

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 5, 2017

M. Nottingham
February 1, 2017

Retrying HTTP Requests draft-nottingham-httpbis-retry-01

Abstract

HTTP allows requests to be automatically retried under certain circumstances. This draft explores how this is implemented, requirements for similar functionality from other parts of the stack, and potential future improvements.

Note to Readers

This draft is not intended to be published as an RFC.

The issues list for this draft can be found at <https://github.com/mnot/I-D/labels/httpbis-retry> .

The most recent (often, unpublished) draft is at <https://mnot.github.io/I-D/httpbis-retry/> .

Recent changes are listed at <https://github.com/mnot/I-D/commits/gh-pages/httpbis-retry> .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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

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

This Internet-Draft will expire on August 5, 2017.

Internet-Draft

Retrying HTTP Requests

February 2017

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	3
1.1.	Notational Conventions	3
2.	Background	3
2.1.	Retries and Replays: A Taxonomy of Repetition	3
2.2.	What the Spec Says: Automatic Retries	4
2.3.	What the Specs Say: Replay	5
2.3.1.	TCP Fast Open	5
2.3.2.	TLS 1.3	5
2.3.3.	QUIC	6
3.	Discussion	6
3.1.	Automatic Retries In Practice	6
3.2.	Replays Are Different	7
4.	Possible Areas of Work	8
4.1.	Updating HTTP's Requirements for Retries	8
4.2.	Protocol Extensions	9
4.3.	Feedback to Transport 0RTT Efforts	9
5.	Security Considerations	9
6.	Acknowledgements	9
7.	References	10
7.1.	Normative References	10
7.2.	Informative References	10
7.3.	URIs	11
Appendix A.	When Clients Retry	11
A.1.	Squid	11
A.2.	Traffic Server	12
A.3.	Firefox	14

A.4.	Chromium	16
A.5.	Curl	17
Author's Address	18

[1.](#) Introduction

One of the benefits of HTTP's well-defined method semantics is that they allow failed requests to be retried, under certain circumstances.

However, interest in extending, redefining or just clarifying HTTP's retry semantics is increasing, for a number of reasons:

- o Since HTTP/1.1's requirements were written, there has been a substantial amount of experience deploying and using HTTP, leading implementations to refine their behaviour, often diverging from the specification.
- o Likewise, changes such as HTTP/2 [[RFC7540](#)] might change the underlying assumptions that these requirements were based upon.
- o Emerging lower-layer developments such as TCP Fast Open [[RFC7413](#)], TLS/1.3 [[I-D.ietf-tls-tls13](#)] and QUIC [[I-D.ietf-quic-transport](#)] introduce the possibility of replayed requests in the beginning of a connection, thanks to Zero Round Trip (0RT) modes. In some ways, these are similar to retries - but not completely.
- o Applications sometimes want requests to be retried by infrastructure, but can't easily express them in a non-idempotent request (such as GET).

This draft gives some background in [Section 2](#), discusses aspects of these issues in [Section 3](#), suggesting possible areas of work in [Section 4](#), and cataloguing current implementation behaviours in [Appendix A](#).

[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.](#) Background

[2.1.](#) Retries and Replays: A Taxonomy of Repetition

In HTTP, there are three similar but separate phenomena that deserve consideration for the purposes of this document:

1. **User Retries** happen when a user initiates an action that results in a duplicate HTTP request message being emitted. For example, a user retry might occur when a "reload" button is

Nottingham

Expires August 5, 2017

[Page 3]

Internet-Draft

Retrying HTTP Requests

February 2017

pressed, a URL is typed in again, "return" is pressed in the URL bar again, or a navigation link or form button is pressed twice while still on screen.

2. **Automatic Retries** happen when an HTTP client implementation resends a previous request message without user intervention or initiation. This might happen when a GET request fails to return a complete response, or when a connection drops before the request is sent. Note that automatic retries can (and are) performed both by user agents and intermediary clients.
3. **Replays** happen when the underlying transport units (e.g., TCP packets, QUIC frames) containing a HTTP request message are re-sent on the network **and** appear to be separate requests to the downstream server, either automatically as part of transport protocol operation, or by an attacker. The upstream HTTP client might not have any indication that a replay has occurred.

