. Implementation bugs in the libraries supporting time_t processing on some systems might cause such user agents to process dates after the year 2038 incorrectly.

4.1.2. Semantics (Non-Normative)

This section describes simplified semantics of the Set-Cookie header. These semantics are detailed enough to be useful for understanding the most common uses of cookies by servers. The full semantics are described in Section 5.

When the user agent receives a Set-Cookie header, the user agent stores the cookie together with its attributes. Subsequently, when the user agent makes an HTTP request, the user agent includes the applicable, non-expired cookies in the Cookie header.

If the user agent receives a new cookie with the same cookie-name, domain-value, and path-value as a cookie that it has already stored, the existing cookie is evicted and replaced with the new cookie. Notice that servers can delete cookies by sending the user agent a new cookie with an Expires attribute with a value in the past.

Unless the cookie's attributes indicate otherwise, the cookie is returned only to the origin server (and not, for example, to any subdomains), and it expires at the end of the current session (as defined by the user agent). User agents ignore unrecognized cookie attributes (but not the entire cookie).

4.1.2.1. The Expires Attribute

The Expires attribute indicates the maximum lifetime of the cookie, represented as the date and time at which the cookie expires. The user agent is not required to retain the cookie until the specified date has passed. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

4.1.2.2. The Max-Age Attribute

The Max-Age attribute indicates the maximum lifetime of the cookie, represented as the number of seconds until the cookie expires. The user agent is not required to retain the cookie for the specified duration. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

NOTE: Some existing user agents do not support the Max-Age attribute. User agents that do not support the Max-Age attribute ignore the attribute.

If a cookie has both the Max-Age and the Expires attribute, the Max-Age attribute has precedence and controls the expiration date of the cookie. If a cookie has neither the Max-Age nor the Expires attribute, the user agent will retain the cookie until "the current session is over" (as defined by the user agent).

4.1.2.3. The Domain Attribute

The Domain attribute specifies those hosts to which the cookie will be sent. For example, if the value of the Domain attribute is "site.example", the user agent will include the cookie in the Cookie header when making HTTP requests to site.example, www.site.example, and www.corp.site.example. (Note that a leading %x2E ("."), if present, is ignored even though that character is not permitted, but a trailing %x2E ("."), if present, will cause the user agent to ignore the attribute.) If the server omits the Domain attribute, the user agent will return the cookie only to the origin server.

WARNING: Some existing user agents treat an absent Domain attribute as if the Domain attribute were present and contained the current host name. For example, if site.example returns a Set-Cookie header without a Domain attribute, these user agents will erroneously send the cookie to www.site.example as well.

The user agent will reject cookies unless the Domain attribute specifies a scope for the cookie that would include the origin server. For example, the user agent will accept a cookie with a Domain attribute of "site.example" or of "foo.site.example" from foo.site.example, but the user agent will not accept a cookie with a Domain attribute of "bar.site.example" or of "baz.foo.site.example".

NOTE: For security reasons, many user agents are configured to reject Domain attributes that correspond to "public suffixes". For example, some user agents will reject Domain attributes of "com" or "co.uk". (See Section 5.4 for more information.)

4.1.2.4. The Path Attribute

The scope of each cookie is limited to a set of paths, controlled by the Path attribute. If the server omits the Path attribute, the user agent will use the "directory" of the request-uri's path component as the default value. (See Section 5.1.4 for more details.)

The user agent will include the cookie in an HTTP request only if the path portion of the request-uri matches (or is a subdirectory of) the cookie's Path attribute, where the %x2F ("/") character is interpreted as a directory separator.

Although seemingly useful for isolating cookies between different paths within a given host, the Path attribute cannot be relied upon for security (see Section 8).

4.1.2.5. The Secure Attribute

The Secure attribute limits the scope of the cookie to "secure" channels (where "secure" is defined by the user agent). When a cookie has the Secure attribute, the user agent will include the cookie in an HTTP request only if the request is transmitted over a secure channel (typically HTTP over Transport Layer Security (TLS) [RFC2818]).

Although seemingly useful for protecting cookies from active network attackers, the Secure attribute protects only the cookie's confidentiality. An active network attacker can overwrite Secure cookies from an insecure channel, disrupting their integrity (see Section 8.6 for more details).

4.1.2.6. The HttpOnly Attribute

The HttpOnly attribute limits the scope of the cookie to HTTP requests. In particular, the attribute instructs the user agent to omit the cookie when providing access to cookies via "non-HTTP" APIs (such as a web browser API that exposes cookies to scripts).

Note that the HttpOnly attribute is independent of the Secure attribute: a cookie can have both the HttpOnly and the Secure attribute.

4.1.2.7. The SameSite Attribute

The "SameSite" attribute limits the scope of the cookie such that it will only be attached to requests if those requests are same-site, as defined by the algorithm in Section 5.2. For example, requests for "https://site.example/sekrit-image" will attach same-site cookies if and only if initiated from a context whose "site for cookies" is "site.example".

If the "SameSite" attribute's value is "Strict", the cookie will only be sent along with "same-site" requests. If the value is "Lax", the cookie will be sent with same-site requests, and with "cross-site" top-level navigations, as described in Section 5.3.7.1. If the value is "None", the cookie will be sent with same-site and cross-site requests. If the "SameSite" attribute's value is something other than these three known keywords, the attribute's value will be treated as "None".

The "SameSite" attribute affects cookie creation as well as delivery. Cookies which assert "SameSite=Lax" or "SameSite=Strict" cannot be set in responses to cross-site subresource requests, or cross-site nested navigations. They can be set along with any top-level navigation, cross-site or otherwise.

4.1.3. Cookie Name Prefixes

Section 8.5 and Section 8.6 of this document spell out some of the drawbacks of cookies' historical implementation. In particular, it is impossible for a server to have confidence that a given cookie was set with a particular set of attributes. In order to provide such confidence in a backwards-compatible way, two common sets of requirements can be inferred from the first few characters of the cookie's name.

The normative requirements for the prefixes described below are detailed in the storage model algorithm defined in Section 5.4.

4.1.3.1. The "__Secure-" Prefix

If a cookie's name begins with a case-sensitive match for the string "__Secure-", then the cookie will have been set with a "Secure" attribute.

For example, the following "Set-Cookie" header would be rejected by a conformant user agent, as it does not have a "Secure" attribute.

```
Set-Cookie: __Secure-SID=12345; Domain=site.example
```

Whereas the following "Set-Cookie" header would be accepted:

```
Set-Cookie: __Secure-SID=12345; Domain=site.example; Secure
```

4.1.3.2. The "__Host-" Prefix

If a cookie's name begins with a case-sensitive match for the string "__Host-", then the cookie will have been set with a "Secure" attribute, a "Path" attribute with a value of "/", and no "Domain" attribute.

This combination yields a cookie that hews as closely as a cookie can to treating the origin as a security boundary. The lack of a "Domain" attribute ensures that the cookie's "host-only-flag" is true, locking the cookie to a particular host, rather than allowing it to span subdomains. Setting the "Path" to "/" means that the cookie is effective for the entire host, and won't be overridden for specific paths. The "Secure" attribute ensures that the cookie is unaltered by non-secure origins, and won't span protocols.

Ports are the only piece of the origin model that "__Host-" cookies continue to ignore.

For example, the following cookies would always be rejected:

```
Set-Cookie: __Host-SID=12345
Set-Cookie: __Host-SID=12345; Secure
Set-Cookie: __Host-SID=12345; Domain=site.example
Set-Cookie: __Host-SID=12345; Domain=site.example; Path=/
Set-Cookie: __Host-SID=12345; Secure; Domain=site.example; Path=/
```

While the would be accepted if set from a secure origin (e.g. "https://site.example/"), and rejected otherwise:

```
Set-Cookie: __Host-SID=12345; Secure; Path=/
```

4.2. Cookie

4.2.1. Syntax

The user agent sends stored cookies to the origin server in the Cookie header. If the server conforms to the requirements in Section 4.1 (and the user agent conforms to the requirements in Section 5), the user agent will send a Cookie header that conforms to the following grammar:

```
cookie-header = "Cookie:" SP cookie-string
cookie-string = cookie-pair *( ";" SP cookie-pair )
```

4.2.2. Semantics

Each cookie-pair represents a cookie stored by the user agent. The cookie-pair contains the cookie-name and cookie-value the user agent received in the Set-Cookie header.

Notice that the cookie attributes are not returned. In particular, the server cannot determine from the Cookie header alone when a cookie will expire, for which hosts the cookie is valid, for which paths the cookie is valid, or whether the cookie was set with the Secure or HttpOnly attributes.

The semantics of individual cookies in the Cookie header are not defined by this document. Servers are expected to imbue these cookies with application-specific semantics.

Although cookies are serialized linearly in the Cookie header, servers SHOULD NOT rely upon the serialization order. In particular, if the Cookie header contains two cookies with the same name (e.g., that were set with different Path or Domain attributes), servers SHOULD NOT rely upon the order in which these cookies appear in the header.

5. User Agent Requirements

This section specifies the Cookie and Set-Cookie headers in sufficient detail that a user agent implementing these requirements precisely can interoperate with existing servers (even those that do not conform to the well-behaved profile described in Section 4).

A user agent could enforce more restrictions than those specified herein (e.g., for the sake of improved security); however, experiments have shown that such strictness reduces the likelihood that a user agent will be able to interoperate with existing servers.

5.1. Subcomponent Algorithms

This section defines some algorithms used by user agents to process specific subcomponents of the Cookie and Set-Cookie headers.

5.1.1. Dates

The user agent MUST use an algorithm equivalent to the following algorithm to parse a cookie-date. Note that the various boolean flags defined as a part of the algorithm (i.e., found-time, found-day-of-month, found-month, found-year) are initially "not set".

1. Using the grammar below, divide the cookie-date into date-tokens.

```
cookie-date     = *delimiter date-token-list *delimiter
date-token-list = date-token *( 1*delimiter date-token )
date-token      = 1*non-delimiter

delimiter       = %x09 / %x20-2F / %x3B-40 / %x5B-60 / %x7B-7E
non-delimiter   = %x00-08 / %x0A-1F / DIGIT / ":" / ALPHA / %x7F-FF
non-digit       = %x00-2F / %x3A-FF

day-of-month    = 1*2DIGIT [ non-digit *OCTET ]
month           = ( "jan" / "feb" / "mar" / "apr" /
                    "may" / "jun" / "jul" / "aug" /
                    "sep" / "oct" / "nov" / "dec" ) *OCTET
year            = 2*4DIGIT [ non-digit *OCTET ]
time            = hms-time [ non-digit *OCTET ]
hms-time        = time-field ":" time-field ":" time-field
time-field      = 1*2DIGIT
```

2. Process each date-token sequentially in the order the date-tokens appear in the cookie-date:

    1. If the found-time flag is not set and the token matches the time production, set the found-time flag and set the hour-value, minute-value, and second-value to the numbers denoted by the digits in the date-token, respectively. Skip the remaining sub-steps and continue to the next date-token.

    2. If the found-day-of-month flag is not set and the date-token matches the day-of-month production, set the found-day-of-month flag and set the day-of-month-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

    3. If the found-month flag is not set and the date-token matches the month production, set the found-month flag and set the month-value to the month denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

    4. If the found-year flag is not set and the date-token matches the year production, set the found-year flag and set the year-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

3. If the year-value is greater than or equal to 70 and less than or equal to 99, increment the year-value by 1900.

4. If the year-value is greater than or equal to 0 and less than or equal to 69, increment the year-value by 2000.

    1. NOTE: Some existing user agents interpret two-digit years differently.

5. Abort these steps and fail to parse the cookie-date if:

    * at least one of the found-day-of-month, found-month, found-year, or found-time flags is not set,

    * the day-of-month-value is less than 1 or greater than 31,

    * the year-value is less than 1601,

    * the hour-value is greater than 23,

    * the minute-value is greater than 59, or

    * the second-value is greater than 59.

    (Note that leap seconds cannot be represented in this syntax.)

6. Let the parsed-cookie-date be the date whose day-of-month, month, year, hour, minute, and second (in UTC) are the day-of-month-value, the month-value, the year-value, the hour-value, the minute-value, and the second-value, respectively. If no such date exists, abort these steps and fail to parse the cookie-date.

7. Return the parsed-cookie-date as the result of this algorithm.

5.1.2. Canonicalized Host Names

A canonicalized host name is the string generated by the following algorithm:

1. Convert the host name to a sequence of individual domain name labels.

2. Convert each label that is not a Non-Reserved LDH (NR-LDH) label, to an A-label (see Section 2.3.2.1 of [RFC5890] for the former and latter), or to a "punycode label" (a label resulting from the "ToASCII" conversion in Section 4 of [RFC3490]), as appropriate (see Section 6.3 of this specification).

3. Concatenate the resulting labels, separated by a %x2E (".") character.

5.1.3. Domain Matching

A string domain-matches a given domain string if at least one of the following conditions hold:

- The domain string and the string are identical. (Note that both the domain string and the string will have been canonicalized to lower case at this point.)

- All of the following conditions hold:

  * The domain string is a suffix of the string.

  * The last character of the string that is not included in the domain string is a %x2E (".") character.

  * The string is a host name (i.e., not an IP address).

5.1.4. Paths and Path-Match

The user agent MUST use an algorithm equivalent to the following algorithm to compute the default-path of a cookie:

1. Let uri-path be the path portion of the request-uri if such a portion exists (and empty otherwise). For example, if the request-uri contains just a path (and optional query string), then the uri-path is that path (without the %x3F ("?") character or query string), and if the request-uri contains a full absoluteURI, the uri-path is the path component of that URI.

2. If the uri-path is empty or if the first character of the uri-path is not a %x2F ("/") character, output %x2F ("/") and skip the remaining steps.

3. If the uri-path contains no more than one %x2F ("/") character, output %x2F ("/") and skip the remaining step.

4. Output the characters of the uri-path from the first character up to, but not including, the right-most %x2F ("/").

A request-path path-matches a given cookie-path if at least one of the following conditions holds:

- The cookie-path and the request-path are identical.

  Note that this differs from the rules in [RFC3986] for equivalence of the path component, and hence two equivalent paths can have different cookies.

- The cookie-path is a prefix of the request-path, and the last character of the cookie-path is %x2F ("/").

- The cookie-path is a prefix of the request-path, and the first character of the request-path that is not included in the cookie-path is a %x2F ("/") character.

5.2. "Same-site" and "cross-site" Requests

A request is "same-site" if its target's URI's origin's registrable domain is an exact match for the request's client's "site for cookies", or if the request has no client. The request is otherwise "cross-site".

For a given request ("request"), the following algorithm returns "same-site" or "cross-site":

1. If "request"'s client is "null", return "same-site".

   Note that this is the case for navigation triggered by the user directly (e.g. by typing directly into a user agent's address bar).

2. Let "site" be "request"'s client's "site for cookies" (as defined in the following sections).

3. Let "target" be the registrable domain of "request"'s current url.

4. If "site" is an exact match for "target", return "same-site".

5. Return "cross-site".

The request's client's "site for cookies" is calculated depending upon its client's type, as described in the following subsections:

5.2.1. Document-based requests

The URI displayed in a user agent's address bar is the only security context directly exposed to users, and therefore the only signal users can reasonably rely upon to determine whether or not they trust a particular website. The registrable domain of that URI's origin represents the context in which a user most likely believes themselves to be interacting. We'll label this domain the "top-level site".

For a document displayed in a top-level browsing context, we can stop here: the document's "site for cookies" is the top-level site.

For documents which are displayed in nested browsing contexts, we need to audit the origins of each of a document's ancestor browsing contexts' active documents in order to account for the "multiple-nested scenarios" described in Section 4 of [RFC7034]. A document's "site for cookies" is the top-level site if and only if the document and each of its ancestor documents' origins have the same registrable domain as the top-level site. Otherwise its "site for cookies" is the empty string.

Given a Document ("document"), the following algorithm returns its "site for cookies" (either a registrable domain, or the empty string):

1. Let "top-document" be the active document in "document"'s browsing context's top-level browsing context.

2. Let "top-origin" be the origin of "top-document"'s URI if "top-document"'s sandboxed origin browsing context flag is set, and "top-document"'s origin otherwise.

3. Let "documents" be a list containing "document" and each of "document"'s ancestor browsing contexts' active documents.

4. For each "item" in "documents":

    1. Let "origin" be the origin of "item"'s URI if "item"'s sandboxed origin browsing context flag is set, and "item"'s origin otherwise.

   2. If "origin"'s host's registrable domain is not an exact match for "top-origin"'s host's registrable domain, return the empty string.

5. Return "top-origin"'s host's registrable domain.

5.2.2. Worker-based requests

Worker-driven requests aren't as clear-cut as document-driven requests, as there isn't a clear link between a top-level browsing context and a worker. This is especially true for Service Workers [SERVICE-WORKERS], which may execute code in the background, without any document visible at all.

Note: The descriptions below assume that workers must be same-origin with the documents that instantiate them. If this invariant changes, we'll need to take the worker's script's URI into account when determining their status.

5.2.2.1. Dedicated and Shared Workers

Dedicated workers are simple, as each dedicated worker is bound to one and only one document. Requests generated from a dedicated worker (via "importScripts", "XMLHttpRequest", "fetch()", etc) define their "site for cookies" as that document's "site for cookies".

Shared workers may be bound to multiple documents at once. As it is quite possible for those documents to have distinct "site for cookie" values, the worker's "site for cookies" will be the empty string in cases where the values diverge, and the shared value in cases where the values agree.

Given a WorkerGlobalScope ("worker"), the following algorithm returns its "site for cookies" (either a registrable domain, or the empty string):

1. Let "site" be "worker"'s origin's host's registrable domain.

2. For each "document" in "worker"'s Documents:

   1. Let "document-site" be "document"'s "site for cookies" (as defined in Section 5.2.1).

   2. If "document-site" is not an exact match for "site", return the empty string.

3. Return "site".

5.2.2.2. Service Workers

Service Workers are more complicated, as they act as a completely separate execution context with only tangential relationship to the Document which registered them.

Requests which simply pass through a Service Worker will be handled as described above: the request's client will be the Document or Worker which initiated the request, and its "site for cookies" will be those defined in Section 5.2.1 and Section 5.2.2.1

Requests which are initiated by the Service Worker itself (via a direct call to "fetch()", for instance), on the other hand, will have a client which is a ServiceWorkerGlobalScope. Its "site for cookies" will be the registrable domain of the Service Worker's URI.

Given a ServiceWorkerGlobalScope ("worker"), the following algorithm returns its "site for cookies" (either a registrable domain, or the empty string):

1. Return "worker"'s origin's host's registrable domain.

5.3. The Set-Cookie Header

When a user agent receives a Set-Cookie header field in an HTTP response, the user agent MAY ignore the Set-Cookie header field in its entirety. For example, the user agent might wish to block responses to "third-party" requests from setting cookies (see Section 7.1).

If the user agent does not ignore the Set-Cookie header field in its entirety, the user agent MUST parse the field-value of the Set-Cookie header field as a set-cookie-string (defined below).

NOTE: The algorithm below is more permissive than the grammar in Section 4.1. For example, the algorithm strips leading and trailing whitespace from the cookie name and value (but maintains internal whitespace), whereas the grammar in Section 4.1 forbids whitespace in these positions. User agents use this algorithm so as to interoperate with servers that do not follow the recommendations in Section 4.

A user agent MUST use an algorithm equivalent to the following algorithm to parse a set-cookie-string:

1. If the set-cookie-string contains a %x3B (";") character:

1. The name-value-pair string consists of the characters up to, but not including, the first %x3B (";"), and the unparsed-attributes consist of the remainder of the set-cookie-string (including the %x3B (";") in question).

Otherwise:

1. The name-value-pair string consists of all the characters contained in the set-cookie-string, and the unparsed-attributes is the empty string.

2. If the name-value-pair string lacks a %x3D ("=") character, then the name string is empty, and the value string is the value of name-value-pair.

   Otherwise, the name string consists of the characters up to, but not including, the first %x3D ("=") character, and the (possibly empty) value string consists of the characters after the first %x3D ("=") character.

3. Remove any leading or trailing WSP characters from the name string and the value string.

4. If both the name string and the value string are empty, ignore the set-cookie-string entirely.

5. The cookie-name is the name string, and the cookie-value is the value string.

The user agent MUST use an algorithm equivalent to the following algorithm to parse the unparsed-attributes:

1. If the unparsed-attributes string is empty, skip the rest of these steps.

2. Discard the first character of the unparsed-attributes (which will be a %x3B (";") character).

3. If the remaining unparsed-attributes contains a %x3B (";") character:

   1. Consume the characters of the unparsed-attributes up to, but not including, the first %x3B (";") character.

   Otherwise:

   1. Consume the remainder of the unparsed-attributes.

Let the cookie-av string be the characters consumed in this step.

4. If the cookie-av string contains a %x3D ("=") character:

    1. The (possibly empty) attribute-name string consists of the characters up to, but not including, the first %x3D ("=") character, and the (possibly empty) attribute-value string consists of the characters after the first %x3D ("=") character.

    Otherwise:

    1. The attribute-name string consists of the entire cookie-av string, and the attribute-value string is empty.

5. Remove any leading or trailing WSP characters from the attribute-name string and the attribute-value string.

6. Process the attribute-name and attribute-value according to the requirements in the following subsections. (Notice that attributes with unrecognized attribute-names are ignored.)

7. Return to Step 1 of this algorithm.

When the user agent finishes parsing the set-cookie-string, the user agent is said to "receive a cookie" from the request-uri with name cookie-name, value cookie-value, and attributes cookie-attribute-list. (See Section 5.4 for additional requirements triggered by receiving a cookie.)

5.3.1. The Expires Attribute

If the attribute-name case-insensitively matches the string "Expires", the user agent MUST process the cookie-av as follows.

1. Let the expiry-time be the result of parsing the attribute-value as cookie-date (see Section 5.1.1).

2. If the attribute-value failed to parse as a cookie date, ignore the cookie-av.

3. If the expiry-time is later than the last date the user agent can represent, the user agent MAY replace the expiry-time with the last representable date.

4. If the expiry-time is earlier than the earliest date the user agent can represent, the user agent MAY replace the expiry-time with the earliest representable date.

5. Append an attribute to the cookie-attribute-list with an attribute-name of Expires and an attribute-value of expiry-time.

5.3.2. The Max-Age Attribute

If the attribute-name case-insensitively matches the string "Max-Age", the user agent MUST process the cookie-av as follows.

1. If the first character of the attribute-value is not a DIGIT or a "-" character, ignore the cookie-av.

2. If the remainder of attribute-value contains a non-DIGIT character, ignore the cookie-av.

3. Let delta-seconds be the attribute-value converted to an integer.

4. If delta-seconds is less than or equal to zero (0), let expiry-time be the earliest representable date and time. Otherwise, let the expiry-time be the current date and time plus delta-seconds seconds.

5. Append an attribute to the cookie-attribute-list with an attribute-name of Max-Age and an attribute-value of expiry-time.

