lsquic Documentation

Release 2.16.2

LiteSpeed Technologies

Jun 12, 2020

Contents

This is the documentation for LSQUIC 2.16.2, last updated Jun 12, 2020.

LiteSpeed QUIC (LSQUIC) Library is an open-source implementation of QUIC and HTTP/3 functionality for servers and clients. LSQUIC is:

- fast;
- flexible; and
- production-ready.

Most of the code in this distribution has been used in our own products – LiteSpeed Web Server, LiteSpeed Web ADC, and OpenLiteSpeed – since 2017.

Currently supported QUIC versions are Q043, Q046, Q050, ID-27, and ID-28. Support for newer versions will be added soon after they are released.

CHAPTER 1

Contents

1.1 Getting Started

1.1.1 Supported Platforms

LSQUIC compiles and runs on Linux, Windows, FreeBSD, Mac OS, and Android. It has been tested on i386, x86_64, and ARM (Raspberry Pi and Android).

1.1.2 Dependencies

LSQUIC library uses:

- zlib;
- BoringSSL; and
- ls-hpack (as a Git submodule).
- ls-qpack (as a Git submodule).

The accompanying demo command-line tools use libevent.

1.1.3 What's in the box

- `src/liblsquic` – the library
- `bin` – demo client and server programs
- `tests` – unit tests

1.1.4 Building

To build the library, follow instructions in the README file.

1.1.5 Demo Examples

Fetch Google home page:

```
./http_client -s www.google.com -p / -o version=Q050
```

Run your own server (it does not touch the filesystem, don't worry):

```
./http_server -c www.example.com,fullchain.pem,privkey.pem -s 0.0.0.0:4433
```

Grab a page from your server:

```
./http_client -H www.example.com -s 127.0.0.1:4433 -p /
```

You can play with various options, of which there are many. Use the `-h` command-line flag to see them.

1.1.6 Next steps

If you want to use LSQUIC in your program, check out the *Tutorial* and the *API Reference*.

Internals covers some library internals.

1.2 Tutorial

1.2.1 Introduction

The LSQUIC library provides facilities for operating a QUIC (Google QUIC or IETF QUIC) server or client with optional HTTP (or HTTP/3) functionality. To do that, it specifies an application programming interface (API) and exposes several basic object types to operate upon:

- engine;
- connection; and
- stream.

An engine manages connections, processes incoming packets, and schedules outgoing packets. An engine operates in one of two modes: client or server.

The LSQUIC library does not use sockets to receive and send packets; that is handled by the user-supplied callbacks. The library also does not mandate the use of any particular event loop. Instead, it has functions to help the user schedule events. (Thus, using an event loop is not even strictly necessary.) The various callbacks and settings are supplied to the engine constructor.

A connection carries one or more streams, ensures reliable data delivery, and handles the protocol details.

A stream usually corresponds to a request/response pair: a client sends its request over a single stream and a server sends its response back using the same stream. This is the Google QUIC and HTTP/3 use case. Nevertheless, the library does not limit one to this scenario. Any application protocol can be implemented using LSQUIC – as long as it can be implemented using the QUIC transport protocol. The library provides hooks for stream events: when a stream is created or closed, when it has data to read or when it can be written to, and so on.

In the following sections, we will describe how to:

- initialize the library;
- configure and instantiate an engine object;

- send and receive packets; and
- work with connections and streams.

Include Files

A single include file, `lsquic.h`, contains all the necessary LSQUIC declarations:

```
#include <lsquic.h>
```

1.2.2 Library Initialization

Before the first engine object is instantiate, the library must be initialized using `lsquic_global_init()`:

```
if (0 != lsquic_global_init(LSQUIC_GLOBAL_CLIENT|LSQUIC_GLOBAL_SERVER))
{
    exit(EXIT_FAILURE);
}
/* OK, do something useful */
```

If you plan to instantiate engines only in a single mode, client or server, you can omit the appropriate flag.