Note that retries initiated by code shipped to the client by the server (e.g., in JavaScript) occupy a grey area here. Because they are not initiated by the generic HTTP client implementation itself, we will consider them user retries for the time being.

Also, this document doesn't include transport layer loss recovery (e.g., TCP retransmission). This is distinguished from replays because the transport automatically suppresses duplicates.

[2.2.](#) What the Spec Says: Automatic Retries

[\[RFC7230\]](#), [Section 6.3.1](#) allows HTTP requests to be retried in certain circumstances:

When an inbound connection is closed prematurely, a client MAY open a new connection and automatically retransmit an aborted sequence of requests if all of those requests have idempotent methods ([Section 4.2.2 of \[RFC7231\]](#)). A proxy MUST NOT automatically retry non-idempotent requests.

A user agent MUST NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are actually idempotent, regardless of the method, or some means to detect that the original request was never applied. For example, a user agent that knows (through design or configuration) that a POST request to a given resource is safe can repeat that request automatically. Likewise, a user agent designed specifically to operate on a version control repository might be able to recover from partial failure conditions by checking the target resource revision(s) after a

failed connection, reverting or fixing any changes that were partially applied, and then automatically retrying the requests that failed.

A client SHOULD NOT automatically retry a failed automatic retry.

Note that the complete list of idempotent methods is maintained in the IANA HTTP Method Registry [\[4\]](#).

[2.3.](#) What the Specs Say: Replay

[2.3.1.](#) TCP Fast Open

[\[RFC7413\]](#), [Section 6.3.1](#) addresses HTTP Request Replay with TCP Fast Open:

While TFO is motivated by Web applications, the browser should not use TFO to send requests in SYNs if those requests cannot tolerate replays. One example is POST requests without application-layer transaction protection (e.g., a unique identifier in the request header).

On the other hand, TFO is particularly useful for GET requests. GET request replay could happen across striped TCP connections: after a server receives an HTTP request but before the ACKs of the requests reach the browser, the browser may time out and retry the same request on another (possibly new) TCP connection. This differs from a TFO replay only in that the replay is initiated by the browser, not by the TCP stack.

The same specification addresses HTTP over TLS in [Section 6.3.2](#):

For Transport Layer Security (TLS) over TCP, it is safe and useful to include a TLS client_hello in the SYN packet to save one RTT in the TLS handshake. There is no concern about violating idempotency. In particular, it can be used alone with the speculative connection above.

[2.3.2](#). TLS 1.3

[[I-D.ietf-tls-tls13](#)], Section 2.3 explains the properties of Zero-RTT Data in TLS 1.3:

IMPORTANT NOTE: The security properties for 0-RTT data (regardless of the cipher suite) are weaker than those for other kinds of TLS data. Specifically:

1. This data is not forward secret, because it is encrypted solely with the PSK.
2. There are no guarantees of non-replay between connections. Unless the server takes special measures outside those provided by TLS, the server has no guarantee that the same 0-RTT data was not transmitted on multiple 0-RTT connections (See [Section 4.2.6.2](#) for more details). This is especially relevant if the data is authenticated either with TLS client authentication or inside the application layer protocol. However, 0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection) and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys.)

[Section 4.2.6](#) defines a mechanism to limit the exposure to replay.

[2.3.3.](#) QUIC

[I-D.ietf-quic-tls] [Section 7.2](#) says this about the risks of replay during the 0RTT handshake:

If 0-RTT keys are available, the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server MUST NOT use 0-RTT keys to protect packets.

[3.](#) Discussion

[3.1.](#) Automatic Retries In Practice

In practice, it has been observed (see [Appendix A](#)) that some client implementations (both user agent and intermediary) do automatically retry requests. However, they do not do so consistently, and

arguably not in the spirit of the specification, unless this vague catch-all:

some means to detect that the original request was never applied is interpreted very broadly.