5.3.3. The Domain Attribute

If the attribute-name case-insensitively matches the string "Domain", the user agent MUST process the cookie-av as follows.

1. If the attribute-value is empty, the behavior is undefined. However, the user agent SHOULD ignore the cookie-av entirely.

2. If the first character of the attribute-value string is %x2E ("."):

    1. Let cookie-domain be the attribute-value without the leading %x2E (".") character.

    Otherwise:

    1. Let cookie-domain be the entire attribute-value.

3. Convert the cookie-domain to lower case.

4. Append an attribute to the cookie-attribute-list with an attribute-name of Domain and an attribute-value of cookie-domain.

5.3.4. The Path Attribute

If the attribute-name case-insensitively matches the string "Path", the user agent MUST process the cookie-av as follows.

1. If the attribute-value is empty or if the first character of the attribute-value is not %x2F ("/"):

    1. Let cookie-path be the default-path.

    Otherwise:

    1. Let cookie-path be the attribute-value.

2. Append an attribute to the cookie-attribute-list with an attribute-name of Path and an attribute-value of cookie-path.

5.3.5. The Secure Attribute

If the attribute-name case-insensitively matches the string "Secure", the user agent MUST append an attribute to the cookie-attribute-list with an attribute-name of Secure and an empty attribute-value.

5.3.6. The HttpOnly Attribute

If the attribute-name case-insensitively matches the string "HttpOnly", the user agent MUST append an attribute to the cookie-attribute-list with an attribute-name of HttpOnly and an empty attribute-value.

5.3.7. The SameSite Attribute

If the attribute-name case-insensitively matches the string "SameSite", the user agent MUST process the cookie-av as follows:

1. Let "enforcement" be "None".

2. If cookie-av's attribute-value is a case-insensitive match for "Strict", set "enforcement" to "Strict".

3. If cookie-av's attribute-value is a case-insensitive match for "Lax", set "enforcement" to "Lax".

4. Append an attribute to the cookie-attribute-list with an attribute-name of "SameSite" and an attribute-value of "enforcement".

Note: This algorithm maps the "None" value, as well as any unknown value, to the "None" behavior, which is helpful for backwards compatibility when introducing new variants.

5.3.7.1. "Strict" and "Lax" enforcement

Same-site cookies in "Strict" enforcement mode will not be sent along with top-level navigations which are triggered from a cross-site document context. As discussed in Section 8.8.2, this might or might not be compatible with existing session management systems. In the interests of providing a drop-in mechanism that mitigates the risk of CSRF attacks, developers may set the "SameSite" attribute in a "Lax" enforcement mode that carves out an exception which sends same-site cookies along with cross-site requests if and only if they are top-level navigations which use a "safe" (in the [RFC7231] sense) HTTP method.

Lax enforcement provides reasonable defense in depth against CSRF attacks that rely on unsafe HTTP methods (like "POST"), but does not offer a robust defense against CSRF as a general category of attack:

1. Attackers can still pop up new windows or trigger top-level navigations in order to create a "same-site" request (as described in Section 5.2.1), which is only a speedbump along the road to exploitation.

2. Features like "<link rel='prerender'>" [prerendering] can be exploited to create "same-site" requests without the risk of user detection.

When possible, developers should use a session management mechanism such as that described in Section 8.8.2 to mitigate the risk of CSRF more completely.

5.4. Storage Model

The user agent stores the following fields about each cookie: name, value, expiry-time, domain, path, creation-time, last-access-time, persistent-flag, host-only-flag, secure-only-flag, http-only-flag, and same-site-flag.

When the user agent "receives a cookie" from a request-uri with name cookie-name, value cookie-value, and attributes cookie-attribute-list, the user agent MUST process the cookie as follows:

1. A user agent MAY ignore a received cookie in its entirety. For example, the user agent might wish to block receiving cookies

from "third-party" responses or the user agent might not wish to store cookies that exceed some size.

2. Create a new cookie with name cookie-name, value cookie-value. Set the creation-time and the last-access-time to the current date and time.

3. If the cookie-attribute-list contains an attribute with an attribute-name of "Max-Age":

    1. Set the cookie's persistent-flag to true.

    2. Set the cookie's expiry-time to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Max-Age".

    Otherwise, if the cookie-attribute-list contains an attribute with an attribute-name of "Expires" (and does not contain an attribute with an attribute-name of "Max-Age"):

    1. Set the cookie's persistent-flag to true.

    2. Set the cookie's expiry-time to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Expires".

    Otherwise:

    1. Set the cookie's persistent-flag to false.

    2. Set the cookie's expiry-time to the latest representable date.

4. If the cookie-attribute-list contains an attribute with an attribute-name of "Domain":

    1. Let the domain-attribute be the attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Domain".

    Otherwise:

    1. Let the domain-attribute be the empty string.

5. If the user agent is configured to reject "public suffixes" and the domain-attribute is a public suffix:

1. If the domain-attribute is identical to the canonicalized request-host:

    1. Let the domain-attribute be the empty string.

    Otherwise:

    1. Ignore the cookie entirely and abort these steps.

NOTE: This step prevents "attacker.example" from disrupting the integrity of "site.example" by setting a cookie with a Domain attribute of "example".

6. If the domain-attribute is non-empty:

    1. If the canonicalized request-host does not domain-match the domain-attribute:

        1. Ignore the cookie entirely and abort these steps.

        Otherwise:

        1. Set the cookie's host-only-flag to false.

        2. Set the cookie's domain to the domain-attribute.

    Otherwise:

    1. Set the cookie's host-only-flag to true.

    2. Set the cookie's domain to the canonicalized request-host.

7. If the cookie-attribute-list contains an attribute with an attribute-name of "Path", set the cookie's path to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Path". Otherwise, set the cookie's path to the default-path of the request-uri.

8. If the cookie-attribute-list contains an attribute with an attribute-name of "Secure", set the cookie's secure-only-flag to true. Otherwise, set the cookie's secure-only-flag to false.

9. If the scheme component of the request-uri does not denote a "secure" protocol (as defined by the user agent), and the cookie's secure-only-flag is true, then abort these steps and ignore the cookie entirely.

10. If the cookie-attribute-list contains an attribute with an attribute-name of "HttpOnly", set the cookie's http-only-flag to true. Otherwise, set the cookie's http-only-flag to false.

11. If the cookie was received from a "non-HTTP" API and the cookie's http-only-flag is true, abort these steps and ignore the cookie entirely.

12. If the cookie's secure-only-flag is false, and the scheme component of request-uri does not denote a "secure" protocol, then abort these steps and ignore the cookie entirely if the cookie store contains one or more cookies that meet all of the following criteria:

    1. Their name matches the name of the newly-created cookie.

    2. Their secure-only-flag is true.

    3. Their domain domain-matches the domain of the newly-created cookie, or vice-versa.

    4. The path of the newly-created cookie path-matches the path of the existing cookie.

    Note: The path comparison is not symmetric, ensuring only that a newly-created, non-secure cookie does not overlay an existing secure cookie, providing some mitigation against cookie-fixing attacks. That is, given an existing secure cookie named 'a' with a path of '/login', a non-secure cookie named 'a' could be set for a path of '/' or '/foo', but not for a path of '/login' or '/login/en'.

13. If the cookie-attribute-list contains an attribute with an attribute-name of "SameSite", set the cookie's same-site-flag to the attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "SameSite" (i.e. either "Strict", "Lax", or "None"). Otherwise, set the cookie's same-site-flag to "None".

14. If the cookie's "same-site-flag" is not "None":

    1. If the cookie was received from a "non-HTTP" API, and the API was called from a context whose "site for cookies" is not an exact match for request-uri's host's registrable domain, then abort these steps and ignore the newly created cookie entirely.

2. If the cookie was received from a "same-site" request (as defined in Section 5.2), skip the remaining substeps and continue processing the cookie.

3. If the cookie was received from a request which is navigating a top-level browsing context [HTML] (e.g. if the request's "reserved client" is either "null" or an environment whose "target browsing context" is a top-level browing context), skip the remaining substeps and continue processing the cookie.

   Note: Top-level navigations can create a cookie with any "SameSite" value, even if the new cookie wouldn't have been sent along with the request had it already existed prior to the navigation.

4. Abort these steps and ignore the newly created cookie entirely.

15. If the cookie-name begins with a case-sensitive match for the string "__Secure-", abort these steps and ignore the cookie entirely unless the cookie's secure-only-flag is true.

16. If the cookie-name begins with a case-sensitive match for the string "__Host-", abort these steps and ignore the cookie entirely unless the cookie meets all the following criteria:

    1. The cookie's secure-only-flag is true.

    2. The cookie's host-only-flag is true.

    3. The cookie-attribute-list contains an attribute with an attribute-name of "Path", and the cookie's path is "/".

17. If the cookie store contains a cookie with the same name, domain, host-only-flag, and path as the newly-created cookie:

    1. Let old-cookie be the existing cookie with the same name, domain, host-only-flag, and path as the newly-created cookie. (Notice that this algorithm maintains the invariant that there is at most one such cookie.)

    2. If the newly-created cookie was received from a "non-HTTP" API and the old-cookie's http-only-flag is true, abort these steps and ignore the newly created cookie entirely.

    3. Update the creation-time of the newly-created cookie to match the creation-time of the old-cookie.

   4. Remove the old-cookie from the cookie store.

18. Insert the newly-created cookie into the cookie store.

A cookie is "expired" if the cookie has an expiry date in the past.

The user agent MUST evict all expired cookies from the cookie store if, at any time, an expired cookie exists in the cookie store.

At any time, the user agent MAY "remove excess cookies" from the cookie store if the number of cookies sharing a domain field exceeds some implementation-defined upper bound (such as 50 cookies).

At any time, the user agent MAY "remove excess cookies" from the cookie store if the cookie store exceeds some predetermined upper bound (such as 3000 cookies).

When the user agent removes excess cookies from the cookie store, the user agent MUST evict cookies in the following priority order:

1. Expired cookies.

2. Cookies whose secure-only-flag is false, and which share a domain field with more than a predetermined number of other cookies.

3. Cookies that share a domain field with more than a predetermined number of other cookies.

4. All cookies.

If two cookies have the same removal priority, the user agent MUST evict the cookie with the earliest last-access-time first.

When "the current session is over" (as defined by the user agent), the user agent MUST remove from the cookie store all cookies with the persistent-flag set to false.

5.5. The Cookie Header

The user agent includes stored cookies in the Cookie HTTP request header.

When the user agent generates an HTTP request, the user agent MUST NOT attach more than one Cookie header field.

A user agent MAY omit the Cookie header in its entirety. For example, the user agent might wish to block sending cookies during "third-party" requests from setting cookies (see Section 7.1).

If the user agent does attach a Cookie header field to an HTTP request, the user agent MUST send the cookie-string (defined below) as the value of the header field.

The user agent MUST use an algorithm equivalent to the following algorithm to compute the cookie-string from a cookie store and a request-uri:

1. Let cookie-list be the set of cookies from the cookie store that meets all of the following requirements:

   * Either:

     + The cookie's host-only-flag is true and the canonicalized request-host is identical to the cookie's domain.

     Or:

     + The cookie's host-only-flag is false and the canonicalized request-host domain-matches the cookie's domain.

   * The request-uri's path path-matches the cookie's path.

   * If the cookie's secure-only-flag is true, then the request-uri's scheme must denote a "secure" protocol (as defined by the user agent).

     NOTE: The notion of a "secure" protocol is not defined by this document. Typically, user agents consider a protocol secure if the protocol makes use of transport-layer security, such as SSL or TLS. For example, most user agents consider "https" to be a scheme that denotes a secure protocol.

   * If the cookie's http-only-flag is true, then exclude the cookie if the cookie-string is being generated for a "non-HTTP" API (as defined by the user agent).

   * If the cookie's same-site-flag is not "None", and the HTTP request is cross-site (as defined in Section 5.2) then exclude the cookie unless all of the following statements hold:

     1. The same-site-flag is "Lax"

     2. The HTTP request's method is "safe".

     3. The HTTP request's target browsing context is a top-level browsing context.

2. The user agent SHOULD sort the cookie-list in the following order:

   * Cookies with longer paths are listed before cookies with shorter paths.

   * Among cookies that have equal-length path fields, cookies with earlier creation-times are listed before cookies with later creation-times.

   NOTE: Not all user agents sort the cookie-list in this order, but this order reflects common practice when this document was written, and, historically, there have been servers that (erroneously) depended on this order.

3. Update the last-access-time of each cookie in the cookie-list to the current date and time.

4. Serialize the cookie-list into a cookie-string by processing each cookie in the cookie-list in order:

   1. If the cookies' name is not empty, output the cookie's name followed by the %x3D ("=") character.

   2. If the cookies' value is not empty, output the cookie's value.

   3. If there is an unprocessed cookie in the cookie-list, output the characters %x3B and %x20 ("; ").

NOTE: Despite its name, the cookie-string is actually a sequence of octets, not a sequence of characters. To convert the cookie-string (or components thereof) into a sequence of characters (e.g., for presentation to the user), the user agent might wish to try using the UTF-8 character encoding [RFC3629] to decode the octet sequence. This decoding might fail, however, because not every sequence of octets is valid UTF-8.

6. Implementation Considerations

6.1. Limits

Practical user agent implementations have limits on the number and size of cookies that they can store. General-use user agents SHOULD provide each of the following minimum capabilities:

- o At least 4096 bytes per cookie (as measured by the sum of the length of the cookie's name, value, and attributes).

- At least 50 cookies per domain.
- At least 3000 cookies total.

Servers SHOULD use as few and as small cookies as possible to avoid reaching these implementation limits and to minimize network bandwidth due to the Cookie header being included in every request.

Servers SHOULD gracefully degrade if the user agent fails to return one or more cookies in the Cookie header because the user agent might evict any cookie at any time on orders from the user.

6.2. Application Programming Interfaces

One reason the Cookie and Set-Cookie headers use such esoteric syntax is that many platforms (both in servers and user agents) provide a string-based application programming interface (API) to cookies, requiring application-layer programmers to generate and parse the syntax used by the Cookie and Set-Cookie headers, which many programmers have done incorrectly, resulting in interoperability problems.

Instead of providing string-based APIs to cookies, platforms would be well-served by providing more semantic APIs. It is beyond the scope of this document to recommend specific API designs, but there are clear benefits to accepting an abstract "Date" object instead of a serialized date string.

6.3. IDNA Dependency and Migration

IDNA2008 [RFC5890] supersedes IDNA2003 [RFC3490]. However, there are differences between the two specifications, and thus there can be differences in processing (e.g., converting) domain name labels that have been registered under one from those registered under the other. There will be a transition period of some time during which IDNA2003-based domain name labels will exist in the wild. User agents SHOULD implement IDNA2008 [RFC5890] and MAY implement [UTS46] or [RFC5895] in order to facilitate their IDNA transition. If a user agent does not implement IDNA2008, the user agent MUST implement IDNA2003 [RFC3490].

7. Privacy Considerations

Cookies are often criticized for letting servers track users. For example, a number of "web analytics" companies use cookies to recognize when a user returns to a web site or visits another web site. Although cookies are not the only mechanism servers can use to track users across HTTP requests, cookies facilitate tracking because they are persistent across user agent sessions and can be shared between hosts.

7.1. Third-Party Cookies

Particularly worrisome are so-called "third-party" cookies. In rendering an HTML document, a user agent often requests resources from other servers (such as advertising networks). These third-party servers can use cookies to track the user even if the user never visits the server directly. For example, if a user visits a site that contains content from a third party and then later visits another site that contains content from the same third party, the third party can track the user between the two sites.

Given this risk to user privacy, some user agents restrict how third-party cookies behave, and those restrictions vary widly. For instance, user agents might block third-party cookies entirely by refusing to send Cookie headers or process Set-Cookie headers during third-party requests. They might take a less draconian approach by partitioning cookies based on the first-party context, sending one set of cookies to a given third party in one first-party context, and another to the same third party in another.

This document grants user agents wide latitude to experiment with third-party cookie policies that balance the privacy and compatibility needs of their users. However, this document does not endorse any particular third-party cookie policy.

Third-party cookie blocking policies are often ineffective at achieving their privacy goals if servers attempt to work around their restrictions to track users. In particular, two collaborating servers can often track users without using cookies at all by injecting identifying information into dynamic URLs.

7.2. User Controls

User agents SHOULD provide users with a mechanism for managing the cookies stored in the cookie store. For example, a user agent might let users delete all cookies received during a specified time period or all the cookies related to a particular domain. In addition, many user agents include a user interface element that lets users examine the cookies stored in their cookie store.

User agents SHOULD provide users with a mechanism for disabling cookies. When cookies are disabled, the user agent MUST NOT include a Cookie header in outbound HTTP requests and the user agent MUST NOT process Set-Cookie headers in inbound HTTP responses.

Some user agents provide users the option of preventing persistent storage of cookies across sessions. When configured thusly, user agents MUST treat all received cookies as if the persistent-flag were set to false. Some popular user agents expose this functionality via "private browsing" mode [Aggarwal2010].

Some user agents provide users with the ability to approve individual writes to the cookie store. In many common usage scenarios, these controls generate a large number of prompts. However, some privacy-conscious users find these controls useful nonetheless.

7.3. Expiration Dates

Although servers can set the expiration date for cookies to the distant future, most user agents do not actually retain cookies for multiple decades. Rather than choosing gratuitously long expiration periods, servers SHOULD promote user privacy by selecting reasonable cookie expiration periods based on the purpose of the cookie. For example, a typical session identifier might reasonably be set to expire in two weeks.

8. Security Considerations

8.1. Overview

Cookies have a number of security pitfalls. This section overviews a few of the more salient issues.

In particular, cookies encourage developers to rely on ambient authority for authentication, often becoming vulnerable to attacks such as cross-site request forgery [CSRF]. Also, when storing session identifiers in cookies, developers often create session fixation vulnerabilities.

Transport-layer encryption, such as that employed in HTTPS, is insufficient to prevent a network attacker from obtaining or altering a victim's cookies because the cookie protocol itself has various vulnerabilities (see "Weak Confidentiality" and "Weak Integrity", below). In addition, by default, cookies do not provide confidentiality or integrity from network attackers, even when used in conjunction with HTTPS.

8.2. Ambient Authority

A server that uses cookies to authenticate users can suffer security vulnerabilities because some user agents let remote parties issue HTTP requests from the user agent (e.g., via HTTP redirects or HTML forms). When issuing those requests, user agents attach cookies even if the remote party does not know the contents of the cookies, potentially letting the remote party exercise authority at an unwary server.

Although this security concern goes by a number of names (e.g., cross-site request forgery, confused deputy), the issue stems from cookies being a form of ambient authority. Cookies encourage server operators to separate designation (in the form of URLs) from authorization (in the form of cookies). Consequently, the user agent might supply the authorization for a resource designated by the attacker, possibly causing the server or its clients to undertake actions designated by the attacker as though they were authorized by the user.

Instead of using cookies for authorization, server operators might wish to consider entangling designation and authorization by treating URLs as capabilities. Instead of storing secrets in cookies, this approach stores secrets in URLs, requiring the remote entity to supply the secret itself. Although this approach is not a panacea, judicious application of these principles can lead to more robust security.

8.3. Clear Text

Unless sent over a secure channel (such as TLS), the information in the Cookie and Set-Cookie headers is transmitted in the clear.

1. All sensitive information conveyed in these headers is exposed to an eavesdropper.

2. A malicious intermediary could alter the headers as they travel in either direction, with unpredictable results.

3. A malicious client could alter the Cookie header before transmission, with unpredictable results.

Servers SHOULD encrypt and sign the contents of cookies (using whatever format the server desires) when transmitting them to the user agent (even when sending the cookies over a secure channel). However, encrypting and signing cookie contents does not prevent an attacker from transplanting a cookie from one user agent to another or from replaying the cookie at a later time.

In addition to encrypting and signing the contents of every cookie, servers that require a higher level of security SHOULD use the Cookie and Set-Cookie headers only over a secure channel. When using cookies over a secure channel, servers SHOULD set the Secure attribute (see Section 4.1.2.5) for every cookie. If a server does not set the Secure attribute, the protection provided by the secure channel will be largely moot.

For example, consider a webmail server that stores a session identifier in a cookie and is typically accessed over HTTPS. If the server does not set the Secure attribute on its cookies, an active network attacker can intercept any outbound HTTP request from the user agent and redirect that request to the webmail server over HTTP. Even if the webmail server is not listening for HTTP connections, the user agent will still include cookies in the request. The active network attacker can intercept these cookies, replay them against the server, and learn the contents of the user's email. If, instead, the server had set the Secure attribute on its cookies, the user agent would not have included the cookies in the clear-text request.

8.4. Session Identifiers

Instead of storing session information directly in a cookie (where it might be exposed to or replayed by an attacker), servers commonly store a nonce (or "session identifier") in a cookie. When the server receives an HTTP request with a nonce, the server can look up state information associated with the cookie using the nonce as a key.

Using session identifier cookies limits the damage an attacker can cause if the attacker learns the contents of a cookie because the nonce is useful only for interacting with the server (unlike non-nonce cookie content, which might itself be sensitive). Furthermore, using a single nonce prevents an attacker from "splicing" together cookie content from two interactions with the server, which could cause the server to behave unexpectedly.

Using session identifiers is not without risk. For example, the server SHOULD take care to avoid "session fixation" vulnerabilities. A session fixation attack proceeds in three steps. First, the attacker transplants a session identifier from his or her user agent to the victim's user agent. Second, the victim uses that session identifier to interact with the server, possibly imbuing the session identifier with the user's credentials or confidential information. Third, the attacker uses the session identifier to interact with server directly, possibly obtaining the user's authority or confidential information.

8.5. Weak Confidentiality

Cookies do not provide isolation by port. If a cookie is readable by a service running on one port, the cookie is also readable by a service running on another port of the same server. If a cookie is writable by a service on one port, the cookie is also writable by a service running on another port of the same server. For this reason, servers SHOULD NOT both run mutually distrusting services on different ports of the same host and use cookies to store security-sensitive information.

Cookies do not provide isolation by scheme. Although most commonly used with the http and https schemes, the cookies for a given host might also be available to other schemes, such as ftp and gopher. Although this lack of isolation by scheme is most apparent in non-HTTP APIs that permit access to cookies (e.g., HTML's document.cookie API), the lack of isolation by scheme is actually present in requirements for processing cookies themselves (e.g., consider retrieving a URI with the gopher scheme via HTTP).

Cookies do not always provide isolation by path. Although the network-level protocol does not send cookies stored for one path to another, some user agents expose cookies via non-HTTP APIs, such as HTML's document.cookie API. Because some of these user agents (e.g., web browsers) do not isolate resources received from different paths, a resource retrieved from one path might be able to access cookies stored for another path.

8.6. Weak Integrity

Cookies do not provide integrity guarantees for sibling domains (and their subdomains). For example, consider foo.site.example and bar.site.example. The foo.site.example server can set a cookie with a Domain attribute of "site.example" (possibly overwriting an existing "site.example" cookie set by bar.site.example), and the user agent will include that cookie in HTTP requests to bar.site.example. In the worst case, bar.site.example will be unable to distinguish this cookie from a cookie it set itself. The foo.site.example server might be able to leverage this ability to mount an attack against bar.site.example.