After all engines have been destroyed and the LSQUIC library is no longer going to be used, the global initialization can be undone:

```
lsquic_global_cleanup();
exit(EXIT_SUCCESS);
```

1.2.3 Engine Instantiation

Engine instantiation is performed by *lsquic_engine_new()*:

```
/* Create an engine in server mode with HTTP behavior: */
lsquic_engine_t *engine
    = lsquic_engine_new(LSENG_SERVER|LSENG_HTTP, &engine_api);
```

The engine mode is selected by using the `LSENG_SERVER` flag. If present, the engine will be in server mode; if not, the engine will be in client mode.

Using the `LSENG_HTTP` flag enables the HTTP behavior: The library hides the interaction between the HTTP application layer and the QUIC transport layer and presents a simple, unified (between Google QUIC and HTTP/3) way of sending and receiving HTTP messages. Behind the scenes, the library will compress and uncompress HTTP headers, add and remove HTTP/3 stream framing, and operate the necessary control streams.

Engine Configuration

The second argument to *lsquic_engine_new()* is a pointer to a struct of type *lsquic_engine_api*. This structure lists several user-specified function pointers that the engine is to use to perform various functions. Mandatory among these are:

- function to set packets out, *lsquic_engine_api.ea_packets_out*;
- functions linked to connection and stream events, *lsquic_engine_api.ea_stream_if*;
- function to look up certificate to use, *lsquic_engine_api.ea_lookup_cert* (in server mode); and

- function to fetch SSL context, *lsquic_engine_api.ea_get_ssl_ctx* (in server mode).

The minimal structure for a client will look like this:

```
lsquic_engine_api engine_api = {
    .ea_packets_out     = send_packets_out,
    .ea_packets_out_ctx = (void *) sockfd,  /* For example */
    .ea_stream_if       = &stream_callbacks,
    .ea_stream_if_ctx   = &some_context,
};
```

Engine Settings

Engine settings can be changed by specifying *lsquic_engine_api.ea_settings*. There are **many** parameters to tweak: supported QUIC versions, amount of memory dedicated to connections and streams, various timeout values, and so on. See *Engine Settings* for full details. If `ea_settings` is set to `NULL`, the engine will use the defaults, which should be OK.

1.2.4 Sending Packets

The *lsquic_engine_api.ea_packets_out* is the function that gets called when an engine instance has packets to send. It could look like this:

```
/* Return number of packets sent or -1 on error */
static int
send_packets_out (void *ctx, const struct lsquic_out_spec *specs,
                                                    unsigned n_specs)
{
    struct msghdr msg;
    int sockfd;
    unsigned n;

    memset(&msg, 0, sizeof(msg));
    sockfd = (int) (uintptr_t) ctx;

    for (n = 0; n < n_specs; ++n)
    {
        msg.msg_name       = (void *) specs[n].dest_sa;
        msg.msg_namelen    = sizeof(struct sockaddr_in);
        msg.msg_iov        = specs[n].iov;
        msg.msg_iovlen     = specs[n].iovlen;
        if (sendmsg(sockfd, &msg, 0) < 0)
            break;
    }

    return (int) n;
}
```

Note that the version above is very simple. *lsquic_out_spec* also specifies local address as well as ECN value. These are set using ancillary data in a platform-dependent way.

1.2.5 Receiving Packets

The user reads packets and provides them to an engine instance using *lsquic_engine_packet_in()*.

TODO

1.2.6 Running Connections

A connection needs to be processed once in a while. It needs to be processed when one of the following is true:

- There are incoming packets;
- A stream is both readable by the user code and the user code wants to read from it;
- A stream is both writeable by the user code and the user code wants to write to it;
- User has written to stream outside of on_write() callbacks (that is allowed) and now there are packets ready to be sent;
- A timer (pacer, retransmission, idle, etc) has expired;
- A control frame needs to be sent out;
- A stream needs to be serviced or created.

Each of these use cases is handled by a single function, *lsquic_engine_process_conns()*.

The connections to which the conditions above apply are processed (or "ticked") in the least recently ticked order. After calling this function, you can see when is the next time a connection needs to be processed using *lsquic_engine_earliest_adv_tick()*.

Based on this value, next event can be scheduled (in the event loop of your choice).