On the server side, it has been widely observed that content on the

Web doesn't always honour HTTP idempotency semantics, with many GET requests incurring side effects, and with some sites even requiring browsers to retry POST requests in order to properly interoperate.

Despite this situation, the Web seems to work reasonably well to date (with notable exceptions [5]).

The status quo, therefore, is that no Web application can read HTTP's retry requirements as a guarantee that any given request won't be retried, even for methods that are not idempotent. As a result, applications that care about avoiding duplicate requests need to build a way to detect not only user retries but also automatic retries into the application "above" HTTP itself.

[3.2.](#) Replays Are Different

TCP Fast Open [RFC7413], TLS/1.3 [I-D.ietf-tls-tls13] and QUIC [I-D.ietf-quic-transport] all have mechanisms to carry application data on the first packet sent by a client, to avoid the latency of connection setup.

The request(s) in this first packet might be `_replayed_`, either because the first packet (now carrying a HTTP request) is thought to be lost and retransmitted, or because an attacker observes the packet and sends a duplicate at some point in the future.

At first glance, it seems as if the idempotency semantics of HTTP request methods could be used to determine what requests are suitable for inclusion in the first packet of various 0RTT mechanisms being discussed (as suggested by TCP Fast Open). For example, we could disallow POST (and other non-idempotent methods) in 0RTT data.

Upon reflection, though, the observations above lead us to believe that since any request might be retried (automatically or by users), applications will still need to have a means of detecting duplicate requests, thereby preventing side effects from replays as well as retries. Thus, any HTTP request can be included in the first packet of a 0RTT, despite the risk of replay.

Two types of attack specific to replayed HTTP requests need to be

taken into account, however:

1. A replay is a potential Denial of Service vector. An attacker that can replay a request many times can probe for weaknesses in retry protections, and can bring a server that needs to do any substantial processing down.
2. An attacker might use a replayed request to leak information about the response over time. If they can observe the encrypted payload on the wire, they can infer the size of the response (e.g., it might get bigger if the user's bank account has more in it).

The first attack cannot be mitigated by HTTP; the ORT mechanism itself needs some transport-layer means of scoping the usability of the first packet on a connection so that it cannot be reused broadly. For example, this might be by time, or by network location.

The second attack is more difficult to mitigate; scoping the usability of the first packet helps, but does not completely prevent the attack. If the replayed request is state-changing, the application's retry detection should kick in and prevent information leakage (since the response will likely contain an error, instead of the desired information).

If it is not (e.g., a GET), the information being targeted is vulnerable as long as both the first packet and the credentials in the request (if any) are valid.

[4.](#) Possible Areas of Work

[4.1.](#) Updating HTTP's Requirements for Retries

The currently language in [[RFC7230](#)] about retries is vague about the conditions under which a request can be retried, leading to significant variance in implementation behaviour. For example, it's been observed that many automated clients fail under circumstances when browsers succeed, because they do not retry in the same way.

As a result, more carefully specifying the conditions under which a request can be retried would be helpful. Such work would need to take into account varying conditions, such as:

- o Connection closes
- o TCP RST

- o Connection timeouts
- o Whether or not any part of the response has been received
- o Whether or not it is the first request on the connection
- o Variance due to use of HTTP/2, TLS/1.3, TCP Fast Open and QUIC.

Furthermore, readers might mistake the language in [RFC7230](#) as guaranteeing that some requests (e.g., POST) are never automatically retried; this should be clarified.

[4.2.](#) Protocol Extensions

A number of mechanisms have been mooted at various times, e.g.:

- o Adding a header to automatically retried requests, to aid de-duplication by servers
- o Defining a request header to be added by intermediaries when they have received a request in a way that could have been replayed
- o Defining a status code to allow servers to indicate that the request needs to be sent in a way that can't be replayed

[4.3.](#) Feedback to Transport 0RTT Efforts

If the observations above hold, we should disabuse any notion that HTTP method idempotency is a useful way to avoid problems with replay attacks. Instead, we should encourage development of mechanisms to mitigate the aspects of replay that are different than retries (e.g., potential for DOS attacks).