Even though the Set-Cookie header supports the Path attribute, the Path attribute does not provide any integrity protection because the user agent will accept an arbitrary Path attribute in a Set-Cookie header. For example, an HTTP response to a request for http://site.example/foo/bar can set a cookie with a Path attribute of "/qux". Consequently, servers SHOULD NOT both run mutually distrusting services on different paths of the same host and use cookies to store security-sensitive information.

An active network attacker can also inject cookies into the Cookie header sent to https://site.example/ by impersonating a response from http://site.example/ and injecting a Set-Cookie header. The HTTPS server at site.example will be unable to distinguish these cookies from cookies that it set itself in an HTTPS response. An active network attacker might be able to leverage this ability to mount an attack against site.example even if site.example uses HTTPS exclusively.

Servers can partially mitigate these attacks by encrypting and signing the contents of their cookies. However, using cryptography does not mitigate the issue completely because an attacker can replay a cookie he or she received from the authentic site.example server in the user's session, with unpredictable results.

Finally, an attacker might be able to force the user agent to delete cookies by storing a large number of cookies. Once the user agent reaches its storage limit, the user agent will be forced to evict some cookies. Servers SHOULD NOT rely upon user agents retaining cookies.

8.7. Reliance on DNS

Cookies rely upon the Domain Name System (DNS) for security. If the DNS is partially or fully compromised, the cookie protocol might fail to provide the security properties required by applications.

8.8. SameSite Cookies

8.8.1. Defense in depth

"SameSite" cookies offer a robust defense against CSRF attack when deployed in strict mode, and when supported by the client. It is, however, prudent to ensure that this designation is not the extent of a site's defense against CSRF, as same-site navigations and submissions can certainly be executed in conjunction with other attack vectors such as cross-site scripting.

Developers are strongly encouraged to deploy the usual server-side defenses (CSRF tokens, ensuring that "safe" HTTP methods are idempotent, etc) to mitigate the risk more fully.

Additionally, client-side techniques such as those described in [app-isolation] may also prove effective against CSRF, and are certainly worth exploring in combination with "SameSite" cookies.

8.8.2. Top-level Navigations

Setting the "SameSite" attribute in "strict" mode provides robust defense in depth against CSRF attacks, but has the potential to confuse users unless sites' developers carefully ensure that their cookie-based session management systems deal reasonably well with top-level navigations.

Consider the scenario in which a user reads their email at MegaCorp Inc's webmail provider "https://site.example/". They might expect that clicking on an emailed link to "https://projects.example/secret/project" would show them the secret project that they're authorized to see, but if "projects.example" has marked their session cookies as "SameSite", then this cross-site navigation won't send them along with the request. "projects.example" will render a 404 error to avoid leaking secret information, and the user will be quite confused.

Developers can avoid this confusion by adopting a session management system that relies on not one, but two cookies: one conceptually granting "read" access, another granting "write" access. The latter could be marked as "SameSite", and its absence would prompt a reauthentication step before executing any non-idempotent action. The former could drop the "SameSite" attribute entirely, or choose the "Lax" version of enforcement, in order to allow users access to data via top-level navigation.

8.8.3. Mashups and Widgets

The "SameSite" attribute is inappropriate for some important use-cases. In particular, note that content intended for embedding in a cross-site contexts (social networking widgets or commenting services, for instance) will not have access to same-site cookies. Cookies may be required for requests triggered in these cross-site contexts in order to provide seamless functionality that relies on a user's state.

Likewise, some forms of Single-Sign-On might require cookie-based authentication in a cross-site context; these mechanisms will not function as intended with same-site cookies.

8.8.4. Server-controlled

SameSite cookies in and of themselves don't do anything to address the general privacy concerns outlined in Section 7.1 of [RFC6265]. The "SameSite" attribute is set by the server, and serves to mitigate the risk of certain kinds of attacks that the server is worried about. The user is not involved in this decision. Moreover, a number of side-channels exist which could allow a server to link distinct requests even in the absence of cookies (for example, connection and/or socket pooling between same-site and cross-site requests).

9. IANA Considerations

9.1. Cookie

The permanent message header field registry (see [RFC3864]) needs to be updated with the following registration:

Header field name: Cookie

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 5.5)

9.2. Set-Cookie

The permanent message header field registry (see [RFC3864]) needs to be updated with the following registration:

Header field name: Set-Cookie

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 5.3)

9.3. Cookie Attribute Registry

The "Cookie Attribute Registry" defines the name space of attribute used to control cookies' behavior. The registry is maintained at https://www.iana.org/assignments/cookie-attribute-names [4].

9.3.1. Procedure

Each registered attribute name is associated with a description, and a reference detailing how the attribute is to be processed and stored.

New registrations happen on a "RFC Required" basis (see Section 4.7 of [RFC8126]). The attribute to be registered MUST match the "extension-av" syntax defined in Section 4.1.1. Note that attribute names are generally defined in CamelCase, but technically accepted case-insensitively.

9.3.2. Registration

The "Cookie Attribute Registry" will be updated with the registrations below:

Name	Reference
Domain	Section 4.1.2.3 of this document
Expires	Section 4.1.2.1 of this document
HttpOnly	{{attribute-httponly} of this document
Max-Age	{{attribute-max-age} of this document
Path	{{attribute-path} of this document
SameSite	{{attribute-samesite} of this document
Secure	{{attribute-secure} of this document

10. References

10.1. Normative References

[FETCH]   van Kesteren, A., "Fetch", n.d., <https://fetch.spec.whatwg.org/>.

[HTML]   Hickson, I., Pieters, S., van Kesteren, A., Jaegenstedt, P., and D. Denicola, "HTML", n.d., <https://html.spec.whatwg.org/>.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <https://www.rfc-editor.org/info/rfc1123>.

[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>.

[RFC3490] Costello, A., "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003.

See Section 6.3 for an explanation why the normative reference to an obsoleted specification is needed.

[RFC4790] Newman, C., Duerst, M., and A. Gulbrandsen, "Internet Application Protocol Collation Registry", RFC 4790, DOI 10.17487/RFC4790, March 2007, <https://www.rfc-editor.org/info/rfc4790>.

[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>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

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

[RFC7230] 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>.

[RFC7231] 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>.

[RFC8126] 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/info/rfc8126>.

[SERVICE-WORKERS]
Russell, A., Song, J., and J. Archibald, "Service Workers", n.d., <http://www.w3.org/TR/service-workers/>.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

10.2. Informative References

[Aggarwal2010]
Aggarwal, G., Burzstein, E., Jackson, C., and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers", 2010, <http://www.usenix.org/events/sec10/tech/full_papers/Aggarwal.pdf>.

[app-isolation]
Chen, E., Bau, J., Reis, C., Barth, A., and C. Jackson, "App Isolation - Get the Security of Multiple Browsers with Just One", 2011, <http://www.collinjackson.com/research/papers/appisolation.pdf>.

[CSRF] Barth, A., Jackson, C., and J. Mitchell, "Robust Defenses for Cross-Site Request Forgery", DOI 10.1145/1455770.1455782, ISBN 978-1-59593-810-7, ACM CCS '08: Proceedings of the 15th ACM conference on Computer and communications security (pages 75-88), October 2008, <http://portal.acm.org/citation.cfm?id=1455770.1455782>.

[I-D.ietf-httpbis-cookie-alone]
West, M., "Deprecate modification of 'secure' cookies from non-secure origins", draft-ietf-httpbis-cookie-alone-01 (work in progress), September 2016.

[I-D.ietf-httpbis-cookie-prefixes]
West, M., "Cookie Prefixes", draft-ietf-httpbis-cookie-prefixes-00 (work in progress), February 2016.

[I-D.ietf-httpbis-cookie-same-site]
West, M. and M. Goodwin, "Same-Site Cookies", draft-ietf-httpbis-cookie-same-site-00 (work in progress), June 2016.

[prerendering]
Bentzel, C., "Chrome Prerendering", n.d., <https://www.chromium.org/developers/design-documents/prerender>.

[PSL] "Public Suffix List", n.d., <https://publicsuffix.org/list/>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc3864>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5895] Resnick, P. and P. Hoffman, "Mapping Characters for Internationalized Domain Names in Applications (IDNA) 2008", RFC 5895, DOI 10.17487/RFC5895, September 2010, <https://www.rfc-editor.org/info/rfc5895>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC7034] Ross, D. and T. Gondrom, "HTTP Header Field X-Frame-Options", RFC 7034, DOI 10.17487/RFC7034, October 2013, <https://www.rfc-editor.org/info/rfc7034>.

[UTS46] Davis, M. and M. Suignard, "Unicode IDNA Compatibility Processing", UNICODE Unicode Technical Standards # 46, June 2016, <http://unicode.org/reports/tr46/>.

10.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

[2] http://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/6265bis

[4] https://www.iana.org/assignments/cookie-attribute-names

[5] https://github.com/httpwg/http-extensions/issues/243

[6] https://github.com/httpwg/http-extensions/issues/246

[7] https://www.rfc-editor.org/errata_search.php?rfc=6265

[8] https://github.com/httpwg/http-extensions/issues/247

[9] https://github.com/httpwg/http-extensions/issues/201

[10] https://github.com/httpwg/http-extensions/issues/204

[11] https://github.com/httpwg/http-extensions/issues/222

[12] https://github.com/httpwg/http-extensions/issues/248

[13] https://github.com/httpwg/http-extensions/issues/295

[14] https://github.com/httpwg/http-extensions/issues/302

[15] https://github.com/httpwg/http-extensions/issues/389

[16] https://github.com/httpwg/http-extensions/issues/199

[17] https://github.com/httpwg/http-extensions/issues/788

[18] https://github.com/httpwg/http-extensions/issues/594

[19] https://github.com/httpwg/http-extensions/issues/159

[20] https://github.com/httpwg/http-extensions/issues/159

[21] https://github.com/httpwg/http-extensions/issues/901

[22] https://github.com/httpwg/http-extensions/pull/1035

[23] https://github.com/httpwg/http-extensions/pull/1038

[24] https://github.com/httpwg/http-extensions/pull/1040

[25] https://github.com/httpwg/http-extensions/pull/1047

[26] https://github.com/httpwg/http-extensions/issues/1059

[27] https://github.com/httpwg/http-extensions/issues/1158

[28] https://github.com/httpwg/http-extensions/pull/1060

[29] https://github.com/httpwg/http-extensions/issues/1074

[30] https://github.com/httpwg/http-extensions/issues/1119

[31] https://github.com/httpwg/http-extensions/pull/1143

[32] https://github.com/httpwg/http-extensions/issues/1159

Appendix A. Changes

A.1. draft-ietf-httpbis-rfc6265bis-00

- Port [RFC6265] to Markdown. No (intentional) normative changes.

A.2. draft-ietf-httpbis-rfc6265bis-01

- Fixes to formatting caused by mistakes in the initial port to Markdown:

  * https://github.com/httpwg/http-extensions/issues/243 [5]

  * https://github.com/httpwg/http-extensions/issues/246 [6]

- Addresses errata 3444 by updating the "path-value" and "extension-av" grammar, errata 4148 by updating the "day-of-month", "year", and "time" grammar, and errata 3663 by adding the requested note. https://www.rfc-editor.org/errata_search.php?rfc=6265 [7]

- Dropped "Cookie2" and "Set-Cookie2" from the IANA Considerations section: https://github.com/httpwg/http-extensions/issues/247 [8]

- Merged the recommendations from [I-D.ietf-httpbis-cookie-alone], removing the ability for a non-secure origin to set cookies with a 'secure' flag, and to overwrite cookies whose 'secure' flag is true.

- Merged the recommendations from [I-D.ietf-httpbis-cookie-prefixes], adding "__Secure-" and "__Host-" cookie name prefix processing instructions.

A.3. draft-ietf-httpbis-rfc6265bis-02

- Merged the recommendations from [I-D.ietf-httpbis-cookie-same-site], adding support for the "SameSite" attribute.

- Closed a number of editorial bugs:

  * Clarified address bar behavior for SameSite cookies: https://github.com/httpwg/http-extensions/issues/201 [9]

* Added the word "Cookies" to the document's name: https://github.com/httpwg/http-extensions/issues/204 [10]

* Clarified that the "__Host-" prefix requires an explicit "Path" attribute: https://github.com/httpwg/http-extensions/issues/222 [11]

* Expanded the options for dealing with third-party cookies to include a brief mention of partitioning based on first-party: https://github.com/httpwg/http-extensions/issues/248 [12]

* Noted that double-quotes in cookie values are part of the value, and are not stripped: https://github.com/httpwg/http-extensions/issues/295 [13]

* Fixed the "site for cookies" algorithm to return something that makes sense: https://github.com/httpwg/http-extensions/issues/302 [14]

A.4. draft-ietf-httpbis-rfc6265bis-03

o Clarified handling of invalid SameSite values: https://github.com/httpwg/http-extensions/issues/389 [15]

o Reflect widespread implementation practice of including a cookie's "host-only-flag" when calculating its uniqueness: https://github.com/httpwg/http-extensions/issues/199 [16]

o Introduced an explicit "None" value for the SameSite attribute: https://github.com/httpwg/http-extensions/issues/788 [17]

A.5. draft-ietf-httpbis-rfc6265bis-04

o Allow "SameSite" cookies to be set for all top-level navigations. https://github.com/httpwg/http-extensions/issues/594 [18]

o Treat "Set-Cookie: token" as creating the cookie "("", "token")": https://github.com/httpwg/http-extensions/issues/159 [19]

o Reject cookies with neither name nor value (e.g. "Set-Cookie: =" and "Set-Cookie:": https://github.com/httpwg/http-extensions/issues/159 [20]

o Clarified behavior of multiple "SameSite" attributes in a cookie string: https://github.com/httpwg/http-extensions/issues/901 [21]

A.6. draft-ietf-httpbis-rfc6265bis-05

- Typos and editorial fixes: https://github.com/httpwg/http-extensions/pull/1035 [22], https://github.com/httpwg/http-extensions/pull/1038 [23], https://github.com/httpwg/http-extensions/pull/1040 [24], https://github.com/httpwg/http-extensions/pull/1047 [25].

A.7. draft-ietf-httpbis-rfc6265bis-06

- Editorial fixes: https://github.com/httpwg/http-extensions/issues/1059 [26], https://github.com/httpwg/http-extensions/issues/1158 [27].

- Created a registry for cookie attribute names: https://github.com/httpwg/http-extensions/pull/1060 [28].

- Tweaks to ABNF for "cookie-pair" and the "Cookie" header production: https://github.com/httpwg/http-extensions/issues/1074 [29], https://github.com/httpwg/http-extensions/issues/1119 [30].

- Fixed serialization for nameless/valueless cookies: https://github.com/httpwg/http-extensions/pull/1143 [31].

- Converted a normative reference to Mozilla's Public Suffix List [PSL] into an informative reference: https://github.com/httpwg/http-extensions/issues/1159 [32].

Acknowledgements

RFC 6265 was written by Adam Barth. This document is a minor update of RFC 6265, adding small features, and aligning the specification with the reality of today's deployments. Here, we're standing upon the shoulders of a giant since the majority of the text is still Adam's.

Authors' Addresses

Mike West (editor)
Google, Inc

Email: mkwst@google.com
URI: https://mikewest.org/

John Wilander (editor)
Apple, Inc

Email: wilander@apple.com

HTTP M. Nottingham
Internet-Draft Fastly
Updates: 7234 (if approved) November 2, 2019
Intended status: Standards Track
Expires: May 5, 2020

HTTP Representation Variants
draft-ietf-httpbis-variants-06

Abstract

This specification introduces an alternative way to select a HTTP response from a cache based upon its request headers, using the HTTP "Variants" and "Variant-Key" response header fields. Its aim is to make HTTP proactive content negotiation more cache-friendly.

Note to Readers

RFC EDITOR: please remove this section before publication

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at https://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/variants [3].

There is a prototype implementation of the algorithms herein at https://github.com/mnot/variants-toy [4].

Copyright Notice

Table of Contents

1. Introduction

HTTP proactive content negotiation ([RFC7231], Section 3.4.1) is seeing renewed interest, both for existing request headers like Accept-Language and for newer ones (for example, see [I-D.ietf-httpbis-client-hints]).

Successfully reusing negotiated responses that have been stored in a HTTP cache requires establishment of a secondary cache key ([RFC7234], Section 4.1). Currently, the Vary header ([RFC7231], Section 7.1.4) does this by nominating a set of request headers. Their values collectively form the secondary cache key for a given response.

HTTP’s caching model allows a certain amount of latitude in normalising those request header field values, so as to increase the chances of a cache hit while still respecting the semantics of that header. However, normalisation is not formally defined, leading to infrequent implementation in cache, and divergence of behaviours when it is.

Even when the headers’ semantics are understood, a cache does not know enough about the possible alternative representations available on the origin server to make an appropriate decision.

For example, if a cache has stored the following request/response pair:

```
GET /foo HTTP/1.1
Host: www.example.com
Accept-Language: en;q=0.5, fr;q=1.0


HTTP/1.1 200 OK
Content-Type: text/html
Content-Language: en
Vary: Accept-Language
Transfer-Encoding: chunked

[English content]
```

Provided that the cache has full knowledge of the semantics of Accept-Language and Content-Language, it will know that an English representation is available and might be able to infer that a French representation is not available. But, it does not know (for example) whether a Japanese representation is available without making another request, incurring possibly unnecessary latency.

This specification introduces the HTTP Variants response header field (Section 2) to enumerate the available variant representations on the origin server, to provide clients and caches with enough information to properly satisfy requests - either by selecting a response from cache or by forwarding the request towards the origin - by following the algorithm defined in Section 4.

Its companion Variant-Key response header field (Section 3) indicates the applicable key(s) that the response is associated with, so that it can be reliably reused in the future. Effectively, it allows the specification of a request header field to define how it affects the secondary cache key.

When this specification is in use, the example above might become:

```
GET /foo HTTP/1.1
Host: www.example.com
Accept-Language: en;q=0.5, fr;q=1.0


HTTP/1.1 200 OK
Content-Type: text/html
Content-Language: en
Vary: Accept-Language
Variants: Accept-Language;de;en;jp
Variant-Key: en
Transfer-Encoding: chunked

[English content]
```

Proactive content negotiation mechanisms that wish to be used with Variants need to define how to do so explicitly; see Section 6. As a result, it is best suited for negotiation over request headers that are well-understood.

Variants also works best when content negotiation takes place over a constrained set of representations; since each variant needs to be listed in the header field, it is ill-suited for open-ended sets of representations.

Variants can be seen as a simpler version of the Alternates header field introduced by [RFC2295]; unlike that mechanism, Variants does not require specification of each combination of attributes, and does not assume that each combination has a unique URL.

1.1. 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 specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] but relies on Structured Headers from [I-D.ietf-httpbis-header-structure] for parsing.

Additionally, it uses the "field-name" rule from [RFC7230], "type", "subtype", "content-coding" and "language-range" from [RFC7231], and "cookie-name" from [RFC6265].

2. The "Variants" HTTP Header Field

The Variants HTTP response header field indicates what representations are available for a given resource at the time that the response is produced, by enumerating the request header fields that it varies on, along with a representation of the values that are available for each.

Variants is a Structured Header Dictionary (Section 3.2 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

```
Variants        = sh-dict
```

Each member-name represents the field-name of a request header that is part of the secondary cache key; each member-value is an inner-list of strings or tokens that convey representations of potential values for that header field, hereafter referred to as "available-values".

If Structured Header parsing fails or a member's value does have the structure outlined above, the client MUST treat the representation as having no Variants header field.

Note that an available-value that is a token is interpreted as a string containing the same characters, and vice versa.

So, given this example header field:

```
Variants: Accept-Encoding=(gzip)
```

a recipient can infer that the only content-coding available for that resource is "gzip" (along with the "identity" non-encoding; see Appendix A.2).

Given:

```
Variants: accept-encoding=()
```

a recipient can infer that no content-codings (beyond identity) are supported. Note that as always, field-name is case-insensitive.

A more complex example:

```
Variants: Accept-Encoding=(gzip br), Accept-Language=(en fr)
```

Here, recipients can infer that two content-codings in addition to "identity" are available, as well as two content languages. Note that, as with all Structured Header dictionaries, they might occur in the same header field or separately, like this:

```
Variants: Accept-Encoding=(gzip brotli)
Variants: Accept-Language=(en fr)
```

The ordering of available-values is significant, as it might be used by the header's algorithm for selecting a response (in this example, the first language is the default; see Appendix A.3).

The ordering of the request header fields themselves indicates descending application of preferences; in the example above, a cache that has all of the possible permutations stored will honour the client's preferences for Accept-Encoding before honouring Accept-Language.

Origin servers SHOULD consistently send Variant header fields on all cacheable (as per [RFC7234], Section 3) responses for a resource, since its absence will trigger caches to fall back to Vary processing.

Likewise, servers MUST send the Variant-Key response header field when sending Variants, since its absence means that the stored response will not be reused when this specification is implemented.

RFC EDITOR: Please remove the next paragraph before publication.

Implementations of drafts of this specification MUST implement an HTTP header field named "Variants-##" instead of the "Variants" header field specified by the final RFC, with "##" replaced by the draft number being implemented. For example, implementations of draft-ietf-httpbis-variants-05 would implement "Variants-05".

2.1. Relationship to Vary

This specification updates [RFC7234] to allow caches that implement it to ignore request header fields in the Vary header for the purposes of secondary cache key calculation ([RFC7234], Section 4.1) when their semantics are implemented as per this specification and their corresponding response header field is listed in Variants.

If any member of the Vary header does not have a corresponding variant that is understood by the implementation, it is still subject to the requirements there.

See Section 5.1.3 for an example.

In practice, implementation of Vary varies considerably. As a result, cache efficiency might drop considerably when Variants does not contain all of the headers referenced by Vary, because some implementations might choose to disable Variants processing when this is the case.

3. The "Variant-Key" HTTP Header Field

The Variant-Key HTTP response header field identifies one or more sets of available-values that identify the secondary cache key(s) that the response it occurs within are associated with.

Variant-Key is a Structured Header List (Section 3.1 of [I-D.ietf-httpbis-header-structure]) whose members are inner-lists of strings or tokens. Its ABNF is:

```
Variant-Key      = sh-list
```

Each member MUST be an inner-list, and MUST itself have the same number of members as there are members of the representation's Variants header field. If not, the client MUST treat the representation as having no Variant-Key header field.

Each member identifies a list of available-values corresponding to the header field-names in the Variants header field, thereby identifying a secondary cache key that can be used with this response. These available-values do not need to explicitly appear in the Variants header field; they can be interpreted by the algorithm specific to processing that field. For example, Accept-Encoding defines an implicit "identity" available-value (Appendix A.2).

Each inner-list member is treated as identifying an available-value for the corresponding variant-axis' field-name. Any list-member that is a token is interpreted as a string containing the same characters.