1.2.7 Stream Reading and Writing

Reading from (or writing to) a stream is best down when that stream is readable (or writeable). To register an interest in an event,

1.3 API Reference

1.3.1 Preliminaries

All declarations are in `lsquic.h`, so it is enough to

```
#incluide <lsquic.h>
```

in each source file.

1.3.2 Library Version

LSQUIC follows the following versioning model. The version number has the form MAJOR.MINOR.PATCH, where

- MAJOR changes when a large redesign occurs;
- MINOR changes when an API change or another significant change occurs; and
- PATCH changes when a bug is fixed or another small, API-compatible change occurs.

1.3.3 QUIC Versions

LSQUIC supports two types of QUIC protocol: Google QUIC and IETF QUIC. The former will at some point become obsolete, while the latter is still being developed by the IETF. Both types are included in a single enum:

enum **lsquic_version**

LSQVER_043
Google QUIC version Q043

LSQVER_046
Google QUIC version Q046

LSQVER_050
Google QUIC version Q050

LSQVER_ID27
IETF QUIC version ID (Internet-Draft) 27

LSQVER_ID28
IETF QUIC version ID 28

N_LSQVER
Special value indicating the number of versions in the enum. It may be used as argument to *lsquic_engine_connect()*.

Several version lists (as bitmasks) are defined in `lsquic.h`:

LSQUIC_SUPPORTED_VERSIONS

List of all supported versions.

LSQUIC_FORCED_TCID0_VERSIONS

List of versions in which the server never includes CID in short packets.

LSQUIC_EXPERIMENTAL_VERSIONS

Experimental versions.

Deprecated versions.

LSQUIC_GQUIC_HEADER_VERSIONS

Versions that have Google QUIC-like headers. Only Q043 remains in this list.

LSQUIC_IETF_VERSIONS

IETF QUIC versions.

LSQUIC_IETF_DRAFT_VERSIONS

IETF QUIC *draft* versions. When IETF QUIC v1 is released, it will not be included in this list.

1.3.4 LSQUIC Types

LSQUIC declares several types used by many of its public functions. They are:

lsquic_engine_t
Instance of LSQUIC engine.

lsquic_conn_t
QUIC connection.

lsquic_stream_t
QUIC stream.

lsquic_stream_id_t
Stream ID.

lsquic_conn_ctx_t
Connection context. This is the return value of `on_new_conn()`. To LSQUIC, this is just an opaque pointer. User code is expected to use it for its own purposes.

lsquic_stream_ctx_t
Stream context. This is the return value of `on_new_stream()`. To LSQUIC, this is just an opaque pointer. User code is expected to use it for its own purposes.

lsquic_http_headers_t
HTTP headers

1.3.5 Library Initialization

Before using the library, internal structures must be initialized using the global initialization function:

```
if (0 == lsquic_global_init(LSQUIC_GLOBAL_CLIENT|LSQUIC_GLOBAL_SERVER))
    /* OK, do something useful */
    ;
```

This call only needs to be made once. Afterwards, any number of LSQUIC engines may be instantiated.

After a process is done using LSQUIC, it should clean up:

```
lsquic_global_cleanup();
```

1.3.6 Logging

struct **lsquic_logger_if**

int **(*log_buf)** (void *logger_ctx, const char *buf, size_t len)

void **lsquic_logger_init** (const struct lsquic_logger_if *logger_if, void *logger_ctx, enum lsquic_logger_timestamp_style)
Call this if you want to do something with LSQUIC log messages, as they are thrown out by default.

int **lsquic_set_log_level** (const char *log_level)
Set log level for all LSQUIC modules.

Parameters

- **log_level** – Acceptable values are debug, info, notice, warning, error, alert, emerg, crit (case-insensitive).

Returns 0 on success or -1 on failure (invalid log level).

int **lsquic_logger_lopt** (const char *log_specs)
Set log level for a particular module or several modules.

Parameters

- **log_specs** – One or more "module=level" specifications serapated by comma. For example, "event=debug,engine=info". See *List of Log Modules*

1.3.7 Engine Instantiation and Destruction

To use the library, an instance of the `struct lsquic_engine` needs to be created:

lsquic_engine_t ***lsquic_engine_new** (unsigned *flags*, const struct *lsquic_engine_api* **api*)
Create a new engine.