[5.](#) Security Considerations

Yep.

[6.](#) Acknowledgements

Thanks to Brad Fitzpatrick, Leif Hedstrom, Subodh Iyengar, Amos Jeffries, Patrick McManus, Matt Menke, Miroslav Ponec, Daniel Stenberg and Martin Thomson for their input and feedback.

Thanks also to the participants in the 2016 HTTP Workshop for their lively discussion of this topic.

[7.](#) References

[7.1.](#) Normative References

- [I-D.ietf-quic-tls]
Thomson, M. and (. (Unknown), "Using Transport Layer Security (TLS) to Secure QUIC", [draft-ietf-quic-tls-01](#) (work in progress), January 2017.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [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, <<http://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, <<http://www.rfc-editor.org/info/rfc7231>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", [RFC 7413](#), DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.

[7.2.](#) Informative References

- [I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", [draft-ietf-quic-transport-01](#) (work in progress), January 2017.
- [I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", [draft-ietf-tls-tls13-18](#) (work in progress), October 2016.

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

Nottingham

Expires August 5, 2017

[Page 10]

Internet-Draft

Retrying HTTP Requests

February 2017

[7.3](#). URIs

- [1] <https://www.iana.org/assignments/http-methods/http-methods.xhtml>
- [2] https://signalvnoise.com/archives2/google_web_accelerator_hey_not_so_fast_an_alert_for_web_app_designers.php
- [3] <http://bazaar.launchpad.net/~squid/squid/trunk/view/head:/src/FwdState.cc#L594>
- [4] <https://git-wip-us.apache.org/repos/asf?p=trafficserver.git;a=blob;f=proxy/http/HttpTransact.cc;h=8a1f5364d47654b118296a07a2a95284f119d84b;hb=HEAD#l6408>
- [5] <https://git-wip-us.apache.org/repos/asf?p=trafficserver.git;a=blob;f=proxy/http/HttpTransact.cc;hb=48d7b25ba8a8229b0471d37cdaa6ef24cc634bb0#l3634>
- [6] <http://mxr.mozilla.org/mozilla-release/source/netwerk/protocol/http/nsHttpTransaction.cpp#938>
- [7] <http://mxr.mozilla.org/mozilla-release/source/netwerk/protocol/http/nsHttpRequestHead.cpp#67>
- [8] <https://www.fxsitecompat.com/en-CA/docs/2016/post-request-fails-on-certain-sites-showing-connection-reset-page/>
- [9] https://chromium.googlesource.com/chromium/src.git/+/master/net/http/http_network_transaction.cc#l657
- [10] <https://github.com/curl/curl/blob/master/lib/transfer.c#L1892>

[Appendix A](#). When Clients Retry

In implementations, clients have been observed to retry requests in a number of circumstances.

Note: This section is intended to inform the discussion, not to be published as a standard. If you have relevant information about these or other implementations (open or closed), please get in touch.

[A.1](#). Squid

Squid is a caching proxy server that retries requests that it considers safe *or* idempotent, as long as there is not a request body:

Nottingham

Expires August 5, 2017

[Page 11]

Internet-Draft

Retrying HTTP Requests

February 2017

```
/// Whether we may try sending this request again after a failure.
```

```
bool
```

```
FwdState::checkRetriable()
```

```
{
```

```
    // Optimize: A compliant proxy may retry PUTs, but Squid lacks the [rather
    // complicated] code required to protect the PUT request body from being
    // nibbled during the first try. Thus, Squid cannot retry some PUTs today.
    if (request->body_pipe != NULL)
        return false;
```

```
    // RFC2616 9.1 Safe and Idempotent Methods
```

```
    return (request->method.isHttpSafe() || request->method.isIdempotent());
```

```
}
```

(source [\[6\]](#))

Currently, it considers GET, HEAD, OPTIONS, REPORT, PROPFIND, SEARCH and PRI to be safe, and GET, HEAD, PUT, DELETE, OPTIONS, TRACE, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, UNLOCK, and PRI to be idempotent.