For example:

```
Variants: Accept-Encoding=(gzip br), Accept-Language=(en fr)
Variant-Key: (gzip fr)
```

This header pair indicates that the representation has a "gzip" content-coding and "fr" content-language.

If the response can be used to satisfy more than one request, they can be listed in additional members. For example:

```
Variants: Accept-Encoding=(gzip br), Accept-Language=(en fr)
Variant-Key: (gzip fr), ("identity" fr)
```

indicates that this response can be used for requests whose Accept-Encoding algorithm selects "gzip" or "identity", as long as the Accept-Language algorithm selects "fr" - perhaps because there is no gzip-compressed French representation.

When more than one Variant-Key value is in a response, the first one present MUST correspond to the request that caused that response to be generated. For example:

```
Variants: Accept-Encoding=(gzip br), Accept-Language=(en fr)
Variant-Key: (gzip fr), (identity fr), (br fr oops)
```

is treated as if the Variant-Key header were completely absent, which will tend to disable caching for the representation that contains it.

Note that in

```
Variant-Key: (gzip  fr)
Variant-Key: ("gzip " fr)
```

The whitespace after "gzip" in the first header field value is excluded by the parsing algorithm, but the whitespace in the second header field value is included by the string parsing algorithm. This will likely cause the second header field value to fail to match client requests.

RFC EDITOR: Please remove the next paragraph before publication.

Implementations of drafts of this specification MUST implement an HTTP header field named "Variant-Key-##" instead of the "Variant-Key" header field specified by the final RFC, with "##" replaced by the draft number being implemented. For example, implementations of draft-ietf-httpbis-variants-05 would implement "Variant-Key-05".

4. Cache Behaviour

Caches that implement the Variants header field and the relevant semantics of the field-names it contains can use that knowledge to either select an appropriate stored representation, or forward the request if no appropriate representation is stored.

They do so by running this algorithm (or its functional equivalent) upon receiving a request:

Given incoming-request (a mapping of field-names to field-values, after being combined as allowed by Section 3.2.2 of [RFC7230]), and stored-responses (a list of stored responses suitable for reuse as defined in Section 4 of [RFC7234], excepting the requirement to calculate a secondary cache key):

1. If stored-responses is empty, return an empty list.

2. Order stored-responses by the "Date" header field, most recent to least recent.

3. Let sorted-variants be an empty list.

4. If the freshest member of stored-responses (as per [RFC7234], Section 4.2) has one or more "Variants" header field(s) that successfully parse according to Section 2:

    1. Select one member of stored-responses with a "Variants" header field-value(s) that successfully parses according to Section 2 and let variants-header be this parsed value. This SHOULD be the most recent response, but MAY be from an older one as long as it is still fresh.

    2. For each variant-axis in variants-header:

        1. If variant-axis' field-name corresponds to the request header field identified by a content negotiation mechanism that the implementation supports:

            1. Let request-value be the field-value associated with field-name in incoming-request, or null if field-name is not in incoming-request.

2. Let sorted-values be the result of running the algorithm defined by the content negotiation mechanism with request-value and variant-axis' available-values.

3. Append sorted-values to sorted-variants.

At this point, sorted-variants will be a list of lists, each member of the top-level list corresponding to a variant-axis in the Variants header field-value, containing zero or more items indicating available-values that are acceptable to the client, in order of preference, greatest to least.

5. Return result of running Compute Possible Keys (Section 4.1) on sorted-variants, an empty list and an empty list.

This returns a list of lists of strings suitable for comparing to the parsed Variant-Keys (Section 3) that represent possible responses on the server that can be used to satisfy the request, in preference order, provided that their secondary cache key (after removing the headers covered by Variants) matches. Section 4.2 illustrates one way to do this.

4.1. Compute Possible Keys

This algorithm computes the cross-product of the elements of key-facets.

Given key-facets (a list of lists of strings), and key-stub (a list of strings representing a partial key), and possible-keys (a list of lists of strings):

1. Let values be the first member of key-facets.

2. Let remaining-facets be a copy of all of the members of key-facets except the first.

3. For each value in values:

    1. Let this-key be a copy of key-stub.

    2. Append value to this-key.

    3. If remaining-facets is empty, append this-key to possible-keys.

    4. Otherwise, run Compute Possible Keys on remaining-facets, this-key and possible-keys.

4. Return possible-keys.

4.2. Check Vary

This algorithm is an example of how an implementation can meet the requirement to apply the members of the Vary header field that are not covered by Variants.

Given incoming-request (a mapping of field-names to field-values, after being combined as allowed by Section 3.2.2 of [RFC7230]), and stored-response (a stored response):

1. Let filtered-vary be the field-value(s) of stored-response's "Vary" header field.

2. Let processed-variants be a list containing the request header fields that identify the content negotiation mechanisms supported by the implementation.

3. Remove any member of filtered-vary that is a case-insensitive match for a member of processed-variants.

4. If the secondary cache key (as calculated in [RFC7234], Section 4.1) for stored_response matches incoming-request, using filtered-vary for the value of the "Vary" response header, return True.

5. Return False.

This returns a Boolean that indicates whether stored-response can be used to satisfy the request.

Note that implementation of the Vary header field varies in practice, and the algorithm above illustrates only one way to apply it. It is equally viable to forward the request if there is a request header listed in Vary but not Variants.

4.3. Example of Cache Behaviour

For example, if the selected variants-header was:

```
Variants: Accept-Language=(en fr de), Accept-Encoding=(gzip br)
```

and the request contained the headers:

```
Accept-Language: fr;q=1.0, en;q=0.1
Accept-Encoding: gzip
```

Then the sorted-variants would be:

```
[
  ["fr", "en"]         // prefers French, will accept English
  ["gzip", "identity"] // prefers gzip encoding, will accept identity
]
```

Which means that the result of the Cache Behaviour algorithm would be:

```
[
  ["fr", "gzip"],
  ["fr", "identity"],
  ["en", "gzip"],
  ["en", "identity"]
]
```

Representing a first preference of a French, gzip'd response. Thus, if a cache has a response with:

```
Variant-Key: (fr gzip)
```

it could be used to satisfy the first preference. If not, responses corresponding to the other keys could be returned, or the request could be forwarded towards the origin.

4.3.1. A Variant Missing From the Cache

If the selected variants-header was:

```
Variants: Accept-Language=(en fr de)
```

And a request comes in with the following headers:

```
Accept-Language: de;q=1.0, es;q=0.8
```

Then sorted-variants in Cache Behaviour is:

```
[
  ["de"]         // prefers German; will not accept English
]
```

If the cache contains responses with the following Variant-Keys:

```
Variant-Key: (fr)
Variant-Key: (en)
```

Then the cache needs to forward the request to the origin server, since Variants indicates that "de" is available, and that is acceptable to the client.

4.3.2. Variants That Don't Overlap the Client's Request

If the selected variants-header was:

```
Variants: Accept-Language=(en fr de)
```

And a request comes in with the following headers:

```
Accept-Language: es;q=1.0, ja;q=0.8
```

Then sorted-variants in Cache Behaviour are:

```
[
  ["en"]
]
```

This allows the cache to return a "Variant-Key: en" response even though it's not in the set the client prefers.

5. Origin Server Behaviour

Origin servers that wish to take advantage of Variants will need to generate both the Variants (Section 2) and Variant-Key (Section 3) header fields in all cacheable responses for a given resource. If either is omitted and the response is stored, it will have the effect of disabling caching for that resource until it is no longer stored (e.g., it expires, or is evicted).

Likewise, origin servers will need to assure that the members of both header field values are in the same order and have the same length, since discrepancies will cause caches to avoid using the responses they occur in.

The value of the Variants header should be relatively stable for a given resource over time; when it changes, it can have the effect of invalidating previously stored responses.

As per Section 2.1, the Vary header is required to be set appropriately when Variants is in use, so that caches that do not implement this specification still operate correctly.

Origin servers are advised to carefully consider which content negotiation mechanisms to enumerate in Variants; if a mechanism is not supported by a receiving cache, it will "downgrade" to Vary handling, which can negatively impact cache efficiency.

5.1. Examples

The operation of Variants is illustrated by the examples below.

5.1.1. Single Variant

Given a request/response pair:

```
GET /clancy HTTP/1.1
Host: www.example.com
Accept-Language: en;q=1.0, fr;q=0.5


HTTP/1.1 200 OK
Content-Type: image/gif
Content-Language: en
Cache-Control: max-age=3600
Variants: Accept-Language=(en de)
Variant-Key: (en)
Vary: Accept-Language
Transfer-Encoding: chunked
```

Upon receipt of this response, the cache knows that two representations of this resource are available, one with a language of "en", and another whose language is "de".

Subsequent requests (while this response is fresh) will cause the cache to either reuse this response or forward the request, depending on what the selection algorithm determines.

So, if a request with "en" in Accept-Language is received and its q-value indicates that it is acceptable, the stored response is used. A request that indicates that "de" is acceptable will be forwarded to the origin, thereby populating the cache. A cache receiving a request that indicates both languages are acceptable will use the q-value to make a determination of what response to return.

A cache receiving a request that does not list either language as acceptable (or does not contain an Accept-Language at all) will return the "en" representation (possibly fetching it from the origin), since it is listed first in the Variants list.

Note that Accept-Language is listed in Vary, to assure backwards-compatibility with caches that do not support Variants.

5.1.2. Multiple Variants

A more complicated request/response pair:

```
GET /murray HTTP/1.1
Host: www.example.net
Accept-Language: en;q=1.0, fr;q=0.5
Accept-Encoding: gzip, br
```

```
HTTP/1.1 200 OK
Content-Type: image/gif
Content-Language: en
Content-Encoding: br
Variants: Accept-Language=(en jp de)
Variants: Accept-Encoding=(br gzip)
Variant-Key: (en br)
Vary: Accept-Language, Accept-Encoding
Transfer-Encoding: chunked
```

Here, the cache knows that there are two axes that the response varies upon; language and encoding. Thus, there are a total of nine possible representations for the resource (including the identity encoding), and the cache needs to consider the selection algorithms for both axes.

Upon a subsequent request, if both selection algorithms return a stored representation, it can be served from cache; otherwise, the request will need to be forwarded to origin.

5.1.3. Partial Coverage

Now, consider the previous example, but where only one of the Vary'd axes (encoding) is listed in Variants:

```
GET /bar HTTP/1.1
Host: www.example.net
Accept-Language: en;q=1.0, fr;q=0.5
Accept-Encoding: gzip, br
```

```
HTTP/1.1 200 OK
Content-Type: image/gif
Content-Language: en
Content-Encoding: br
Variants: Accept-Encoding=(br gzip)
Variant-Key: (br)
Vary: Accept-Language, Accept-Encoding
Transfer-Encoding: chunked
```

Here, the cache will need to calculate a secondary cache key as per [RFC7234], Section 4.1 - but considering only Accept-Language to be in its field-value - and then continue processing Variants for the set of stored responses that the algorithm described there selects.

6. Defining Content Negotiation Using Variants

To be usable with Variants, proactive content negotiation mechanisms need to be specified to take advantage of it. Specifically, they:

- MUST define a request header field that advertises the clients preferences or capabilities, whose field-name SHOULD begin with "Accept-".

- MUST define the syntax of an available-value that will occur in Variants and Variant-Key.

- MUST define an algorithm for selecting a result. It MUST return a list of available-values that are suitable for the request, in order of preference, given the value of the request header nominated above (or null if the request header is absent) and an available-values list from the Variants header. If the result is an empty list, it implies that the cache cannot satisfy the request.

Appendix A fulfils these requirements for some existing proactive content negotiation mechanisms in HTTP.

7. IANA Considerations

This specification registers the following entry in the Permanent Message Header Field Names registry established by [RFC3864]:

- Header field name: Variants

- Applicable protocol: http

- Status: standard

o Author/Change Controller: IETF

o Specification document(s): [this document]

o Related information:

This specification registers the following entry in the Permanent Message Header Field Names registry established by [RFC3864]:

o Header field name: Variant-Key

o Applicable protocol: http

o Status: standard

o Author/Change Controller: IETF

o Specification document(s): [this document]

o Related information:

8. Security Considerations

If the number or advertised characteristics of the representations available for a resource are considered sensitive, the Variants header by its nature will leak them.

Note that the Variants header is not a commitment to make representations of a certain nature available; the runtime behaviour of the server always overrides hints like Variants.

9. References

9.1. Normative References

[I-D.ietf-httpbis-header-structure]
Nottingham, M. and P. Kamp, "Structured Headers for HTTP", draft-ietf-httpbis-header-structure-13 (work in progress), August 2019.

[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>.

[RFC4647] Phillips, A. and M. Davis, "Matching of Language Tags", BCP 47, RFC 4647, DOI 10.17487/RFC4647, September 2006, <https://www.rfc-editor.org/info/rfc4647>.

[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>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC7230] 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>.

[RFC7231] 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>.

[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>.

[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>.

9.2. Informative References

[I-D.ietf-httpbis-client-hints] Grigorik, I., "HTTP Client Hints", draft-ietf-httpbis-client-hints-07 (work in progress), March 2019.

[RFC2295] Holtman, K. and A. Mutz, "Transparent Content Negotiation in HTTP", RFC 2295, DOI 10.17487/RFC2295, March 1998, <https://www.rfc-editor.org/info/rfc2295>.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc3864>.

9.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

[2] https://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/variants

[4] https://github.com/mnot/variants-toy

Appendix A. Variants for Existing Content Negotiation Mechanisms

This appendix defines the required information to use existing proactive content negotiation mechanisms (as defined in [RFC7231], Section 5.3) with the Variants header field.

A.1. Accept

This section defines variant handling for the Accept request header (section 5.3.2 of [RFC7231]).

The syntax of an available-value for Accept is:

accept-available-value = type "/" subtype

To perform content negotiation for Accept given a request-value and available-values:

1. Let preferred-available be an empty list.

2. Let preferred-types be a list of the types in the request-value (or the empty list if request-value is null), ordered by their weight, highest to lowest, as per Section 5.3.2 of [RFC7231] (omitting any coding with a weight of 0). If a type lacks an explicit weight, an implementation MAY assign one.

3. For each preferred-type in preferred-types:

    1. If any member of available-values matches preferred-type, using the media-range matching mechanism specified in Section 5.3.2 of [RFC7231] (which is case-insensitive), append those members of available-values to preferred-available (preserving the precedence order implied by the media ranges' specificity).

4. If preferred-available is empty, append the first member of available-values to preferred-available. This makes the first available-value the default when none of the client's preferences are available.

5. Return preferred-available.

Note that this algorithm explicitly ignores extension parameters on media types (e.g., "charset").

A.2. Accept-Encoding

This section defines variant handling for the Accept-Encoding request header (section 5.3.4 of [RFC7231]).

The syntax of an available-value for Accept-Encoding is:

accept-encoding-available-value = content-coding / "identity"

To perform content negotiation for Accept-Encoding given a request-value and available-values:

1. Let preferred-available be an empty list.

2. Let preferred-codings be a list of the codings in the request-value (or the empty list if request-value is null), ordered by their weight, highest to lowest, as per Section 5.3.1 of [RFC7231] (omitting any coding with a weight of 0). If a coding lacks an explicit weight, an implementation MAY assign one.

3. If "identity" is not a member of preferred-codings, append "identity".

4. Append "identity" to available-values.

5. For each preferred-coding in preferred-codings:

    1. If there is a case-insensitive, character-for-character match for preferred-coding in available-values, append that member of available-values to preferred-available.

6. Return preferred-available.

Note that the unencoded variant needs to have a Variant-Key header field with a value of "identity" (as defined in Section 5.3.4 of [RFC7231]).

A.3. Accept-Language

This section defines variant handling for the Accept-Language request header (section 5.3.5 of [RFC7231]).

The syntax of an available-value for Accept-Language is:

accept-encoding-available-value = language-range

To perform content negotiation for Accept-Language given a request-value and available-values:

1. Let preferred-available be an empty list.

2. Let preferred-langs be a list of the language-ranges in the request-value (or the empty list if request-value is null), ordered by their weight, highest to lowest, as per Section 5.3.1 of [RFC7231] (omitting any language-range with a weight of 0). If a language-range lacks a weight, an implementation MAY assign one.

3. For each preferred-lang in preferred-langs:

    1. If any member of available-values matches preferred-lang, using either the Basic or Extended Filtering scheme defined in Section 3.3 of [RFC4647], append those members of available-values to preferred-available (preserving their order).

4. If preferred-available is empty, append the first member of available-values to preferred-available. This makes the first available-value the default when none of the client's preferences are available.

5. Return preferred-available.

A.4. Cookie

This section defines variant handling for the Cookie request header ([RFC6265]).

This syntax of an available-value for Cookie is:

```
cookie-available-value = cookie-name
```

To perform content negotiation for Cookie given a request-value and available-values:

1. Let cookies-available be an empty list.

2. For each available-value of available-values:

    1. Parse request-value as a Cookie header field [RFC6265] and let request-cookie-value be the cookie-value corresponding to a cookie with a cookie-name that matches available-value. If no match is found, continue to the next available-value.

    2. append request-cookie-value to cookies-available.

3. Return cookies-available.

A simple example is allowing a page designed for users that aren't logged in (denoted by the "logged_in" cookie-name) to be cached:

```
Variants: Cookie=(logged_in)
Variant-Key: (0)
Vary: Cookie
```

Here, a cache that implements Variants will only use this response to satisfy requests with "Cookie: logged_in=0". Caches that don't implement Variants will vary the response on all Cookie headers.

Or, consider this example:

```
Variants: Cookie=(user_priority)
Variant-Key: (silver), ("bronze")
Vary: Cookie
```

Here, the "user_priority" cookie-name allows requests from "gold" users to be separated from "silver" and "bronze" ones; this response is only served to the latter two.

It is possible to target a response to a single user; for example:

```
Variants: Cookie=(user_id)
Variant-Key: (some_person)
Vary: Cookie
```

Here, only the "some_person" "user_id" will have this response served to them again.

Note that if more than one cookie-name serves as a cache key, they'll need to be listed in separate Variants members, like this:

```
Variants: Cookie=(user_priority), Cookie=(user_region)
Variant-Key: (gold europe)
Vary: Cookie
```

Acknowledgements

This protocol is conceptually similar to, but simpler than, Transparent Content Negotiation [RFC2295]. Thanks to its authors for their inspiration.

It is also a generalisation of a Fastly VCL feature designed by Rogier 'DocWilco' Mulhuijzen.

Thanks to Hooman Beheshti, Ilya Grigorik, Leif Hedstrom, and Jeffrey Yasskin for their review and input.

Author's Address

Mark Nottingham
Fastly

Email: mnot@mnot.net
URI: https://www.mnot.net/

QUIC M. Bishop, Ed.
Internet-Draft Akamai
Intended status: Standards Track 20 October 2020
Expires: 23 April 2021

Hypertext Transfer Protocol Version 3 (HTTP/3)
draft-ietf-quic-http-32

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to HTTP/3.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic.

Working Group information can be found at https://github.com/quicwg; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-http.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

HTTP semantics ([SEMANTICS]) are used for a broad range of services on the Internet. These semantics have most commonly been used with HTTP/1.1, over a variety of transport and session layers, and with HTTP/2 over TLS. HTTP/3 supports the same semantics over a new transport protocol, QUIC.

1.1. Prior versions of HTTP

HTTP/1.1 ([HTTP11]) uses whitespace-delimited text fields to convey HTTP messages. While these exchanges are human-readable, using whitespace for message formatting leads to parsing complexity and excessive tolerance of variant behavior. Because HTTP/1.x does not include a multiplexing layer, multiple TCP connections are often used to service requests in parallel. However, that has a negative impact on congestion control and network efficiency, since TCP does not share congestion control across multiple connections.

HTTP/2 ([HTTP2]) introduced a binary framing and multiplexing layer to improve latency without modifying the transport layer. However, because the parallel nature of HTTP/2's multiplexing is not visible to TCP's loss recovery mechanisms, a lost or reordered packet causes all active transactions to experience a stall regardless of whether that transaction was directly impacted by the lost packet.

1.2. Delegation to QUIC

The QUIC transport protocol incorporates stream multiplexing and per-stream flow control, similar to that provided by the HTTP/2 framing layer. By providing reliability at the stream level and congestion control across the entire connection, QUIC has the capability to improve the performance of HTTP compared to a TCP mapping. QUIC also incorporates TLS 1.3 ([TLS13]) at the transport layer, offering comparable security to running TLS over TCP, with the improved connection setup latency of TCP Fast Open ([TFO]).

This document defines a mapping of HTTP semantics over the QUIC transport protocol, drawing heavily on the design of HTTP/2. While delegating stream lifetime and flow control issues to QUIC, a similar binary framing is used on each stream. Some HTTP/2 features are subsumed by QUIC, while other features are implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [HTTP2].

2. HTTP/3 Protocol Overview

HTTP/3 provides a transport for HTTP semantics using the QUIC transport protocol and an internal framing layer similar to HTTP/2.

Once a client knows that an HTTP/3 server exists at a certain endpoint, it opens a QUIC connection. QUIC provides protocol negotiation, stream-based multiplexing, and flow control. Discovery of an HTTP/3 endpoint is described in Section 3.2.

Within each stream, the basic unit of HTTP/3 communication is a frame (Section 7.2). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 4.1).

Multiplexing of requests is performed using the QUIC stream abstraction, described in Section 2 of [QUIC-TRANSPORT]. Each request-response pair consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

Server push is an interaction mode introduced in HTTP/2 ([HTTP2]) that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. Several HTTP/3 frames are used to manage server push, such as PUSH_PROMISE, MAX_PUSH_ID, and CANCEL_PUSH.

As in HTTP/2, request and response fields are compressed for transmission. Because HPACK ([HPACK]) relies on in-order transmission of compressed field sections (a guarantee not provided by QUIC), HTTP/3 replaces HPACK with QPACK ([QPACK]). QPACK uses separate unidirectional streams to modify and track field table state, while encoded field sections refer to the state of the table without modifying it.

2.1. Document Organization

The following sections provide a detailed overview of the lifecycle of an HTTP/3 connection:

* Connection Setup and Management (Section 3) covers how an HTTP/3 endpoint is discovered and an HTTP/3 connection is established.

* HTTP Request Lifecycle (Section 4) describes how HTTP semantics are expressed using frames.

* Connection Closure (Section 5) describes how HTTP/3 connections are terminated, either gracefully or abruptly.

The details of the wire protocol and interactions with the transport are described in subsequent sections:

* Stream Mapping and Usage (Section 6) describes the way QUIC streams are used.

* HTTP Framing Layer (Section 7) describes the frames used on most streams.

* Error Handling (Section 8) describes how error conditions are handled and expressed, either on a particular stream or for the connection as a whole.

Additional resources are provided in the final sections:

* Extensions to HTTP/3 (Section 9) describes how new capabilities can be added in future documents.

* A more detailed comparison between HTTP/2 and HTTP/3 can be found in Appendix A.

2.2. Conventions and Terminology

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 variable-length integer encoding from [QUIC-TRANSPORT].