Parameters

- **flags** – This is is a bitmask of `LSENG_SERVER`` and `LSENG_HTTP`.
- **api** – Pointer to an initialized *`lsquic_engine_api`*.

The engine can be instantiated either in server mode (when `LSENG_SERVER` is set) or client mode. If you need both server and client in your program, create two engines (or as many as you'd like).

Specifying `LSENG_HTTP` flag enables the HTTP functionality: HTTP/2-like for Google QUIC connections and HTTP/3 functionality for IETF QUIC connections.

void **lsquic_engine_cooldown** (*lsquic_engine_t* **engine*)
This function closes all mini connections and marks all full connections as going away. In server mode, this also causes the engine to stop creating new connections.

void **lsquic_engine_destroy** (*lsquic_engine_t* **engine*)
Destroy engine and all its resources.

1.3.8 Engine Callbacks

`struct lsquic_engine_api` contains a few mandatory members and several optional members.

struct **lsquic_engine_api**

const struct *lsquic_stream_if* ***ea_stream_if**

void ***ea_stream_if_ctx**
`ea_stream_if` is mandatory. This structure contains pointers to callbacks that handle connections and stream events.

lsquic_packets_out_f **ea_packets_out**

void ***ea_packets_out_ctx**
`ea_packets_out` is used by the engine to send packets.

const struct *lsquic_engine_settings* ***ea_settings**
If `ea_settings` is set to NULL, the engine uses default settings (see *`lsquic_engine_init_settings()`*)

lsquic_lookup_cert_f **ea_lookup_cert**

void ***ea_cert_lu_ctx**
Look up certificate. Mandatory in server mode.

struct ssl_ctx_st * **(*ea_get_ssl_ctx)** (void **peer_ctx*)
Get SSL_CTX associated with a peer context. Mandatory in server mode. This is use for default values for SSL instantiation.

const struct *lsquic_hset_if* ***ea_hsi_if**

void ***ea_hsi_ctx**
Optional header set interface. If not specified, the incoming headers are converted to HTTP/1.x format and are read from stream and have to be parsed again.

const struct *lsquic_shared_hash_if* ***ea_shi**

void ***ea_shi_ctx**
Shared hash interface can be used to share state between several processes of a single QUIC server.

const struct *lsquic_packout_mem_if* ***ea_pmi**

void ***ea_pmi_ctx**
Optional set of functions to manage memory allocation for outgoing packets.

lsquic_cids_update_f **ea_new_scids**

lsquic_cids_update_f **ea_live_scids**

lsquic_cids_update_f **ea_old_scids**

void ***ea_cids_update_ctx**
In a multi-process setup, it may be useful to observe the CID lifecycle. This optional set of callbacks makes it possible.

1.3.9 Engine Settings

Engine behavior can be controlled by several settings specified in the settings structure:

struct **lsquic_engine_settings**

unsigned **es_versions**
This is a bit mask wherein each bit corresponds to a value in *lsquic_version*. Client starts negotiating with the highest version and goes down. Server supports either of the versions specified here. This setting applies to both Google and IETF QUIC.

The default value is *LSQUIC_DF_VERSIONS*.

unsigned **es_cfcw**
Initial default connection flow control window.

In server mode, per-connection values may be set lower than this if resources are scarce.

Do not set es_cfcw and es_sfcw lower than *LSQUIC_MIN_FCW*.

unsigned **es_sfcw**
Initial default stream flow control window.

In server mode, per-connection values may be set lower than this if resources are scarce.

Do not set es_cfcw and es_sfcw lower than *LSQUIC_MIN_FCW*.

unsigned **es_max_cfcw**
This value is used to specify maximum allowed value connection flow control window is allowed to reach due to window auto-tuning. By default, this value is zero, which means that CFCW is not allowed to increase from its initial value.

unsigned **es_max_sfcw**
This value is used to specify maximum allowed value stream flow control window is allowed to reach due to window auto-tuning. By default, this value is zero, which means that CFCW is not allowed to increase from its initial value.

unsigned **es_max_streams_in**
Maximum incoming streams, a.k.a. MIDS.