[A.2](#). Traffic Server

Apache Traffic Server, a caching proxy server, ties retry-ability to whether the request required a "tunnel" - i.e., forwarding the

request body to the next server. This is indicated by "request_body_start", which is set when a POST tunnel is used.

```
// bool HttpTransact::is_request_retryable
//
//   If we started a POST/PUT tunnel then we can
//   not retry failed requests
//
bool
HttpTransact::is_request_retryable(State *s)
{
    if (s->hdr_info.request_body_start == true) {
        return false;
    }

    if (s->state_machine->plugin_tunnel_type != HTTP_NO_PLUGIN_TUNNEL) {
        // API can override
        if (s->state_machine->plugin_tunnel_type == HTTP_PLUGIN_AS_SERVER &&
            s->api_info.retry_intercept_failures == true) {
            // This used to be an == comparison, which made no sense. Changed
            // to be an assignment, hoping the state is correct.

```



```

        s->state_machine->plugin_tunnel_type = HTTP_NO_PLUGIN_TUNNEL;
    } else {
        return false;
    }
}

return true;
}

```

(source [\[7\]](#))

When connected to an origin server, Traffic Server attempts to retry under a number of failure conditions:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name      : handle_response_from_server
// Description: response is from the origin server
//
// Details   :
//
//  response from the origin server. one of three things can happen now.
//  if the response is bad, then we can either retry (by first downgrading
//  the request, maybe making it non-keepalive, etc.), or we can give up.
//  the latter case is handled by handle_server_connection_not_open and
//  sends an error response back to the client. if the response is good
//  handle_forward_server_connection_open is called.
//
//
// Possible Next States From Here:
//

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void
HttpTransact::handle_response_from_server(State *s)
{

[...]

    switch (s->current.state) {
    case CONNECTION_ALIVE:
        DebugTxn("http_trans", "[hrfs] connection alive");
        SET_VIA_STRING(VIA_DETAIL_SERVER_CONNECT, VIA_DETAIL_SERVER_SUCCESS);
    }
}

```



```

s->current.server->clear_connect_fail();
handle_forward_server_connection_open(s);
break;

```

[...]

```

case OPEN_RAW_ERROR:
/* fall through */
case CONNECTION_ERROR:
/* fall through */
case STATE_UNDEFINED:
/* fall through */
case INACTIVE_TIMEOUT:
    // Set to generic I/O error if not already set specifically.
    if (!s->current.server->had_connect_fail())
        s->current.server->set_connect_fail(EIO);

    if (is_server_negative_cached(s)) {
        max_connect_retries = s->txn_conf->connect_attempts_max_retries_dead_serv
    } else {
        // server not yet negative cached - use default number of retries
        max_connect_retries = s->txn_conf->connect_attempts_max_retries;
    }
    if (s->pCongestionEntry != NULL)
        max_connect_retries = s->pCongestionEntry->connect_retries();

    if (is_request_retryable(s) && s->current.attempts < max_connect_retries) {

(source [8])

```

[A.3.](#) Firefox

Firefox is a Web browser that retries under the following conditions:

```

// if the connection was reset or closed before we wrote any part of the
// request or if we wrote the request but didn't receive any part of the
// response and the connection was being reused, then we can (and really
// should) assume that we wrote to a stale connection and we must therefore

```



```
// repeat the request over a new connection.
//
// We have decided to retry not only in case of the reused connections, but
// all safe methods(bug 1236277).
//
// NOTE: the conditions under which we will automatically retry the HTTP
// request have to be carefully selected to avoid duplication of the
// request from the point-of-view of the server. such duplication could
// have dire consequences including repeated purchases, etc.
//
// NOTE: because of the way SSL proxy CONNECT is implemented, it is
// possible that the transaction may have received data without having
// sent any data. for this reason, mSendData == FALSE does not imply
// mReceivedData == FALSE. (see bug 203057 for more info.)
//
[...]
```