The following terms are used:

abort: An abrupt termination of a connection or stream, possibly due to an error condition.

client: The endpoint that initiates an HTTP/3 connection. Clients send HTTP requests and receive HTTP responses.

connection: A transport-layer connection between two endpoints, using QUIC as the transport protocol.

connection error: An error that affects the entire HTTP/3 connection.

endpoint: Either the client or server of the connection.

frame: The smallest unit of communication on a stream in HTTP/3, consisting of a header and a variable-length sequence of bytes structured according to the frame type. Protocol elements called "frames" exist in both this document and [QUIC-TRANSPORT]. Where frames from [QUIC-TRANSPORT] are referenced, the frame name will be prefaced with "QUIC." For example, "QUIC CONNECTION_CLOSE frames." References without this preface refer to frames defined in Section 7.2.

HTTP/3 connection: A QUIC connection where the negotiated application protocol is HTTP/3.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

receiver: An endpoint that is receiving frames.

sender: An endpoint that is transmitting frames.

server: The endpoint that accepts an HTTP/3 connection. Servers receive HTTP requests and send HTTP responses.

stream: A bidirectional or unidirectional bytestream provided by the QUIC transport.

stream error: An error on the individual HTTP/3 stream.

The term "payload body" is defined in Section 5.5.4 of [SEMANTICS].

Finally, the terms "gateway", "intermediary", "proxy", and "tunnel" are defined in Section 3.7 of [SEMANTICS]. Intermediaries act as both client and server at different times.

Packet diagrams in this document use the format defined in Section 1.3 of [QUIC-TRANSPORT] to illustrate the order and size of fields.

3. Connection Setup and Management

3.1. Draft Version Identification

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

HTTP/3 uses the token "h3" to identify itself in ALPN and Alt-Svc. Only implementations of the final, published RFC can identify themselves as "h3". Until such an RFC exists, implementations MUST NOT identify themselves using this string.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-quic-http-01 is identified using the string "h3-01".

Draft versions MUST use the corresponding draft transport version as their transport. For example, the application protocol defined in draft-ietf-quic-http-25 uses the transport defined in draft-ietf-quic-transport-25.

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation based on draft-ietf-quic-http-09 that reserves an extra stream for unsolicited transmission of 1980s pop music might identify itself as "h3-09-rickroll". Note that any label MUST conform to the "token" syntax defined in Section 5.7.2 of [SEMANTICS]. Experimenters are encouraged to coordinate their experiments on the quic@ietf.org mailing list.

3.2. Discovering an HTTP/3 Endpoint

HTTP relies on the notion of an authoritative response: a response that has been determined to be the most appropriate response for that request given the state of the target resource at the time of response message origination by (or at the direction of) the origin server identified within the target URI. Locating an authoritative server for an HTTP URL is discussed in Section 4.3 of [SEMANTICS].

The "https" scheme associates authority with possession of a certificate that the client considers to be trustworthy for the host identified by the authority component of the URL. If a server presents a certificate and proof that it controls the corresponding private key, then a client will accept a secured TLS session with that server as being authoritative for all origins with the "https" scheme and a host identified in the certificate.

A client MAY attempt access to a resource with an "https" URI by resolving the host identifier to an IP address, establishing a QUIC connection to that address on the indicated port, and sending an HTTP/3 request message targeting the URI to the server over that secured connection. Unless some other mechanism is used to select HTTP/3, the token "h3" is used in the Application Layer Protocol Negotiation (ALPN; see [RFC7301]) extension during the TLS handshake.

Connectivity problems (e.g., blocking UDP) can result in QUIC connection establishment failure; clients SHOULD attempt to use TCP-based versions of HTTP in this case.

Servers MAY serve HTTP/3 on any UDP port; an alternative service advertisement always includes an explicit port, and URLs contain either an explicit port or a default port associated with the scheme.

3.2.1. HTTP Alternative Services

An HTTP origin advertises the availability of an equivalent HTTP/3 endpoint via the Alt-Svc HTTP response header field or the HTTP/2 ALTSVC frame ([ALTSVC]), using the "h3" ALPN token.

For example, an origin could indicate in an HTTP response that HTTP/3 was available on UDP port 50781 at the same hostname by including the following header field:

```
Alt-Svc: h3=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/3 support, a client MAY attempt to establish a QUIC connection to the indicated host and port; if this connection is successful, the client can send HTTP requests using the mapping described in this document.

3.2.2. Other Schemes

Although HTTP is independent of the transport protocol, the "http" scheme associates authority with the ability to receive TCP connections on the indicated port of whatever host is identified within the authority component. Because HTTP/3 does not use TCP, HTTP/3 cannot be used for direct access to the authoritative server for a resource identified by an "http" URI. However, protocol extensions such as [ALTSVC] permit the authoritative server to identify other services that are also authoritative and that might be reachable over HTTP/3.

Prior to making requests for an origin whose scheme is not "https", the client MUST ensure the server is willing to serve that scheme. For origins whose scheme is "http", an experimental method to accomplish this is described in [RFC8164]. Other mechanisms might be defined for various schemes in the future.

3.3. Connection Establishment

HTTP/3 relies on QUIC version 1 as the underlying transport. The use of other QUIC transport versions with HTTP/3 MAY be defined by future specifications.

QUIC version 1 uses TLS version 1.3 or greater as its handshake protocol. HTTP/3 clients MUST support a mechanism to indicate the target host to the server during the TLS handshake. If the server is identified by a DNS name, clients MUST send the Server Name Indication (SNI; [RFC6066]) TLS extension unless an alternative mechanism to indicate the target host is used.

QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/3 support is indicated by selecting the ALPN token "h3" in the TLS handshake. Support for other application-layer protocols MAY be offered in the same handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP/3-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 7.2.4) MUST be sent by each endpoint as the initial frame of their respective HTTP control stream; see Section 6.2.1.

3.4. Connection Reuse

HTTP/3 connections are persistent across multiple requests. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. In general, a server is considered authoritative for all URIs with the "https" scheme for which the hostname in the URI is present in the authenticated certificate provided by the server, either as the CN field of the certificate subject or as a dNSName in the subjectAltName field of the certificate; see [RFC6125]. For a host that is an IP address, the client MUST verify that the address appears as an iPAddress in the subjectAltName field of the certificate. If the hostname or address is not present in the certificate, the client MUST NOT consider the server authoritative for origins containing that hostname or address. See Section 4.3 of [SEMANTICS] for more detail on authoritative access.

Clients SHOULD NOT open more than one HTTP/3 connection to a given host and port pair, where the host is derived from a URI, a selected alternative service ([ALTSVC]), or a configured proxy. A client MAY open multiple HTTP/3 connections to the same IP address and UDP port using different transport or TLS configurations but SHOULD avoid creating multiple connections with the same configuration.

Servers are encouraged to maintain open HTTP/3 connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the HTTP/3 connection, the terminating endpoint SHOULD first send a GOAWAY frame (Section 5.2) so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

A server that does not wish clients to reuse HTTP/3 connections for a particular origin can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request; see Section 9.1.2 of [HTTP2].

4. HTTP Request Lifecycle

4.1. HTTP Message Exchanges

A client sends an HTTP request on a client-initiated bidirectional QUIC stream. A client MUST send only a single request on a given stream. A server sends zero or more interim HTTP responses on the same stream as the request, followed by a single final HTTP response, as detailed below. See Section 14 of [SEMANTICS] for a description of interim and final HTTP responses.

Pushed responses are sent on a server-initiated unidirectional QUIC stream; see Section 6.2.2. A server sends zero or more interim HTTP responses, followed by a single final HTTP response, in the same manner as a standard response. Push is described in more detail in Section 4.4.

On a given stream, receipt of multiple requests or receipt of an additional HTTP response following a final HTTP response MUST be treated as malformed (Section 4.1.3).

An HTTP message (request or response) consists of:

1. the header field section, sent as a single HEADERS frame (see Section 7.2.2),

2. optionally, the payload body, if present, sent as a series of DATA frames (see Section 7.2.1), and

3. optionally, the trailer field section, if present, sent as a single HEADERS frame.

Header and trailer field sections are described in Sections 5.4 and 5.6 of [SEMANTICS]; the payload body is described in Section 5.5.4 of [SEMANTICS].

Receipt of an invalid sequence of frames MUST be treated as a connection error of type H3_FRAME_UNEXPECTED (Section 8). In particular, a DATA frame before any HEADERS frame, or a HEADERS or DATA frame after the trailing HEADERS frame is considered invalid. Other frame types, especially unknown frame types, might be permitted subject to their own rules; see Section 9.

A server MAY send one or more PUSH_PROMISE frames (Section 7.2.5) before, after, or interleaved with the frames of a response message. These PUSH_PROMISE frames are not part of the response; see Section 4.4 for more details. PUSH_PROMISE frames are not permitted on push streams; a pushed response that includes PUSH_PROMISE frames MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

Frames of unknown types (Section 9), including reserved frames (Section 7.2.8) MAY be sent on a request or push stream before, after, or interleaved with other frames described in this section.

The HEADERS and PUSH_PROMISE frames might reference updates to the QPACK dynamic table. While these updates are not directly part of the message exchange, they must be received and processed before the message can be consumed. See Section 4.1.1 for more details.

The "chunked" transfer encoding defined in Section 7.1 of [HTTP11] MUST NOT be used.

A response MAY consist of multiple messages when and only when one or more interim responses (1xx; see Section 14.2 of [SEMANTICS]) precede a final response to the same request. Interim responses do not contain a payload body or trailers.

An HTTP request/response exchange fully consumes a client-initiated bidirectional QUIC stream. After sending a request, a client MUST close the stream for sending. Unless using the CONNECT method (see Section 4.2), clients MUST NOT make stream closure dependent on receiving a response to their request. After sending a final response, the server MUST close the stream for sending. At this point, the QUIC stream is fully closed.

When a stream is closed, this indicates the end of the final HTTP message. Because some messages are large or unbounded, endpoints SHOULD begin processing partial HTTP messages once enough of the message has been received to make progress. If a client-initiated stream terminates without enough of the HTTP message to provide a complete response, the server SHOULD abort its response with the error code H3_REQUEST_INCOMPLETE.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When the server does not need to receive the remainder of the request, it MAY abort reading the request stream, send a complete response, and cleanly close the sending part of the stream. The error code H3_NO_ERROR SHOULD be used when requesting that the client stop sending on the request stream. Clients MUST NOT discard complete responses as a result of having their request terminated abruptly, though clients can always discard responses at their discretion for other reasons. If the server sends a partial or complete response but does not abort reading the request, clients SHOULD continue sending the body of the request and close the stream normally.

4.1.1. Field Formatting and Compression

HTTP messages carry metadata as a series of key-value pairs, called HTTP fields. For a listing of registered HTTP fields, see the "Hypertext Transfer Protocol (HTTP) Field Name Registry" maintained at https://www.iana.org/assignments/http-fields/.

> *Note:* This registry will not exist until [SEMANTICS] is approved. *RFC Editor*, please remove this note prior to publication.

As in previous versions of HTTP, field names are strings containing a subset of ASCII characters that are compared in a case-insensitive fashion. Properties of HTTP field names and values are discussed in more detail in Section 5.4.3 of [SEMANTICS]. As in HTTP/2, characters in field names MUST be converted to lowercase prior to their encoding. A request or response containing uppercase characters in field names MUST be treated as malformed (Section 4.1.3).

Like HTTP/2, HTTP/3 does not use the Connection header field to indicate connection-specific fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/3 field section containing connection-specific fields; any message containing connection-specific fields MUST be treated as malformed (Section 4.1.3).

The only exception to this is the TE header field, which MAY be present in an HTTP/3 request header; when it is, it MUST NOT contain any value other than "trailers".

This means that an intermediary transforming an HTTP/1.x message to HTTP/3 will need to remove any fields nominated by the Connection field, along with the Connection field itself. Such intermediaries

SHOULD also remove other connection-specific fields, such as Keep-Alive, Proxy-Connection, Transfer-Encoding, and Upgrade, even if they are not nominated by the Connection field.

4.1.1.1. Pseudo-Header Fields

Like HTTP/2, HTTP/3 employs a series of pseudo-header fields where the field name begins with the ':' character (ASCII 0x3a). These pseudo-header fields convey the target URI, the method of the request, and the status code for the response.

Pseudo-header fields are not HTTP fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document; however, an extension could negotiate a modification of this restriction; see Section 9.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailers. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 4.1.3).

All pseudo-header fields MUST appear in the header field section before regular header fields. Any request or response that contains a pseudo-header field that appears in a header field section after a regular header field MUST be treated as malformed (Section 4.1.3).

The following pseudo-header fields are defined for requests:

":method": Contains the HTTP method (Section 8 of [SEMANTICS])

":scheme": Contains the scheme portion of the target URI (Section 3.1 of [URI])

":scheme" is not restricted to "http" and "https" schemed URIs. A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

":authority": Contains the authority portion of the target URI (Section 3.2 of [URI]). The authority MUST NOT include the deprecated "userinfo" subcomponent for "http" or "https" schemed URIs.

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form; see Section 3.2 of [HTTP11]. Clients that generate HTTP/3 requests directly SHOULD use the ":authority" pseudo-header field instead of the Host field. An intermediary that converts an HTTP/3 request to HTTP/1.1 MUST create a Host field if one is not present in a request by copying the value of the ":authority" pseudo-header field.

":path": Contains the path and query parts of the target URI (the "path-absolute" production and optionally a '?' character followed by the "query" production; see Sections 3.3 and 3.4 of [URI]. A request in asterisk form includes the value '*' for the ":path" pseudo-header field.

This pseudo-header field MUST NOT be empty for "http" or "https" URIs; "http" or "https" URIs that do not contain a path component MUST include a value of '/'. The exception to this rule is an OPTIONS request for an "http" or "https" URI that does not include a path component; these MUST include a ":path" pseudo-header field with a value of '*'; see Section 3.2.4 of [HTTP11].

All HTTP/3 requests MUST include exactly one value for the ":method", ":scheme", and ":path" pseudo-header fields, unless it is a CONNECT request; see Section 4.2.

If the ":scheme" pseudo-header field identifies a scheme that has a mandatory authority component (including "http" and "https"), the request MUST contain either an ":authority" pseudo-header field or a "Host" header field. If these fields are present, they MUST NOT be empty. If both fields are present, they MUST contain the same value. If the scheme does not have a mandatory authority component and none is provided in the request target, the request MUST NOT contain the ":authority" pseudo-header or "Host" header fields.

An HTTP request that omits mandatory pseudo-header fields or contains invalid values for those pseudo-header fields is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

For responses, a single ":status" pseudo-header field is defined that carries the HTTP status code; see Section 14 of [SEMANTICS]. This pseudo-header field MUST be included in all responses; otherwise, the response is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

4.1.1.2. Field Compression

HTTP/3 uses QPACK field compression as described in [QPACK], a variation of HPACK that allows the flexibility to avoid compression-induced head-of-line blocking. See that document for additional details.

To allow for better compression efficiency, the "Cookie" field ([RFC6265]) MAY be split into separate field lines, each with one or more cookie-pairs, before compression. If a decompressed field section contains multiple cookie field lines, these MUST be concatenated into a single octet string using the two-octet delimiter of 0x3b, 0x20 (the ASCII string "; ") before being passed into a context other than HTTP/2 or HTTP/3, such as an HTTP/1.1 connection, or a generic HTTP server application.

4.1.1.3. Header Size Constraints

An HTTP/3 implementation MAY impose a limit on the maximum size of the message header it will accept on an individual HTTP message. A server that receives a larger header section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process. The size of a field list is calculated based on the uncompressed size of fields, including the length of the name and value in bytes plus an overhead of 32 bytes for each field.

If an implementation wishes to advise its peer of this limit, it can be conveyed as a number of bytes in the SETTINGS_MAX_FIELD_SECTION_SIZE parameter. An implementation that has received this parameter SHOULD NOT send an HTTP message header that exceeds the indicated size, as the peer will likely refuse to process it. However, because this limit is applied at each hop, messages below this limit are not guaranteed to be accepted.

4.1.2. Request Cancellation and Rejection

Once a request stream has been opened, the request MAY be cancelled by either endpoint. Clients cancel requests if the response is no longer of interest; servers cancel requests if they are unable to or choose not to respond. When possible, it is RECOMMENDED that servers send an HTTP response with an appropriate status code rather than canceling a request it has already begun processing.

Implementations SHOULD cancel requests by abruptly terminating any directions of a stream that are still open. This means resetting the sending parts of streams and aborting reading on receiving parts of streams; see Section 2.4 of [QUIC-TRANSPORT].

When the server cancels a request without performing any application processing, the request is considered "rejected." The server SHOULD abort its response stream with the error code H3_REQUEST_REJECTED. In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result. The client can treat requests rejected by the server as though they had never been sent at all, thereby allowing them to be retried later.

Servers MUST NOT use the H3_REQUEST_REJECTED error code for requests that were partially or fully processed. When a server abandons a response after partial processing, it SHOULD abort its response stream with the error code H3_REQUEST_CANCELLED.

Client SHOULD use the error code H3_REQUEST_CANCELLED to cancel requests. Upon receipt of this error code, a server MAY abruptly terminate the response using the error code H3_REQUEST_REJECTED if no processing was performed. Clients MUST NOT use the H3_REQUEST_REJECTED error code, except when a server has requested closure of the request stream with this error code.

If a stream is canceled after receiving a complete response, the client MAY ignore the cancellation and use the response. However, if a stream is cancelled after receiving a partial response, the response SHOULD NOT be used. Automatically retrying such requests is not possible, unless this is otherwise permitted (e.g., idempotent actions like GET, PUT, or DELETE).

4.1.3. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid sequence of frames but is invalid due to:

* the presence of prohibited fields or pseudo-header fields,

* the absence of mandatory pseudo-header fields,

* invalid values for pseudo-header fields,

* pseudo-header fields after fields,

* an invalid sequence of HTTP messages,

* the inclusion of uppercase field names, or

* the inclusion of invalid characters in field names or values

A request or response that includes a payload body can include a Content-Length header field. A request or response is also malformed if the value of a content-length header field does not equal the sum of the DATA frame payload lengths that form the body. A response that is defined to have no payload, as described in Section 5.5.4 of [SEMANTICS], can have a non-zero content-length field, even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error (Section 8) of type H3_GENERAL_PROTOCOL_ERROR.

For malformed requests, a server MAY send an HTTP response indicating the error prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

4.2. The CONNECT Method

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target (Section 3.2 of [HTTP11]). It is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources.

In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2 and HTTP/3, the CONNECT method is used to establish a tunnel over a single stream.

A CONNECT request MUST be constructed as follows:

* The ":method" pseudo-header field is set to "CONNECT"

* The ":scheme" and ":path" pseudo-header fields are omitted

* The ":authority" pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests; see Section 3.2.3 of [HTTP11])

The request stream remains open at the end of the request to carry the data to be transferred. A CONNECT request that does not conform to these restrictions is malformed; see Section 4.1.3.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in Section 14.3 of [SEMANTICS].

All DATA frames on the stream correspond to data sent or received on the TCP connection. The payload of any DATA frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is packaged into DATA frames by the proxy. Note that the size and number of TCP segments is not guaranteed to map predictably to the size and number of HTTP DATA or QUIC STREAM frames.

Once the CONNECT method has completed, only DATA frames are permitted to be sent on the stream. Extension frames MAY be used if specifically permitted by the definition of the extension. Receipt of any other known frame type MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

The TCP connection can be closed by either peer. When the client ends the request stream (that is, the receive stream at the proxy enters the "Data Recvd" state), the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will close the send stream that it sends to the client. TCP connections that remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT close a stream for sending while they still expect to receive data from the target of the CONNECT.

A TCP connection error is signaled by abruptly terminating the stream. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type H3_CONNECT_ERROR (Section 8.1). Correspondingly, if a proxy detects an error with the stream or the QUIC connection, it MUST close the TCP connection. If the underlying TCP implementation permits it, the proxy SHOULD send a TCP segment with the RST bit set.

4.3. HTTP Upgrade

HTTP/3 does not support the HTTP Upgrade mechanism (Section 6.6 of [SEMANTICS]) or 101 (Switching Protocols) informational status code (Section 14.2.2 of [SEMANTICS]).

4.4. Server Push

Server push is an interaction mode that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. HTTP/3 server push is similar to what is described in Section 8.2 of [HTTP2], but uses different mechanisms.

Each server push is assigned a unique Push ID by the server. The Push ID is used to refer to the push in various contexts throughout the lifetime of the HTTP/3 connection.

The Push ID space begins at zero, and ends at a maximum value set by the MAX_PUSH_ID frame; see Section 7.2.7. In particular, a server is not able to push until after the client sends a MAX_PUSH_ID frame. A client sends MAX_PUSH_ID frames to control the number of pushes that a server can promise. A server SHOULD use Push IDs sequentially, beginning from zero. A client MUST treat receipt of a push stream as a connection error of type H3_ID_ERROR when no MAX_PUSH_ID frame has been sent or when the stream references a Push ID that is greater than the maximum Push ID.

The Push ID is used in one or more PUSH_PROMISE frames (Section 7.2.5) that carry the header section of the request message. These frames are sent on the request stream that generated the push. This allows the server push to be associated with a client request. When the same Push ID is promised on multiple request streams, the decompressed request field sections MUST contain the same fields in the same order, and both the name and the value in each field MUST be identical.

The Push ID is then included with the push stream that ultimately fulfills those promises; see Section 6.2.2. The push stream identifies the Push ID of the promise that it fulfills, then contains a response to the promised request as described in Section 4.1.

Finally, the Push ID can be used in CANCEL_PUSH frames; see Section 7.2.3. Clients use this frame to indicate they do not wish to receive a promised resource. Servers use this frame to indicate they will not be fulfilling a previous promise.

Not all requests can be pushed. A server MAY push requests that have the following properties:

* cacheable; see Section 8.2.3 of [SEMANTICS]

* safe; see Section 8.2.1 of [SEMANTICS]

* does not include a request body or trailer section

The server MUST include a value in the ":authority" pseudo-header field for which the server is authoritative; see Section 3.4.

Clients SHOULD send a CANCEL_PUSH frame upon receipt of a PUSH_PROMISE frame carrying a request that is not cacheable, is not known to be safe, that indicates the presence of a request body, or for which it does not consider the server authoritative. Any corresponding responses MUST NOT be used or cached.

Each pushed response is associated with one or more client requests. The push is associated with the request stream on which the PUSH_PROMISE frame was received. The same server push can be associated with additional client requests using a PUSH_PROMISE frame with the same Push ID on multiple request streams. These associations do not affect the operation of the protocol, but MAY be considered by user agents when deciding how to use pushed resources.

Ordering of a PUSH_PROMISE frame in relation to certain parts of the response is important. The server SHOULD send PUSH_PROMISE frames prior to sending HEADERS or DATA frames that reference the promised responses. This reduces the chance that a client requests a resource that will be pushed by the server.

Due to reordering, push stream data can arrive before the corresponding PUSH_PROMISE frame. When a client receives a new push stream with an as-yet-unknown Push ID, both the associated client request and the pushed request header fields are unknown. The client can buffer the stream data in expectation of the matching PUSH_PROMISE. The client can use stream flow control (see section 4.1 of [QUIC-TRANSPORT]) to limit the amount of data a server may commit to the pushed stream.

Push stream data can also arrive after a client has canceled a push. In this case, the client can abort reading the stream with an error code of H3_REQUEST_CANCELLED. This asks the server not to transfer additional data and indicates that it will be discarded upon receipt.

Pushed responses that are cacheable (see Section 3 of [CACHING]) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present; see Section 5.2.2.3 of [CACHING]) at the time the pushed response is received.

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

5. Connection Closure

Once established, an HTTP/3 connection can be used for many requests and responses over time until the connection is closed. Connection closure can happen in any of several different ways.

5.1. Idle Connections

Each QUIC endpoint declares an idle timeout during the handshake. If the QUIC connection remains idle (no packets received) for longer than this duration, the peer will assume that the connection has been closed. HTTP/3 implementations will need to open a new HTTP/3 connection for new requests if the existing connection has been idle for longer than the server's advertised idle timeout, and SHOULD do so if approaching the idle timeout.

HTTP clients are expected to request that the transport keep connections open while there are responses outstanding for requests or server pushes, as described in Section 10.1.2 of [QUIC-TRANSPORT]. If the client is not expecting a response from the server, allowing an idle connection to time out is preferred over expending effort maintaining a connection that might not be needed. A gateway MAY maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers. Servers SHOULD NOT actively keep connections open.

5.2. Connection Shutdown

Even when a connection is not idle, either endpoint can decide to stop using the connection and initiate a graceful connection close. Endpoints initiate the graceful shutdown of an HTTP/3 connection by sending a GOAWAY frame (Section 7.2.6). The GOAWAY frame contains an identifier that indicates to the receiver the range of requests or pushes that were or might be processed in this connection. The server sends a client-initiated bidirectional Stream ID; the client sends a Push ID (Section 4.4). Requests or pushes with the indicated identifier or greater are rejected (Section 4.1.2) by the sender of the GOAWAY. This identifier MAY be zero if no requests or pushes were processed.

The information in the GOAWAY frame enables a client and server to agree on which requests or pushes were accepted prior to the shutdown of the HTTP/3 connection. Upon sending a GOAWAY frame, the endpoint SHOULD explicitly cancel (see Section 4.1.2 and Section 7.2.3) any requests or pushes that have identifiers greater than or equal to that indicated, in order to clean up transport state for the affected streams. The endpoint SHOULD continue to do so as more requests or pushes arrive.

Endpoints MUST NOT initiate new requests or promise new pushes on the connection after receipt of a GOAWAY frame from the peer. Clients MAY establish a new connection to send additional requests.

Some requests or pushes might already be in transit:

* Upon receipt of a GOAWAY frame, if the client has already sent requests with a Stream ID greater than or equal to the identifier contained in the GOAWAY frame, those requests will not be processed. Clients can safely retry unprocessed requests on a different HTTP connection. A client that is unable to retry requests loses all requests that are in flight when the server closes the connection.

  Requests on Stream IDs less than the Stream ID in a GOAWAY frame from the server might have been processed; their status cannot be known until a response is received, the stream is reset individually, another GOAWAY is received, or the connection terminates.

  Servers MAY reject individual requests on streams below the indicated ID if these requests were not processed.

* If a server receives a GOAWAY frame after having promised pushes with a Push ID greater than or equal to the identifier contained in the GOAWAY frame, those pushes will not be accepted.

Servers SHOULD send a GOAWAY frame when the closing of a connection is known in advance, even if the advance notice is small, so that the remote peer can know whether a request has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a QUIC connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

An endpoint MAY send multiple GOAWAY frames indicating different identifiers, but the identifier in each frame MUST NOT be greater than the identifier in any previous frame, since clients might already have retried unprocessed requests on another HTTP connection. Receiving a GOAWAY containing a larger identifier than previously received MUST be treated as a connection error of type H3_ID_ERROR.

An endpoint that is attempting to gracefully shut down a connection can send a GOAWAY frame with a value set to the maximum possible value (2^62-4 for servers, 2^62-1 for clients). This ensures that the peer stops creating new requests or pushes. After allowing time for any in-flight requests or pushes to arrive, the endpoint can send another GOAWAY frame indicating which requests or pushes it might accept before the end of the connection. This ensures that a connection can be cleanly shut down without losing requests.

A client has more flexibility in the value it chooses for the Push ID in a GOAWAY that it sends. A value of 2^62 - 1 indicates that the server can continue fulfilling pushes that have already been promised. A smaller value indicates the client will reject pushes with Push IDs greater than or equal to this value. Like the server, the client MAY send subsequent GOAWAY frames so long as the specified Push ID is no greater than any previously sent value.

Even when a GOAWAY indicates that a given request or push will not be processed or accepted upon receipt, the underlying transport resources still exist. The endpoint that initiated these requests can cancel them to clean up transport state.

Once all accepted requests and pushes have been processed, the endpoint can permit the connection to become idle, or MAY initiate an immediate closure of the connection. An endpoint that completes a graceful shutdown SHOULD use the H3_NO_ERROR error code when closing the connection.

If a client has consumed all available bidirectional stream IDs with requests, the server need not send a GOAWAY frame, since the client is unable to make further requests.

5.3. Immediate Application Closure

An HTTP/3 implementation can immediately close the QUIC connection at any time. This results in sending a QUIC CONNECTION_CLOSE frame to the peer indicating that the application layer has terminated the connection. The application error code in this frame indicates to the peer why the connection is being closed. See Section 8 for error codes that can be used when closing a connection in HTTP/3.

Before closing the connection, a GOAWAY frame MAY be sent to allow the client to retry some requests. Including the GOAWAY frame in the same packet as the QUIC CONNECTION_CLOSE frame improves the chances of the frame being received by clients.

5.4. Transport Closure

For various reasons, the QUIC transport could indicate to the application layer that the connection has terminated. This might be due to an explicit closure by the peer, a transport-level error, or a change in network topology that interrupts connectivity.

If a connection terminates without a GOAWAY frame, clients MUST assume that any request that was sent, whether in whole or in part, might have been processed.

6. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. The transport layer buffers and orders received QUIC STREAM frames, exposing the data contained within as a reliable byte stream to the application. Although QUIC permits out-of-order delivery within a stream, HTTP/3 does not make use of this feature.

QUIC streams can be either unidirectional, carrying data only from initiator to receiver, or bidirectional. Streams can be initiated by either the client or the server. For more detail on QUIC streams, see Section 2 of [QUIC-TRANSPORT].

When HTTP fields and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction or to the entire HTTP/3 connection context.

6.1. Bidirectional Streams

All client-initiated bidirectional streams are used for HTTP requests and responses. A bidirectional stream ensures that the response can be readily correlated with the request. This means that the client's first request occurs on QUIC stream 0, with subsequent requests on stream 4, 8, and so on. In order to permit these streams to open, an HTTP/3 server SHOULD configure non-zero minimum values for the number of permitted streams and the initial stream flow control window. So as to not unnecessarily limit parallelism, at least 100 requests SHOULD be permitted at a time.

HTTP/3 does not use server-initiated bidirectional streams, though an extension could define a use for these streams. Clients MUST treat receipt of a server-initiated bidirectional stream as a connection error of type H3_STREAM_CREATION_ERROR unless such an extension has been negotiated.

6.2. Unidirectional Streams

Unidirectional streams, in either direction, are used for a range of purposes. The purpose is indicated by a stream type, which is sent as a variable-length integer at the start of the stream. The format and structure of data that follows this integer is determined by the stream type.