Google QUIC only.

unsigned long **es_handshake_to**
Handshake timeout in microseconds.

For client, this can be set to an arbitrary value (zero turns the timeout off).

For server, this value is limited to about 16 seconds. Do not set it to zero.

Defaults to *LSQUIC_DF_HANDSHAKE_TO*.

unsigned long **es_idle_conn_to**
Idle connection timeout, a.k.a ICSL, in microseconds; GQUIC only.

Defaults to *LSQUIC_DF_IDLE_CONN_TO*

int **es_silent_close**
SCLS (silent close)

unsigned **es_max_header_list_size**
This corresponds to SETTINGS_MAX_HEADER_LIST_SIZE (**RFC 7540#section-6.5.2**). 0 means no limit. Defaults to *LSQUIC_DF_MAX_HEADER_LIST_SIZE()*.

const char ***es_ua**

UAID – User-Agent ID. Defaults to *LSQUIC_DF_UA*.

Google QUIC only.

More parameters for server

unsigned **es_max_inchoate**
Maximum number of incoming connections in inchoate state. (In other words, maximum number of mini connections.)

This is only applicable in server mode.

Defaults to *LSQUIC_DF_MAX_INCHOATE*.

int **es_support_push**
Setting this value to 0 means that

For client:

1. we send a SETTINGS frame to indicate that we do not support server push; and
2. all incoming pushed streams get reset immediately.

(For maximum effect, set es_max_streams_in to 0.)

For server:

1. *lsquic_conn_push_stream()* will return -1.

int **es_support_tcid0**
If set to true value, the server will not include connection ID in outgoing packets if client's CHLO specifies TCID=0.

For client, this means including TCID=0 into CHLO message. Note that in this case, the engine tracks connections by the (source-addr, dest-addr) tuple, thereby making it necessary to create a socket for each connection.

This option has no effect in Q046, as the server never includes CIDs in the short packets.

The default is *LSQUIC_DF_SUPPORT_TCID0()*.

int **es_support_nstp**
Q037 and higher support "No STOP_WAITING frame" mode. When set, the client will send NSTP option in its Client Hello message and will not sent STOP_WAITING frames, while ignoring incoming

STOP_WAITING frames, if any. Note that if the version negotiation happens to downgrade the client below Q037, this mode will *not* be used.

This option does not affect the server, as it must support NSTP mode if it was specified by the client.

Defaults to *LSQUIC_DF_SUPPORT_NSTP*.

int **es_honor_prst**
If set to true value, the library will drop connections when it receives corresponding Public Reset packet. The default is to ignore these packets.

int **es_send_prst**
If set to true value, the library will send Public Reset packets in response to incoming packets with unknown Connection IDs.

The default is *LSQUIC_DF_SEND_PRST*.

unsigned **es_progress_check**
A non-zero value enables internal checks that identify suspected infinite loops in user `on_read()` and `on_write()` callbacks and break them. An infinite loop may occur if user code keeps on performing the same operation without checking status, e.g. reading from a closed stream etc.

The value of this parameter is as follows: should a callback return this number of times in a row without making progress (that is, reading, writing, or changing stream state), loop break will occur.

The defaut value is *LSQUIC_DF_PROGRESS_CHECK*.

int **es_rw_once**
A non-zero value make stream dispatch its read-write events once per call.

When zero, read and write events are dispatched until the stream is no longer readable or writeable, respectively, or until the user signals unwillingness to read or write using *lsquic_stream_wantread()* or *lsquic_stream_wantwrite()* or shuts down the stream.

The default value is *LSQUIC_DF_RW_ONCE*.

unsigned **es_proc_time_thresh**
If set, this value specifies that number of microseconds that *lsquic_engine_process_conns()* and *lsquic_engine_send_unsent_packets()* are allowed to spend before returning.

This is not an exact science and the connections must make progress, so the deadline is checked after all connections get a chance to tick (in the case of `lsquic_engine_process_conns())()` and at least one batch of packets is sent out.

When processing function runs out of its time slice, immediate calls to *lsquic_engine_has_unsent_packets()* return false.