```
if (!mReceivedData &&
    ((mRequestHead && mRequestHead->IsSafeMethod()) ||
     !reallySentData || connReused)) {
    // if restarting fails, then we must proceed to close the pipe,
    // which will notify the channel that the transaction failed.
```

(source [\[9\]](#))

... and it considers GET, HEAD, OPTIONS, TRACE, PROPFIND, REPORT, and SEARCH to be safe:


```
bool
nsHttpRequestHead::IsSafeMethod() const
{
    // This code will need to be extended for new safe methods, otherwise
    // they'll default to "not safe".
    if (IsGet() || IsHead() || IsOptions() || IsTrace()) {
        return true;
    }

    if (mParsedMethod != kMethod_Custom) {
        return false;
    }

    return (!strcmp(mMethod.get(), "PROPFIND") ||
            !strcmp(mMethod.get(), "REPORT") ||
            !strcmp(mMethod.get(), "SEARCH"));
}
```

(source [[10](#)])

Note that "connReused" is tested; if a connection has been used before, Firefox will retry *any* request, safe or not. A recent change attempted to remove this behaviour, but it caused compatibility problems [11], and is being backed out.

[A.4.](#) Chromium

Chromium is a Web browser that appears to retry any request when a connection is broken, as long as it's successfully used the connection before, and hasn't received any response headers yet:

```
bool HttpNetworkTransaction::ShouldResendRequest() const {
    bool connection_is_proven = stream_>IsConnectionReused();
    bool has_received_headers = GetResponseHeaders() != NULL;

    // NOTE: we resend a request only if we reused a keep-alive connection.
    // This automatically prevents an infinite resend loop because we'll run
    // out of the cached keep-alive connections eventually.
    if (connection_is_proven && !has_received_headers)
        return true;
    return false;
}
```

(source [[12](#)])

[A.5.](#) Curl

Curl is both a command-line client and widely-used library for HTTP. Like Chromium, it will retry a request if the response hasn't started.

```
CURLcode Curl_retry_request(struct connectdata *conn,
                           char **url)
{
    struct Curl_easy *data = conn->data;

    *url = NULL;

    /* if we're talking upload, we can't do the checks below, unless the protocol
       is HTTP as when uploading over HTTP we will still get a response */
    if(data->set.upload &&
        !(conn->handler->protocol&(PROTO_FAMILY_HTTP|CURLPROTO_RTSP)))
        return CURLE_OK;

    if((data->req.bytecount + data->req.headerbytecount == 0) &&
        conn->bits.reuse &&
        (data->set.rtspreq != RTSPREQ_RECEIVE)) {
        /* We didn't get a single byte when we attempted to re-use a
           connection. This might happen if the connection was left alive when we
           were done using it before, but that was closed when we wanted to use it
           again. Bad luck. Retry the same request on a fresh connect! */
        infof(conn->data, "Connection died, retrying a fresh connect\n");
        *url = strdup(conn->data->change.url);
        if(!*url)
            return CURLE_OUT_OF_MEMORY;

        connclose(conn, "retry"); /* close this connection */
        conn->bits.retry = TRUE; /* mark this as a connection we're about
                                to retry. Marking it this way should
                                prevent i.e HTTP transfers to return
                                error just because nothing has been
                                transferred! */

        if(conn->handler->protocol&PROTO_FAMILY_HTTP) {
```



```
    struct HTTP *http = data->req.protop;
    if(http->writebytecount)
        return Curl_readrewind(conn);
    }
}
return CURLE_OK;
}
```

Nottingham

Expires August 5, 2017

[Page 17]

Internet-Draft

Retrying HTTP Requests

February 2017

(source [13])

Author's Address

Mark Nottingham

Email: mnot@mnot.net

URI: <https://www.mnot.net/>

Nottingham

Expires August 5, 2017

[Page 18]