```
Unidirectional Stream Header {
  Stream Type (i),
}
```

Figure 1: Unidirectional Stream Header

Two stream types are defined in this document: control streams (Section 6.2.1) and push streams (Section 6.2.2). [QPACK] defines two additional stream types. Other stream types can be defined by extensions to HTTP/3; see Section 9 for more details. Some stream types are reserved (Section 6.2.3).

The performance of HTTP/3 connections in the early phase of their lifetime is sensitive to the creation and exchange of data on unidirectional streams. Endpoints that excessively restrict the number of streams or the flow control window of these streams will increase the chance that the remote peer reaches the limit early and becomes blocked. In particular, implementations should consider that remote peers may wish to exercise reserved stream behavior (Section 6.2.3) with some of the unidirectional streams they are permitted to use. To avoid blocking, the transport parameters sent by both clients and servers MUST allow the peer to create at least one unidirectional stream for the HTTP control stream plus the number of unidirectional streams required by mandatory extensions (three being the minimum number required for the base HTTP/3 protocol and QPACK), and SHOULD provide at least 1,024 bytes of flow control credit to each stream.

Note that an endpoint is not required to grant additional credits to create more unidirectional streams if its peer consumes all the initial credits before creating the critical unidirectional streams. Endpoints SHOULD create the HTTP control stream as well as the unidirectional streams required by mandatory extensions (such as the QPACK encoder and decoder streams) first, and then create additional streams as allowed by their peer.

If the stream header indicates a stream type that is not supported by the recipient, the remainder of the stream cannot be consumed as the semantics are unknown. Recipients of unknown stream types MAY abort reading of the stream with an error code of H3_STREAM_CREATION_ERROR or a reserved error code (Section 8.1), but MUST NOT consider such streams to be a connection error of any kind.

Implementations MAY send stream types before knowing whether the peer supports them. However, stream types that could modify the state or semantics of existing protocol components, including QPACK or other extensions, MUST NOT be sent until the peer is known to support them.

A sender can close or reset a unidirectional stream unless otherwise specified. A receiver MUST tolerate unidirectional streams being closed or reset prior to the reception of the unidirectional stream header.

6.2.1. Control Streams

A control stream is indicated by a stream type of 0x00. Data on this stream consists of HTTP/3 frames, as defined in Section 7.2.

Each side MUST initiate a single control stream at the beginning of the connection and send its SETTINGS frame as the first frame on this stream. If the first frame of the control stream is any other frame type, this MUST be treated as a connection error of type H3_MISSING_SETTINGS. Only one control stream per peer is permitted; receipt of a second stream claiming to be a control stream MUST be treated as a connection error of type H3_STREAM_CREATION_ERROR. The sender MUST NOT close the control stream, and the receiver MUST NOT request that the sender close the control stream. If either control stream is closed at any point, this MUST be treated as a connection error of type H3_CLOSED_CRITICAL_STREAM.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon as it is able. Depending on whether 0-RTT is enabled on the QUIC connection, either client or server might be able to send stream data first after the cryptographic handshake completes.

6.2.2. Push Streams

Server push is an optional feature introduced in HTTP/2 that allows a server to initiate a response before a request has been made. See Section 4.4 for more details.

A push stream is indicated by a stream type of 0x01, followed by the Push ID of the promise that it fulfills, encoded as a variable-length integer. The remaining data on this stream consists of HTTP/3 frames, as defined in Section 7.2, and fulfills a promised server push by zero or more interim HTTP responses followed by a single final HTTP response, as defined in Section 4.1. Server push and Push IDs are described in Section 4.4.

Only servers can push; if a server receives a client-initiated push stream, this MUST be treated as a connection error of type H3_STREAM_CREATION_ERROR.

```
Push Stream Header {
  Stream Type (i) = 0x01,
  Push ID (i),
}
```

Figure 2: Push Stream Header

Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type H3_ID_ERROR.

6.2.3. Reserved Stream Types

Stream types of the format "0x1f * N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored. These streams have no semantics, and can be sent when application-layer padding is desired. They MAY also be sent on connections where no data is currently being transferred. Endpoints MUST NOT consider these streams to have any meaning upon receipt.

The payload and length of the stream are selected in any manner the sending implementation chooses. When sending a reserved stream type, the implementation MAY either terminate the stream cleanly or reset it. When resetting the stream, either the H3_NO_ERROR error code or a reserved error code (Section 8.1) SHOULD be used.

7. HTTP Framing Layer

HTTP frames are carried on QUIC streams, as described in Section 6. HTTP/3 defines three stream types: control stream, request stream, and push stream. This section describes HTTP/3 frame formats and their permitted stream types; see Table 1 for an overview. A comparison between HTTP/2 and HTTP/3 frames is provided in Appendix A.2.

Frame	Control Stream	Request Stream	Push Stream	Section
DATA	No	Yes	Yes	Section 7.2.1
HEADERS	No	Yes	Yes	Section 7.2.2
CANCEL_PUSH	Yes	No	No	Section 7.2.3
SETTINGS	Yes (1)	No	No	Section 7.2.4
PUSH_PROMISE	No	Yes	No	Section 7.2.5
GOAWAY	Yes	No	No	Section 7.2.6
MAX_PUSH_ID	Yes	No	No	Section 7.2.7
Reserved	Yes	Yes	Yes	Section 7.2.8

Table 1: HTTP/3 Frames and Stream Type Overview

Certain frames can only occur as the first frame of a particular stream type; these are indicated in Table 1 with a (1). Specific guidance is provided in the relevant section.

Note that, unlike QUIC frames, HTTP/3 frames can span multiple packets.

7.1. Frame Layout

All frames have the following format:

```
HTTP/3 Frame Format {
  Type (i),
  Length (i),
  Frame Payload (..),
}
```

Figure 3: HTTP/3 Frame Format

A frame includes the following fields:

Type: A variable-length integer that identifies the frame type.

Length: A variable-length integer that describes the length in bytes of the Frame Payload.

Frame Payload: A payload, the semantics of which are determined by the Type field.

Each frame's payload MUST contain exactly the fields identified in its description. A frame payload that contains additional bytes after the identified fields or a frame payload that terminates before the end of the identified fields MUST be treated as a connection error (Section 8) of type H3_FRAME_ERROR.

When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error (Section 8) of type H3_FRAME_ERROR. Streams that terminate abruptly may be reset at any point in a frame.

7.2. Frame Definitions

7.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of bytes associated with an HTTP request or response payload body.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on a control stream, the recipient MUST respond with a connection error (Section 8) of type H3_FRAME_UNEXPECTED.

```
DATA Frame {
  Type (i) = 0x0,
  Length (i),
  Data (..),
}
```

Figure 4: DATA Frame

7.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry an HTTP field section, encoded using QPACK. See [QPACK] for more details.

```
HEADERS Frame {
  Type (i) = 0x1,
  Length (i),
  Encoded Field Section (..),
}
```

Figure 5: HEADERS Frame

HEADERS frames can only be sent on request or push streams. If a HEADERS frame is received on a control stream, the recipient MUST respond with a connection error (Section 8) of type H3_FRAME_UNEXPECTED.

7.2.3. CANCEL_PUSH

The CANCEL_PUSH frame (type=0x3) is used to request cancellation of a server push prior to the push stream being received. The CANCEL_PUSH frame identifies a server push by Push ID (see Section 4.4), encoded as a variable-length integer.

When a client sends CANCEL_PUSH, it is indicating that it does not wish to receive the promised resource. The server SHOULD abort sending the resource, but the mechanism to do so depends on the state of the corresponding push stream. If the server has not yet created a push stream, it does not create one. If the push stream is open, the server SHOULD abruptly terminate that stream. If the push stream has already ended, the server MAY still abruptly terminate the stream or MAY take no action.

A server sends CANCEL_PUSH to indicate that it will not be fulfilling a promise which was previously sent. The client cannot expect the corresponding promise to be fulfilled, unless it has already received and processed the promised response. Regardless of whether a push stream has been opened, a server SHOULD send a CANCEL_PUSH frame when it determines that promise will not be fulfilled. If a stream has already been opened, the server can abort sending on the stream with an error code of H3_REQUEST_CANCELLED.

Sending a CANCEL_PUSH frame has no direct effect on the state of existing push streams. A client SHOULD NOT send a CANCEL_PUSH frame when it has already received a corresponding push stream. A push stream could arrive after a client has sent a CANCEL_PUSH frame, because a server might not have processed the CANCEL_PUSH. The client SHOULD abort reading the stream with an error code of H3_REQUEST_CANCELLED.

A CANCEL_PUSH frame is sent on the control stream. Receiving a CANCEL_PUSH frame on a stream other than the control stream MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

```
CANCEL_PUSH Frame {
  Type (i) = 0x3,
  Length (i),
  Push ID (..),
}
```

Figure 6: CANCEL_PUSH Frame

The CANCEL_PUSH frame carries a Push ID encoded as a variable-length integer. The Push ID identifies the server push that is being cancelled; see Section 4.4. If a CANCEL_PUSH frame is received that references a Push ID greater than currently allowed on the connection, this MUST be treated as a connection error of type H3_ID_ERROR.

If the client receives a CANCEL_PUSH frame, that frame might identify a Push ID that has not yet been mentioned by a PUSH_PROMISE frame due to reordering. If a server receives a CANCEL_PUSH frame for a Push ID that has not yet been mentioned by a PUSH_PROMISE frame, this MUST be treated as a connection error of type H3_ID_ERROR.

7.2.4. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. Individually, a SETTINGS parameter can also be referred to as a "setting"; the identifier and value of each setting parameter can be referred to as a "setting identifier" and a "setting value".

SETTINGS frames always apply to an entire HTTP/3 connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of each control stream (see Section 6.2.1) by each peer, and MUST NOT be sent subsequently. If an endpoint receives a second SETTINGS frame on the control stream, the endpoint MUST respond with a connection error of type H3_FRAME_UNEXPECTED.

SETTINGS frames MUST NOT be sent on any stream other than the control stream. If an endpoint receives a SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type H3_FRAME_UNEXPECTED.

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer that can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - each peer uses SETTINGS to advertise a set of supported values. The definition of the setting would describe how each peer combines the two sets to conclude which choice will be used. SETTINGS does not provide a mechanism to identify when the choice takes effect.

Different values for the same parameter can be advertised by each peer. For example, a client might be willing to consume a very large response field section, while servers are more cautious about request size.

The same setting identifier MUST NOT occur more than once in the SETTINGS frame. A receiver MAY treat the presence of duplicate setting identifiers as a connection error of type H3_SETTINGS_ERROR.

The payload of a SETTINGS frame consists of zero or more parameters. Each parameter consists of a setting identifier and a value, both encoded as QUIC variable-length integers.

```
Setting {
  Identifier (i),
  Value (i),
}

SETTINGS Frame {
  Type (i) = 0x4,
  Length (i),
  Setting (..) ...,
}
```

Figure 7: SETTINGS Frame

An implementation MUST ignore the contents for any SETTINGS identifier it does not understand.

7.2.4.1. Defined SETTINGS Parameters

The following settings are defined in HTTP/3:

SETTINGS_MAX_FIELD_SECTION_SIZE (0x6): The default value is unlimited. See Section 4.1.1.3 for usage.

Setting identifiers of the format "0x1f * N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown identifiers be ignored. Such settings have no defined meaning. Endpoints SHOULD include at least one such setting in their SETTINGS frame. Endpoints MUST NOT consider such settings to have any meaning upon receipt.

Because the setting has no defined meaning, the value of the setting can be any value the implementation selects.

Setting identifiers which were used in HTTP/2 where there is no corresponding HTTP/3 setting have also been reserved (Section 11.2.2). These settings MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3_SETTINGS_ERROR.

Additional settings can be defined by extensions to HTTP/3; see Section 9 for more details.

7.2.4.2. Initialization

An HTTP implementation MUST NOT send frames or requests that would be invalid based on its current understanding of the peer’s settings.

All settings begin at an initial value. Each endpoint SHOULD use these initial values to send messages before the peer's SETTINGS frame has arrived, as packets carrying the settings can be lost or delayed. When the SETTINGS frame arrives, any settings are changed to their new values.

This removes the need to wait for the SETTINGS frame before sending messages. Endpoints MUST NOT require any data to be received from the peer prior to sending the SETTINGS frame; settings MUST be sent as soon as the transport is ready to send data.

For servers, the initial value of each client setting is the default value.

For clients using a 1-RTT QUIC connection, the initial value of each server setting is the default value. 1-RTT keys will always become available prior to the packet containing SETTINGS being processed by QUIC, even if the server sends SETTINGS immediately. Clients SHOULD NOT wait indefinitely for SETTINGS to arrive before sending requests, but SHOULD process received datagrams in order to increase the likelihood of processing SETTINGS before sending the first request.

When a 0-RTT QUIC connection is being used, the initial value of each server setting is the value used in the previous session. Clients SHOULD store the settings the server provided in the HTTP/3 connection where resumption information was provided, but MAY opt not to store settings in certain cases (e.g., if the session ticket is received before the SETTINGS frame). A client MUST comply with stored settings - or default values, if no values are stored - when attempting 0-RTT. Once a server has provided new settings, clients MUST comply with those values.

A server can remember the settings that it advertised, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the HTTP/3 settings values in determining whether to accept 0-RTT data. If the server cannot determine that the settings remembered by a client are compatible with its current settings, it MUST NOT accept 0-RTT data. Remembered settings are compatible if a client complying with those settings would not violate the server's current settings.

A server MAY accept 0-RTT and subsequently provide different settings in its SETTINGS frame. If 0-RTT data is accepted by the server, its SETTINGS frame MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. The server MUST include all settings that differ from their default values. If a server accepts 0-RTT but then sends settings that are not compatible with the previously specified settings, this MUST be treated as a connection error of type H3_SETTINGS_ERROR. If a server accepts 0-RTT but then sends a SETTINGS frame that omits a setting value that the client understands (apart from reserved setting identifiers) that was previously specified to have a non-default value, this MUST be treated as a connection error of type H3_SETTINGS_ERROR.

7.2.5. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x5) is used to carry a promised request header field section from server to client on a request stream, as in HTTP/2.

```
PUSH_PROMISE Frame {
  Type (i) = 0x5,
  Length (i),
  Push ID (i),
  Encoded Field Section (..),
}
```

Figure 8: PUSH_PROMISE Frame

The payload consists of:

Push ID: A variable-length integer that identifies the server push operation. A Push ID is used in push stream headers (Section 4.4) and CANCEL_PUSH frames (Section 7.2.3).

Encoded Field Section: QPACK-encoded request header fields for the promised response. See [QPACK] for more details.

A server MUST NOT use a Push ID that is larger than the client has provided in a MAX_PUSH_ID frame (Section 7.2.7). A client MUST treat receipt of a PUSH_PROMISE frame that contains a larger Push ID than the client has advertised as a connection error of H3_ID_ERROR.

A server MAY use the same Push ID in multiple PUSH_PROMISE frames. If so, the decompressed request header sets MUST contain the same fields in the same order, and both the name and the value in each field MUST be exact matches. Clients SHOULD compare the request header sections for resources promised multiple times. If a client receives a Push ID that has already been promised and detects a mismatch, it MUST respond with a connection error of type H3_GENERAL_PROTOCOL_ERROR. If the decompressed field sections match exactly, the client SHOULD associate the pushed content with each stream on which a PUSH_PROMISE frame was received.

Allowing duplicate references to the same Push ID is primarily to reduce duplication caused by concurrent requests. A server SHOULD avoid reusing a Push ID over a long period. Clients are likely to consume server push responses and not retain them for reuse over time. Clients that see a PUSH_PROMISE frame that uses a Push ID that they have already consumed and discarded are forced to ignore the promise.

If a PUSH_PROMISE frame is received on the control stream, the client MUST respond with a connection error (Section 8) of type H3_FRAME_UNEXPECTED.

A client MUST NOT send a PUSH_PROMISE frame. A server MUST treat the receipt of a PUSH_PROMISE frame as a connection error of type H3_FRAME_UNEXPECTED.

See Section 4.4 for a description of the overall server push mechanism.

7.2.6. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of an HTTP/3 connection by either endpoint. GOAWAY allows an endpoint to stop accepting new requests or pushes while still finishing processing of previously received requests and pushes. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

```
GOAWAY Frame {
  Type (i) = 0x7,
  Length (i),
  Stream ID/Push ID (..),
}
```

Figure 9: GOAWAY Frame

The GOAWAY frame is always sent on the control stream. In the server to client direction, it carries a QUIC Stream ID for a client-initiated bidirectional stream encoded as a variable-length integer. A client MUST treat receipt of a GOAWAY frame containing a Stream ID of any other type as a connection error of type H3_ID_ERROR.

In the client to server direction, the GOAWAY frame carries a Push ID encoded as a variable-length integer.

The GOAWAY frame applies to the entire connection, not a specific stream. A client MUST treat a GOAWAY frame on a stream other than the control stream as a connection error (Section 8) of type H3_FRAME_UNEXPECTED.

See Section 5.2 for more information on the use of the GOAWAY frame.

7.2.7. MAX_PUSH_ID

The MAX_PUSH_ID frame (type=0xd) is used by clients to control the number of server pushes that the server can initiate. This sets the maximum value for a Push ID that the server can use in PUSH_PROMISE and CANCEL_PUSH frames. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit maintained by the QUIC transport.

The MAX_PUSH_ID frame is always sent on the control stream. Receipt of a MAX_PUSH_ID frame on any other stream MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

A server MUST NOT send a MAX_PUSH_ID frame. A client MUST treat the receipt of a MAX_PUSH_ID frame as a connection error of type H3_FRAME_UNEXPECTED.

The maximum Push ID is unset when an HTTP/3 connection is created, meaning that a server cannot push until it receives a MAX_PUSH_ID frame. A client that wishes to manage the number of promised server pushes can increase the maximum Push ID by sending MAX_PUSH_ID frames as the server fulfills or cancels server pushes.

```
MAX_PUSH_ID Frame {
  Type (i) = 0xd,
  Length (i),
  Push ID (i),
}
```

Figure 10: MAX_PUSH_ID Frame

The MAX_PUSH_ID frame carries a single variable-length integer that identifies the maximum value for a Push ID that the server can use; see Section 4.4. A MAX_PUSH_ID frame cannot reduce the maximum Push ID; receipt of a MAX_PUSH_ID frame that contains a smaller value than previously received MUST be treated as a connection error of type H3_ID_ERROR.

7.2.8. Reserved Frame Types

Frame types of the format "0x1f * N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored (Section 9). These frames have no semantics, and MAY be sent on any stream where frames are allowed to be sent. This enables their use for application-layer padding. Endpoints MUST NOT consider these frames to have any meaning upon receipt.

The payload and length of the frames are selected in any manner the implementation chooses.

Frame types that were used in HTTP/2 where there is no corresponding HTTP/3 frame have also been reserved (Section 11.2.1). These frame types MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

8. Error Handling

QUIC allows the application to abruptly terminate (reset) individual streams or the entire connection; see Sections 2.4 and 5.3 of [QUIC-TRANSPORT]. These are referred to as "stream errors" or "connection errors" (see Section 11 of [QUIC-TRANSPORT]) and have associated error codes, but do not necessarily indicate a problem with the connection or either implementation. For example, a stream can be reset if the requested resource is no longer needed.

An endpoint MAY choose to treat a stream error as a connection error under certain circumstances. Implementations need to consider the impact on outstanding requests before making this choice.

Because new error codes can be defined without negotiation (see Section 9), use of an error code in an unexpected context or receipt of an unknown error code MUST be treated as equivalent to H3_NO_ERROR. However, closing a stream can have other effects regardless of the error code; for example, see Section 4.1.

This section describes HTTP/3-specific error codes that can be used to express the cause of a connection or stream error.

8.1. HTTP/3 Error Codes

The following error codes are defined for use when abruptly terminating streams, aborting reading of streams, or immediately closing HTTP/3 connections.

H3_NO_ERROR (0x100): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

H3_GENERAL_PROTOCOL_ERROR (0x101): Peer violated protocol requirements in a way that does not match a more specific error code, or endpoint declines to use the more specific error code.

H3_INTERNAL_ERROR (0x102): An internal error has occurred in the HTTP stack.

H3_STREAM_CREATION_ERROR (0x103): The endpoint detected that its peer created a stream that it will not accept.

H3_CLOSED_CRITICAL_STREAM (0x104): A stream required by the HTTP/3 connection was closed or reset.

H3_FRAME_UNEXPECTED (0x105): A frame was received that was not permitted in the current state or on the current stream.

H3_FRAME_ERROR (0x106): A frame that fails to satisfy layout requirements or with an invalid size was received.

H3_EXCESSIVE_LOAD (0x107): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

H3_ID_ERROR (0x108): A Stream ID or Push ID was used incorrectly, such as exceeding a limit, reducing a limit, or being reused.

H3_SETTINGS_ERROR (0x109): An endpoint detected an error in the payload of a SETTINGS frame.

H3_MISSING_SETTINGS (0x10a): No SETTINGS frame was received at the beginning of the control stream.

H3_REQUEST_REJECTED (0x10b): A server rejected a request without performing any application processing.

H3_REQUEST_CANCELLED (0x10c): The request or its response (including pushed response) is cancelled.

H3_REQUEST_INCOMPLETE (0x10d): The client's stream terminated without containing a fully-formed request.

H3_CONNECT_ERROR (0x10f): The TCP connection established in response to a CONNECT request was reset or abnormally closed.

H3_VERSION_FALLBACK (0x110): The requested operation cannot be served over HTTP/3. The peer should retry over HTTP/1.1.

Error codes of the format "0x1f * N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown error codes be treated as equivalent to H3_NO_ERROR (Section 9). Implementations SHOULD select an error code from this space with some probability when they would have sent H3_NO_ERROR.

9. Extensions to HTTP/3

HTTP/3 permits extension of the protocol. Within the limitations described in this section, protocol extensions can be used to provide additional services or alter any aspect of the protocol. Extensions are effective only within the scope of a single HTTP/3 connection.

This applies to the protocol elements defined in this document. This does not affect the existing options for extending HTTP, such as defining new methods, status codes, or fields.

Extensions are permitted to use new frame types (Section 7.2), new settings (Section 7.2.4.1), new error codes (Section 8), or new unidirectional stream types (Section 6.2). Registries are established for managing these extension points: frame types (Section 11.2.1), settings (Section 11.2.2), error codes (Section 11.2.3), and stream types (Section 11.2.4).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames and unidirectional streams that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation. However, where a known frame type is required to be in a specific location, such as the SETTINGS frame as the first frame of the control stream (see Section 6.2.1), an unknown frame type does not satisfy that requirement and SHOULD be treated as an error.

Extensions that could change the semantics of existing protocol components MUST be negotiated before being used. For example, an extension that changes the layout of the HEADERS frame cannot be used until the peer has given a positive signal that this is acceptable. Coordinating when such a revised layout comes into effect could prove complex. As such, allocating new identifiers for new definitions of existing protocol elements is likely to be more effective.

This document does not mandate a specific method for negotiating the use of an extension but notes that a setting (Section 7.2.4.1) could be used for that purpose. If both peers set a value that indicates willingness to use the extension, then the extension can be used. If a setting is used for extension negotiation, the default value MUST be defined in such a fashion that the extension is disabled if the setting is omitted.

10. Security Considerations

The security considerations of HTTP/3 should be comparable to those of HTTP/2 with TLS. However, many of the considerations from Section 10 of [HTTP2] apply to [QUIC-TRANSPORT] and are discussed in that document.

10.1. Server Authority

HTTP/3 relies on the HTTP definition of authority. The security considerations of establishing authority are discussed in Section 16.1 of [SEMANTICS].

10.2. Cross-Protocol Attacks

The use of ALPN in the TLS and QUIC handshakes establishes the target application protocol before application-layer bytes are processed. Because all QUIC packets are encrypted, it is difficult for an attacker to control the plaintext bytes of an HTTP/3 connection, which could be used in a cross-protocol attack on a plaintext protocol.

10.3. Intermediary Encapsulation Attacks

The HTTP/3 field encoding allows the expression of names that are not valid field names in the syntax used by HTTP (Section 5.4.3 of [SEMANTICS]). Requests or responses containing invalid field names MUST be treated as malformed (Section 4.1.3). An intermediary therefore cannot translate an HTTP/3 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/3 can transport field values that are not valid. While most values that can be encoded will not alter field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a field value MUST be treated as malformed (Section 4.1.3). Valid characters are defined by the "field-content" ABNF rule in Section 5.4.4 of [SEMANTICS].

10.4. Cacheability of Pushed Responses

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one tenant. For example, a server might offer multiple users each a small portion of its URI space.

Where multiple tenants share space on the same server, that server MUST ensure that tenants are not able to push representations of resources that they do not have authority over. Failure to enforce this would allow a tenant to provide a representation that would be served out of cache, overriding the actual representation that the authoritative tenant provides.

Clients are required to reject pushed responses for which an origin server is not authoritative; see Section 4.4.

10.5. Denial-of-Service Considerations

An HTTP/3 connection can demand a greater commitment of resources to operate than an HTTP/1.1 or HTTP/2 connection. The use of field compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH_PROMISE frames is constrained in a similar fashion. A client that accepts server push SHOULD limit the number of Push IDs it issues at a time.

Processing capacity cannot be guarded as effectively as state capacity.

The ability to send undefined protocol elements that the peer is required to ignore can be abused to cause a peer to expend additional processing time. This might be done by setting multiple undefined SETTINGS parameters, unknown frame types, or unknown stream types. Note, however, that some uses are entirely legitimate, such as optional-to-understand extensions and padding to increase resistance to traffic analysis.

Compression of field sections also offers some opportunities to waste processing resources; see Section 7 of [QPACK] for more details on potential abuses.

All these features - i.e., server push, unknown protocol elements, field compression - have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that does not monitor this behavior exposes itself to a risk of denial-of-service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error (Section 8) of type H3_EXCESSIVE_LOAD, but false positives will result in disrupting valid connections and requests.

10.5.1. Limits on Field Section Size

A large field section (Section 4.1) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header field section, which prevents streaming of the header field section to its ultimate destination. This ordering and other reasons, such as ensuring cache correctness, mean that an endpoint likely needs to buffer the entire header field section. Since there is no hard limit to the size of a field section, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the SETTINGS_MAX_FIELD_SECTION_SIZE (Section 4.1.1.3) setting to advise peers of limits that might apply on the size of field sections. This setting is only advisory, so endpoints MAY choose to send field sections that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to an HTTP/3 connection, so any request or response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

A server that receives a larger field section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process.

10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on a proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. A proxy might also maintain some resources for a TCP connection beyond the closing of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME_WAIT state. Therefore, a proxy cannot rely on QUIC stream limits alone to control the resources consumed by CONNECT requests.

10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/3 enables compression of fields (Section 4.1.1); the following concerns also apply to the use of HTTP compressed content-codings; see Section 7.5.1 of [SEMANTICS].

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression contexts are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined.

Further considerations regarding the compression of fields sections are described in [QPACK].

10.7. Padding and Traffic Analysis

Padding can be used to obscure the exact size of frame content and is provided to mitigate specific attacks within HTTP, for example, attacks where compressed content includes both attacker-controlled plaintext and secret data (e.g., [BREACH]).

Where HTTP/2 employs PADDING frames and Padding fields in other frames to make a connection more resistant to traffic analysis, HTTP/3 can either rely on transport-layer padding or employ the reserved frame and stream types discussed in Section 7.2.8 and Section 6.2.3. These methods of padding produce different results in terms of the granularity of padding, how padding is arranged in relation to the information that is being protected, whether padding is applied in the case of packet loss, and how an implementation might control padding.

Reserved stream types can be used to give the appearance of sending traffic even when the connection is idle. Because HTTP traffic often occurs in bursts, apparent traffic can be used to obscure the timing or duration of such bursts, even to the point of appearing to send a constant stream of data. However, as such traffic is still flow controlled by the receiver, a failure to promptly drain such streams and provide additional flow control credit can limit the sender's ability to send real traffic.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Use of padding can result in less protection than might seem immediately obvious. Redundant padding could even be counterproductive. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed-sized boundary, which could be possible if an attacker can control plaintext.

10.8. Frame Parsing

Several protocol elements contain nested length elements, typically in the form of frames with an explicit length containing variable-length integers. This could pose a security risk to an incautious implementer. An implementation MUST ensure that the length of a frame exactly matches the length of the fields it contains.

10.9. Early Data

The use of 0-RTT with HTTP/3 creates an exposure to replay attack. The anti-replay mitigations in [HTTP-REPLAY] MUST be applied when using HTTP/3 with 0-RTT.

10.10. Migration

Certain HTTP implementations use the client address for logging or access-control purposes. Since a QUIC client's address might change during a connection (and future versions might support simultaneous use of multiple addresses), such implementations will need to either actively retrieve the client's current address or addresses when they are relevant or explicitly accept that the original address might change.

10.11. Privacy Considerations

Several characteristics of HTTP/3 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the timing of reactions to stimulus, and the handling of any features that are controlled by settings.

As far as these create observable differences in behavior, they could be used as a basis for fingerprinting a specific client.

HTTP/3's preference for using a single QUIC connection allows correlation of a user's activity on a site. Reusing connections for different origins allows for correlation of activity across those origins.

Several features of QUIC solicit immediate responses and can be used by an endpoint to measure latency to their peer; this might have privacy implications in certain scenarios.

11. IANA Considerations

This document registers a new ALPN protocol ID (Section 11.1) and creates new registries that manage the assignment of codepoints in HTTP/3.

11.1. Registration of HTTP/3 Identification String

This document creates a new registration for the identification of HTTP/3 in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "h3" string identifies HTTP/3:

Protocol: HTTP/3

Identification Sequence: 0x68 0x33 ("h3")

Specification: This document

11.2. New Registries

New registries created in this document operate under the QUIC registration policy documented in Section 22.1 of [QUIC-TRANSPORT]. These registries all include the common set of fields listed in Section 22.1.1 of [QUIC-TRANSPORT].

The initial allocations in these registries created in this document are all assigned permanent status and list a change controller of the IETF and a contact of the HTTP working group (ietf-http-wg@w3.org).

11.2.1. Frame Types

This document establishes a registry for HTTP/3 frame type codes. The "HTTP/3 Frame Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using

Standards Action or IESG Approval as defined in Section 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [HTTP2], it is preferable that the assignments parallel each other where the code spaces overlap. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following field:

Frame Type: A name or label for the frame type.

Specifications of frame types MUST include a description of the frame layout and its semantics, including any parts of the frame that are conditionally present.

The entries in Table 2 are registered by this document.

Frame Type	Value	Specification
DATA	0x0	Section 7.2.1
HEADERS	0x1	Section 7.2.2
Reserved	0x2	N/A
CANCEL_PUSH	0x3	Section 7.2.3
SETTINGS	0x4	Section 7.2.4
PUSH_PROMISE	0x5	Section 7.2.5
Reserved	0x6	N/A
GOAWAY	0x7	Section 7.2.6
Reserved	0x8	N/A
Reserved	0x9	N/A
MAX_PUSH_ID	0xd	Section 7.2.7

Table 2: Initial HTTP/3 Frame Types

Additionally, each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3FFFFFFFFFFFFFFE) MUST NOT be assigned by IANA.

11.2.2. Settings Parameters

This document establishes a registry for HTTP/3 settings. The "HTTP/3 Settings" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Section 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Settings" registry defined in [HTTP2], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

Setting Name: A symbolic name for the setting. Specifying a setting name is optional.

Default: The value of the setting unless otherwise indicated. A default SHOULD be the most restrictive possible value.

The entries in Table 3 are registered by this document.

Setting Name	Value	Specification	Default
Reserved	0x2	N/A	N/A
Reserved	0x3	N/A	N/A
Reserved	0x4	N/A	N/A
Reserved	0x5	N/A	N/A
MAX_FIELD_SECTION_SIZE	0x6	Section 7.2.4.1	Unlimited

Table 3: Initial HTTP/3 Settings

Additionally, each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3ffffffffffffffe) MUST NOT be assigned by IANA.

11.2.3. Error Codes

This document establishes a registry for HTTP/3 error codes. The "HTTP/3 Error Code" registry manages a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Section 4.9 and 4.10 of [RFC8126].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated. Use of values that are registered in the "HTTP/2 Error Code" registry is discouraged.

In addition to common fields as described in Section 11.2, this registry includes two additional fields. Permanent registrations in this registry MUST include the following field:

Name: A name for the error code.

Description: A brief description of the error code semantics.

The entries in Table 4 are registered by this document. These error codes were selected from the range that operates on a Specification Required policy to avoid collisions with HTTP/2 error codes.

Name	Value	Description	Specification
H3_NO_ERROR	0x0100	No error	Section 8.1
H3_GENERAL_PROTOCOL_ERROR	0x0101	General protocol error	Section 8.1
H3_INTERNAL_ERROR	0x0102	Internal error	Section 8.1
H3_STREAM_CREATION_ERROR	0x0103	Stream creation	Section 8.1

		error	
H3_CLOSED_CRITICAL_STREAM	0x0104	Critical stream was closed	Section 8.1
H3_FRAME_UNEXPECTED	0x0105	Frame not permitted in the current state	Section 8.1
H3_FRAME_ERROR	0x0106	Frame violated layout or size rules	Section 8.1
H3_EXCESSIVE_LOAD	0x0107	Peer generating excessive load	Section 8.1
H3_ID_ERROR	0x0108	An identifier was used incorrectly	Section 8.1
H3_SETTINGS_ERROR	0x0109	SETTINGS frame contained invalid values	Section 8.1
H3_MISSING_SETTINGS	0x010a	No SETTINGS frame received	Section 8.1
H3_REQUEST_REJECTED	0x010b	Request not processed	Section 8.1
H3_REQUEST_CANCELLED	0x010c	Data no longer needed	Section 8.1
H3_REQUEST_INCOMPLETE	0x010d	Stream terminated early	Section 8.1

H3_CONNECT_ERROR	0x010f	TCP reset or error on CONNECT request	Section 8.1
H3_VERSION_FALLBACK	0x0110	Retry over HTTP/1.1	Section 8.1

Table 4: Initial HTTP/3 Error Codes

Additionally, each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3ffffffffffffffe) MUST NOT be assigned by IANA.

11.2.4. Stream Types

This document establishes a registry for HTTP/3 unidirectional stream types. The "HTTP/3 Stream Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Section 4.9 and 4.10 of [RFC8126].

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

Stream Type: A name or label for the stream type.

Sender: Which endpoint on an HTTP/3 connection may initiate a stream of this type. Values are "Client", "Server", or "Both".

Specifications for permanent registrations MUST include a description of the stream type, including the layout and semantics of the stream contents.

The entries in the following table are registered by this document.

Stream Type	Value	Specification	Sender
Control Stream	0x00	Section 6.2.1	Both
Push Stream	0x01	Section 4.4	Server

Table 5

Additionally, each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3ffffffffffffffe) MUST NOT be assigned by IANA.

12. References

12.1. Normative References

[ALTSVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[CACHING] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-12, 2 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-12.txt>.

[HTTP-REPLAY] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/info/rfc8470>.

[QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-19, 20 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-qpack-19>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-31, 20 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-31>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[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>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/info/rfc6125>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC8126] 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/info/rfc8126>.

[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., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-12, 2 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-semantics-12.txt>.

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

12.2. Informative References

[BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>.

[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/info/rfc7541>.

[HTTP11] Fielding, R., Nottingham, M., and J. Reschke, "HTTP/1.1 Messaging", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-12, 2 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-messaging-12.txt>.

[HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[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>.

[RFC8164] Nottingham, M. and M. Thomson, "Opportunistic Security for HTTP/2", RFC 8164, DOI 10.17487/RFC8164, May 2017, <https://www.rfc-editor.org/info/rfc8164>.

[TFO] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

Appendix A. Considerations for Transitioning from HTTP/2

HTTP/3 is strongly informed by HTTP/2, and bears many similarities. This section describes the approach taken to design HTTP/3, points out important differences from HTTP/2, and describes how to map HTTP/2 extensions into HTTP/3.

HTTP/3 begins from the premise that similarity to HTTP/2 is preferable, but not a hard requirement. HTTP/3 departs from HTTP/2 where QUIC differs from TCP, either to take advantage of QUIC features (like streams) or to accommodate important shortcomings (such as a lack of total ordering). These differences make HTTP/3 similar to HTTP/2 in key aspects, such as the relationship of requests and responses to streams. However, the details of the HTTP/3 design are substantially different from HTTP/2.

These departures are noted in this section.

A.1. Streams

HTTP/3 permits use of a larger number of streams (2^62-1) than HTTP/2. The same considerations about exhaustion of stream identifier space apply, though the space is significantly larger such that it is likely that other limits in QUIC are reached first, such as the limit on the connection flow control window.

In contrast to HTTP/2, stream concurrency in HTTP/3 is managed by QUIC. QUIC considers a stream closed when all data has been received and sent data has been acknowledged by the peer. HTTP/2 considers a stream closed when the frame containing the END_STREAM bit has been committed to the transport. As a result, the stream for an equivalent exchange could remain "active" for a longer period of time. HTTP/3 servers might choose to permit a larger number of concurrent client-initiated bidirectional streams to achieve equivalent concurrency to HTTP/2, depending on the expected usage patterns.

Due to the presence of other unidirectional stream types, HTTP/3 does not rely exclusively on the number of concurrent unidirectional streams to control the number of concurrent in-flight pushes. Instead, HTTP/3 clients use the MAX_PUSH_ID frame to control the number of pushes received from an HTTP/3 server.

A.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END_STREAM flag is not required. This permits the removal of the Flags field from the generic frame layout.

Frame payloads are largely drawn from [HTTP2]. However, QUIC includes many features (e.g., flow control) that are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/3. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/3 implementations. However, even equivalent frames between the two mappings are not identical.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/3 will break them.

Some examples of feature adaptations are described below, as well as general guidance to extension frame implementors converting an HTTP/2 extension to HTTP/3.

A.2.1. Prioritization Differences

HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. HTTP/3 does not provide a means of signaling priority.

Note that while there is no explicit signaling for priority, this does not mean that prioritization is not important for achieving good performance.

A.2.2. Field Compression Differences

HPACK was designed with the assumption of in-order delivery. A sequence of encoded field sections must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

Because this total ordering is not provided by QUIC, HTTP/3 uses a modified version of HPACK, called QPACK. QPACK uses a single unidirectional stream to make all modifications to the dynamic table, ensuring a total order of updates. All frames that contain encoded fields merely reference the table state at a given time without modifying it.

[QPACK] provides additional details.

A.2.3. Flow Control Differences

HTTP/2 specifies a stream flow control mechanism. Although all HTTP/2 frames are delivered on streams, only the DATA frame payload is subject to flow control. QUIC provides flow control for stream data and all HTTP/3 frame types defined in this document are sent on streams. Therefore, all frame headers and payload are subject to flow control.

A.2.4. Guidance for New Frame Type Definitions

Frame type definitions in HTTP/3 often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allows for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/3 use an identifier rather than a Stream ID (e.g., Push IDs). Redefinition of the encoding of extension frame types might be necessary if the encoding includes a Stream ID.

Because the Flags field is not present in generic HTTP/3 frames, those frames that depend on the presence of flags need to allocate space for flags as part of their frame payload.

Other than these issues, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with a control stream in HTTP/3. HTTP/3 extensions will not assume ordering, but would not be harmed by ordering, and would be portable to HTTP/2 in the same manner.

A.2.5. Mapping Between HTTP/2 and HTTP/3 Frame Types

DATA (0x0): Padding is not defined in HTTP/3 frames. See Section 7.2.1.

HEADERS (0x1): The PRIORITY region of HEADERS is not defined in HTTP/3 frames. Padding is not defined in HTTP/3 frames. See Section 7.2.2.

PRIORITY (0x2): As described in Appendix A.2.1, HTTP/3 does not provide a means of signaling priority.

RST_STREAM (0x3): RST_STREAM frames do not exist in HTTP/3, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL_PUSH frame (Section 7.2.3).

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 7.2.4 and Appendix A.3.

PUSH_PROMISE (0x5): The PUSH_PROMISE frame does not reference a stream; instead the push stream references the PUSH_PROMISE frame using a Push ID. See Section 7.2.5.

PING (0x6): PING frames do not exist in HTTP/3, as QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY does not contain an error code. In the client to server direction, it carries a Push ID instead of a server initiated stream ID. See Section 7.2.6.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist in HTTP/3, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist in HTTP/3; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted.

Frame types defined by extensions to HTTP/2 need to be separately registered for HTTP/3 if still applicable. The IDs of frames defined in [HTTP2] have been reserved for simplicity. Note that the frame type space in HTTP/3 is substantially larger (62 bits versus 8 bits), so many HTTP/3 frame types have no equivalent HTTP/2 code points. See Section 11.2.1.

A.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, as the first frame of the control stream, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/3. The HTTP-level options that are retained in HTTP/3 have the same value as in HTTP/2. The superseded settings are reserved, and their receipt is an error. See Section 7.2.4.1 for discussion of both the retained and reserved values.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE: See [QPACK].

SETTINGS_ENABLE_PUSH: This is removed in favor of the MAX_PUSH_ID frame, which provides a more granular control over server push. Specifying a setting with the identifier 0x2 (corresponding to the SETTINGS_ENABLE_PUSH parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_CONCURRENT_STREAMS: QUIC controls the largest open Stream ID as part of its flow control logic. Specifying a setting with the identifier 0x3 (corresponding to the SETTINGS_MAX_CONCURRENT_STREAMS parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE: QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying a setting with the identifier 0x4 (corresponding to the SETTINGS_INITIAL_WINDOW_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE: This setting has no equivalent in HTTP/3. Specifying a setting with the identifier 0x5 (corresponding to the SETTINGS_MAX_FRAME_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE: This setting identifier has been renamed SETTINGS_MAX_FIELD_SECTION_SIZE.

In HTTP/3, setting values are variable-length integers (6, 14, 30, or 62 bits long) rather than fixed-length 32-bit fields as in HTTP/2. This will often produce a shorter encoding, but can produce a longer encoding for settings that use the full 32-bit space. Settings ported from HTTP/2 might choose to redefine their value to limit it to 30 bits for more efficient encoding, or to make use of the 62-bit space if more than 30 bits are required.

Settings need to be defined separately for HTTP/2 and HTTP/3. The IDs of settings defined in [HTTP2] have been reserved for simplicity. Note that the settings identifier space in HTTP/3 is substantially larger (62 bits versus 16 bits), so many HTTP/3 settings have no equivalent HTTP/2 code point. See Section 11.2.2.

As QUIC streams might arrive out of order, endpoints are advised not to wait for the peers' settings to arrive before responding to other streams. See Section 7.2.4.2.

A.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, the differences between HTTP/2 and HTTP/3 mean that error codes are not directly portable between versions.

The HTTP/2 error codes defined in Section 7 of [HTTP2] logically map to the HTTP/3 error codes as follows:

NO_ERROR (0x0): H3_NO_ERROR in Section 8.1.

PROTOCOL_ERROR (0x1): This is mapped to H3_GENERAL_PROTOCOL_ERROR except in cases where more specific error codes have been defined. Such cases include H3_FRAME_UNEXPECTED and H3_CLOSED_CRITICAL_STREAM defined in Section 8.1.

INTERNAL_ERROR (0x2): H3_INTERNAL_ERROR in Section 8.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgement of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management.

FRAME_SIZE_ERROR (0x6): H3_FRAME_ERROR error code defined in Section 8.1.

REFUSED_STREAM (0x7): H3_REQUEST_REJECTED (in Section 8.1) is used to indicate that a request was not processed. Otherwise, not applicable because QUIC handles stream management.

CANCEL (0x8): H3_REQUEST_CANCELLED in Section 8.1.

COMPRESSION_ERROR (0x9): Multiple error codes are defined in [QPACK].

CONNECT_ERROR (0xa): H3_CONNECT_ERROR in Section 8.1.

ENHANCE_YOUR_CALM (0xb): H3_EXCESSIVE_LOAD in Section 8.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): H3_VERSION_FALLBACK in Section 8.1.

Error codes need to be defined for HTTP/2 and HTTP/3 separately. See Section 11.2.3.

A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors

An intermediary that converts between HTTP/2 and HTTP/3 may encounter error conditions from either upstream. It is useful to communicate the occurrence of error to the downstream but error codes largely reflect connection-local problems that generally do not make sense to propagate.

An intermediary that encounters an error from an upstream origin can indicate this by sending an HTTP status code such as 502, which is suitable for a broad class of errors.

There are some rare cases where it is beneficial to propagate the error by mapping it to the closest matching error type to the receiver. For example, an intermediary that receives an HTTP/2 stream error of type REFUSED_STREAM from the origin has a clear signal that the request was not processed and that the request is safe to retry. Propagating this error condition to the client as an HTTP/3 stream error of type H3_REQUEST_REJECTED allows the client to take the action it deems most appropriate. In the reverse direction, the intermediary might deem it beneficial to pass on client request cancellations that are indicated by terminating a stream with H3_REQUEST_CANCELLED; see Section 4.1.2.

Conversion between errors is described in the logical mapping. The error codes are defined in non-overlapping spaces in order to protect against accidental conversion that could result in the use of inappropriate or unknown error codes for the target version. An intermediary is permitted to promote stream errors to connection errors but they should be aware of the cost to the HTTP/3 connection for what might be a temporary or intermittent error.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-http-31

Editorial changes only.

B.2. Since draft-ietf-quic-http-30

Editorial changes only.

B.3. Since draft-ietf-quic-http-29

* Require a connection error if a reserved frame type that corresponds to a frame in HTTP/2 is received (#3991, #3993)

* Require a connection error if a reserved setting that corresponds to a setting in HTTP/2 is received (#3954, #3955)

B.4. Since draft-ietf-quic-http-28

* CANCEL_PUSH is recommended even when the stream is reset (#3698, #3700)

* Use H3_ID_ERROR when GOAWAY contains a larger identifier (#3631, #3634)

B.5. Since draft-ietf-quic-http-27

* Updated text to refer to latest HTTP revisions

* Use the HTTP definition of authority for establishing and coalescing connections (#253, #2223, #3558)

* Define use of GOAWAY from both endpoints (#2632, #3129)

* Require either :authority or Host if the URI scheme has a mandatory authority component (#3408, #3475)

B.6. Since draft-ietf-quic-http-26

* No changes

B.7. Since draft-ietf-quic-http-25

* Require QUICv1 for HTTP/3 (#3117, #3323)

* Remove DUPLICATE_PUSH and allow duplicate PUSH_PROMISE (#3275, #3309)

* Clarify the definition of "malformed" (#3352, #3345)

B.8. Since draft-ietf-quic-http-24

* Removed H3_EARLY_RESPONSE error code; H3_NO_ERROR is recommended instead (#3130,#3208)

* Unknown error codes are equivalent to H3_NO_ERROR (#3276,#3331)

* Some error codes are reserved for greasing (#3325,#3360)

B.9. Since draft-ietf-quic-http-23

* Removed "quic" Alt-Svc parameter (#3061,#3118)

* Clients need not persist unknown settings for use in 0-RTT (#3110,#3113)

* Clarify error cases around CANCEL_PUSH (#2819,#3083)

B.10. Since draft-ietf-quic-http-22

* Removed priority signaling (#2922,#2924)

* Further changes to error codes (#2662,#2551):

  - Error codes renumbered

  - HTTP_MALFORMED_FRAME replaced by HTTP_FRAME_ERROR, HTTP_ID_ERROR, and others

* Clarify how unknown frame types interact with required frame sequence (#2867,#2858)

* Describe interactions with the transport in terms of defined interface terms (#2857,#2805)

* Require the use of the "http-opportunistic" resource (RFC 8164) when scheme is "http" (#2439,#2973)

* Settings identifiers cannot be duplicated (#2979)

* Changes to SETTINGS frames in 0-RTT (#2972,#2790,#2945):

  - Servers must send all settings with non-default values in their SETTINGS frame, even when resuming

  - If a client doesn't have settings associated with a 0-RTT ticket, it uses the defaults

  - Servers can't accept early data if they cannot recover the settings the client will have remembered

* Clarify that Upgrade and the 101 status code are prohibited (#2898,#2889)

* Clarify that frame types reserved for greasing can occur on any stream, but frame types reserved due to HTTP/2 correspondence are prohibited (#2997,#2692,#2693)

* Unknown error codes cannot be treated as errors (#2998,#2816)

B.11. Since draft-ietf-quic-http-21

No changes

B.12. Since draft-ietf-quic-http-20

* Prohibit closing the control stream (#2509, #2666)

* Change default priority to use an orphan node (#2502, #2690)

* Exclusive priorities are restored (#2754, #2781)

* Restrict use of frames when using CONNECT (#2229, #2702)

* Close and maybe reset streams if a connection error occurs for CONNECT (#2228, #2703)

* Encourage provision of sufficient unidirectional streams for QPACK (#2100, #2529, #2762)

* Allow extensions to use server-initiated bidirectional streams (#2711, #2773)

* Clarify use of maximum header list size setting (#2516, #2774)

* Extensive changes to error codes and conditions of their sending

  - Require connection errors for more error conditions (#2511, #2510)

  - Updated the error codes for illegal GOAWAY frames (#2714, #2707)

  - Specified error code for HEADERS on control stream (#2708)

  - Specified error code for servers receiving PUSH_PROMISE (#2709)

  - Specified error code for receiving DATA before HEADERS (#2715)

  - Describe malformed messages and their handling (#2410, #2764)

  - Remove HTTP_PUSH_ALREADY_IN_CACHE error (#2812, #2813)

  - Refactor Push ID related errors (#2818, #2820)

  - Rationalize HTTP/3 stream creation errors (#2821, #2822)

B.13. Since draft-ietf-quic-http-19

* SETTINGS_NUM_PLACEHOLDERS is 0x9 (#2443,#2530)

* Non-zero bits in the Empty field of the PRIORITY frame MAY be treated as an error (#2501)

B.14. Since draft-ietf-quic-http-18

* Resetting streams following a GOAWAY is recommended, but not required (#2256,#2457)

* Use variable-length integers throughout (#2437,#2233,#2253,#2275)

  - Variable-length frame types, stream types, and settings identifiers

  - Renumbered stream type assignments

  - Modified associated reserved values

* Frame layout switched from Length-Type-Value to Type-Length-Value (#2395,#2235)

* Specified error code for servers receiving DUPLICATE_PUSH (#2497)

* Use connection error for invalid PRIORITY (#2507, #2508)

B.15. Since draft-ietf-quic-http-17

* HTTP_REQUEST_REJECTED is used to indicate a request can be retried (#2106, #2325)

* Changed error code for GOAWAY on the wrong stream (#2231, #2343)

B.16. Since draft-ietf-quic-http-16

* Rename "HTTP/QUIC" to "HTTP/3" (#1973)

* Changes to PRIORITY frame (#1865, #2075)

  - Permitted as first frame of request streams

  - Remove exclusive reprioritization

  - Changes to Prioritized Element Type bits

* Define DUPLICATE_PUSH frame to refer to another PUSH_PROMISE (#2072)

* Set defaults for settings, allow request before receiving SETTINGS (#1809, #1846, #2038)

* Clarify message processing rules for streams that aren't closed (#1972, #2003)

* Removed reservation of error code 0 and moved HTTP_NO_ERROR to this value (#1922)

* Removed prohibition of zero-length DATA frames (#2098)

B.17. Since draft-ietf-quic-http-15

Substantial editorial reorganization; no technical changes.

B.18. Since draft-ietf-quic-http-14

* Recommend sensible values for QUIC transport parameters (#1720,#1806)

* Define error for missing SETTINGS frame (#1697,#1808)

* Setting values are variable-length integers (#1556,#1807) and do not have separate maximum values (#1820)

* Expanded discussion of connection closure (#1599,#1717,#1712)

* HTTP_VERSION_FALLBACK falls back to HTTP/1.1 (#1677,#1685)

B.19. Since draft-ietf-quic-http-13

* Reserved some frame types for grease (#1333, #1446)

* Unknown unidirectional stream types are tolerated, not errors; some reserved for grease (#1490, #1525)

* Require settings to be remembered for 0-RTT, prohibit reductions (#1541, #1641)

* Specify behavior for truncated requests (#1596, #1643)

B.20. Since draft-ietf-quic-http-12

* TLS SNI extension isn't mandatory if an alternative method is used (#1459, #1462, #1466)

* Removed flags from HTTP/3 frames (#1388, #1398)

* Reserved frame types and settings for use in preserving extensibility (#1333, #1446)

* Added general error code (#1391, #1397)

* Unidirectional streams carry a type byte and are extensible (#910,#1359)

* Priority mechanism now uses explicit placeholders to enable persistent structure in the tree (#441,#1421,#1422)

B.21. Since draft-ietf-quic-http-11

* Moved QPACK table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

B.22. Since draft-ietf-quic-http-10

* Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.23. Since draft-ietf-quic-http-09

* Selected QCRAM for header compression (#228, #1117)

* The server_name TLS extension is now mandatory (#296, #495)

* Specified handling of unsupported versions in Alt-Svc (#1093, #1097)

B.24. Since draft-ietf-quic-http-08

* Clarified connection coalescing rules (#940, #1024)

B.25. Since draft-ietf-quic-http-07

* Changes for integer encodings in QUIC (#595,#905)

* Use unidirectional streams as appropriate (#515, #240, #281, #886)

* Improvement to the description of GOAWAY (#604, #898)

* Improve description of server push usage (#947, #950, #957)

B.26. Since draft-ietf-quic-http-06

* Track changes in QUIC error code usage (#485)

B.27. Since draft-ietf-quic-http-05

* Made push ID sequential, add MAX_PUSH_ID, remove SETTINGS_ENABLE_PUSH (#709)

* Guidance about keep-alive and QUIC PINGs (#729)

* Expanded text on GOAWAY and cancellation (#757)

B.28. Since draft-ietf-quic-http-04

* Cite RFC 5234 (#404)

* Return to a single stream per request (#245,#557)

* Use separate frame type and settings registries from HTTP/2 (#81)

* SETTINGS_ENABLE_PUSH instead of SETTINGS_DISABLE_PUSH (#477)

* Restored GOAWAY (#696)

* Identify server push using Push ID rather than a stream ID (#702,#281)

* DATA frames cannot be empty (#700)

B.29. Since draft-ietf-quic-http-03

None.

B.30. Since draft-ietf-quic-http-02

* Track changes in transport draft

B.31. Since draft-ietf-quic-http-01

* SETTINGS changes (#181):

  - SETTINGS can be sent only once at the start of a connection; no changes thereafter

  - SETTINGS_ACK removed

  - Settings can only occur in the SETTINGS frame a single time

  - Boolean format updated

* Alt-Svc parameter changed from "v" to "quic"; format updated (#229)

* Closing the connection control stream or any message control stream is a fatal error (#176)

* HPACK Sequence counter can wrap (#173)

* 0-RTT guidance added

* Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.32. Since draft-ietf-quic-http-00

* Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)

* Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)

* Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01

* Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)

* Described CONNECT pseudo-method (#95)

* Updated ALPN token and Alt-Svc guidance (#13,#87)

* Application-layer-defined error codes (#19,#74)

B.33. Since draft-shade-quic-http2-mapping-00

* Adopted as base for draft-ietf-quic-http

* Updated authors/editors list

Acknowledgements

The original authors of this specification were Robbie Shade and Mike Warres.

The IETF QUIC Working Group received an enormous amount of support from many people. Among others, the following people provided substantial contributions to this document:

* Bence Beky

* Daan De Meyer

* Martin Duke

* Roy Fielding

* Alan Frindell
* Alessandro Ghedini
* Nick Harper
* Ryan Hamilton
* Christian Huitema
* Subodh Iyengar
* Robin Marx
* Patrick McManus
* Luca Niccolini
* (Kazuho Oku)
* Lucas Pardue
* Roberto Peon
* Julian Reschke
* Eric Rescorla
* Martin Seemann
* Ben Schwartz
* Ian Swett
* Willy Taureau
* Martin Thomson
* Dmitri Tikhonov
* Tatsuhiro Tsujikawa

A portion of Mike's contribution was supported by Microsoft during his employment there.

Author's Address

Mike Bishop (editor)
Akamai

Email: mbishop@evequefou.be

Network Working Group E. Kinnear
Internet-Draft T. Pauly
Intended status: Standards Track Apple Inc.
Expires: March 14, 2019 September 10, 2018

Using HTTP/2 as a Transport for Arbitrary Bytestreams
draft-kinnear-httpbis-http2-transport-00

Abstract

HTTP/2 provides multiplexing of HTTP requests over a single underlying transport connection. HTTP/2 Transport defines a transport abstraction provided by HTTP/2 framing that is separate from HTTP semantics.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

HTTP/2 [RFC7540] provides a framing layer that describes the exchange of HTTP messages following HTTP semantics. This framing layer provides multiplexing of multiple streams on a single underlying transport connection, flow control, stream dependencies and priorities, and exchange of configuration information between endpoints. HTTP/2 also defines the mapping of HTTP semantics onto that framing layer.

This document defines the use of the HTTP/2 framing layer as a transport for arbitrary byte streams without the use of HTTP semantics.

1.1. 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.

2. The STREAM HTTP/2 Frame

This document defines a new HTTP/2 frame type called STREAM, that allows endpoints to open HTTP/2 streams without header values. Either endpoint can send this frame to open a stream. STREAM frames are treated in all ways as HEADERS frames, including in the stream state machine, but are not required to contain any header values.

2.1. Syntax

The STREAM frame type is 0xd (decimal 13) and contains similar fields to that of the HEADERS frame.

A STREAM frame is shown below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+
|Pad Length? (8)|
+-+-------------+-----------------------------------------------+
|E|                 Stream Dependency? (31)                     |
+-+-------------+-----------------------------------------------+
|  Weight? (8)  |
+-+-------------+-----------------------------------------------+
|                           Padding (*)                       ...
+---------------------------------------------------------------+
```

Figure 1: STREAM Frame Format

The STREAM frame contains the following fields:

Pad Length: An 8-bit field containing the length of the frame padding in units of octets. This field is only present if the PADDED flag is set.

E: A single-bit flag indicating that the stream dependency is exclusive (see Section 5.3 of [RFC7540]). This field is only present if the PRIORITY flag is set.

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on (see Section 5.3 of [RFC7540]). This field is only present if the PRIORITY flag is set.

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see Section 5.3 of [RFC7540]). Add one to the value to obtain a weight between 1 and 256. This field is only present if the PRIORITY flag is set.

Padding: Padding octets.

The STREAM frame defines the following flags:

PADDED (0x8): When set, bit 3 indicates that the Pad Length field and any padding that it describes are present.

PRIORITY (0x20): When set, bit 5 indicates that the Exclusive Flag (E), Stream Dependency, and Weight fields are present; see Section 5.3 of [RFC7540].

STREAM frames MUST be associated with a stream. If a STREAM frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error of type PROTOCOL_ERROR.

The STREAM frame can include padding. Padding fields and flags are identical to those defined for DATA frames.

Prioritization information in a STREAM frame is logically equivalent to a separate PRIORITY frame, but inclusion in a STREAM frame avoids the potential for churn in stream prioritization when new streams are created. Prioritization fields in STREAM frames subsequent to the first on a stream reprioritize the stream (Section 5.3.3 of [RFC7540]).

2.2. Advertising Support for STREAM Frames

As defined in [RFC7540], both endpoints can send SETTINGS_MAX_CONCURRENT_STREAMS in SETTINGS frames to indicate the number of streams that the sender permits the receiver to create. This limit applies to streams created via the STREAM frame as well as streams created via HEADERS frames.

2.3. Processing STREAM Frames

The STREAM frame is a non-critical extension to HTTP/2. Endpoints that do not support this frame can safely ignore it upon receipt.

When received by a client that implements support, the STREAM frame behaves in the same manner as a HEADERS frame, but does not carry any header blocks. This changes the connection state in the same manner as a HEADERS frame, described in Section 4.3 of [RFC7540].

STREAM frames can be sent on a stream in the "idle", "reserved (local)", "open", or "half-closed (remote)" state. STREAM frames can be sent by either endpoint on a connection.

Streams created via a STREAM frame are multiplexed in the same manner on the underlying transport connection as streams created via a HEADERS frame. Flow control also applies to these streams in the same way. Flow control, stream dependencies, and priorities continue to apply to streams as defined by [RFC7540].

Anywhere an endpoint would be permitted to send a HEADERS frame by [RFC7540], it is likewise permitted to send a STREAM frame.

A stream is closed via a RST_STREAM frame or by setting the END_STREAM flag on a DATA frame.

3. IANA Considerations

This specification adds an entry to the "HTTP/2 Frame Type" registry.

- Frame Type: STREAM

- Code: 0xd

- Specification: [[RFC Editor: Please fill in this value with the RFC number for this document.]]

4. Security Considerations

5. Acknowledgments

Thanks to Anthony Chivetta, Joshua Otto, and Valentin Pistol for their contributions in the design and prototyping of this work.

6. Normative References

[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>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[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>.

Authors' Addresses

Eric Kinnear
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: ekinnear@apple.com

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: tpauly@apple.com