The default value is *LSQUIC_DF_PROC_TIME_THRESH()*.

int **es_pace_packets**
If set to true, packet pacing is implemented per connection.

The default value is *LSQUIC_DF_PACE_PACKETS()*.

unsigned **es_clock_granularity**
Clock granularity information is used by the pacer. The value is in microseconds; default is *LSQUIC_DF_CLOCK_GRANULARITY()*.

unsigned **es_init_max_data**
Initial max data.

This is a transport parameter.

Depending on the engine mode, the default value is either *LSQUIC_DF_INIT_MAX_DATA_CLIENT* or *LSQUIC_DF_INIT_MAX_DATA_SERVER*.

IETF QUIC only.

unsigned **es_init_max_stream_data_bidi_remote**
Initial max stream data.

This is a transport parameter.

Depending on the engine mode, the default value is either *LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_REMOTE_CL* or *LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_REMOTE_SERVER*.

IETF QUIC only.

unsigned **es_init_max_stream_data_bidi_local**
Initial max stream data.

This is a transport parameter.

Depending on the engine mode, the default value is either *LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_LOCAL_CLI* or *LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_LOCAL_SERVER*.

IETF QUIC only.

unsigned **es_init_max_stream_data_uni**
Initial max stream data for unidirectional streams initiated by remote endpoint.

This is a transport parameter.

Depending on the engine mode, the default value is either *LSQUIC_DF_INIT_MAX_STREAM_DATA_UNI_CLIENT* or *LSQUIC_DF_INIT_MAX_STREAM_DATA_UNI_SERVER*.

IETF QUIC only.

unsigned **es_init_max_streams_bidi**
Maximum initial number of bidirectional stream.

This is a transport parameter.

Default value is *LSQUIC_DF_INIT_MAX_STREAMS_BIDI*.

IETF QUIC only.

unsigned **es_init_max_streams_uni**
Maximum initial number of unidirectional stream.

This is a transport parameter.

Default value is *LSQUIC_DF_INIT_MAX_STREAMS_UNI_CLIENT* or *LSQUIC_DF_INIT_MAX_STREAM_DATA_UNI_SERVER*.

IETF QUIC only.

unsigned **es_idle_timeout**
Idle connection timeout.

This is a transport parameter.

(Note: `es_idle_conn_to()` is not reused because it is in microseconds, which, I now realize, was not a good choice. Since it will be obsoleted some time after the switchover to IETF QUIC, we do not have to keep on using strange units.)

Default value is *LSQUIC_DF_IDLE_TIMEOUT*.

Maximum value is 600 seconds.

IETF QUIC only.

unsigned **es_ping_period**
Ping period. If set to non-zero value, the connection will generate and send PING frames in the absence of other activity.

By default, the server does not send PINGs and the period is set to zero. The client's defaut value is *LSQUIC_DF_PING_PERIOD*.

IETF QUIC only.

unsigned **es_scid_len**
Source Connection ID length. Valid values are 0 through 20, inclusive.

Default value is `LSQUIC_DF_SCID_LEN`.

IETF QUIC only.

unsigned **es_scid_iss_rate**
Source Connection ID issuance rate. This field is measured in CIDs per minute. Using value 0 indicates that there is no rate limit for CID issuance.

Default value is *LSQUIC_DF_SCID_ISS_RATE*.

IETF QUIC only.

unsigned **es_qpack_dec_max_size**
Maximum size of the QPACK dynamic table that the QPACK decoder will use.

The default is *LSQUIC_DF_QPACK_DEC_MAX_SIZE*.

IETF QUIC only.

unsigned **es_qpack_dec_max_blocked**
Maximum number of blocked streams that the QPACK decoder is willing to tolerate.

The default is *LSQUIC_DF_QPACK_DEC_MAX_BLOCKED*.

IETF QUIC only.

unsigned **es_qpack_enc_max_size**
Maximum size of the dynamic table that the encoder is willing to use. The actual size of the dynamic table will not exceed the minimum of this value and the value advertized by peer.

The default is *LSQUIC_DF_QPACK_ENC_MAX_SIZE*.

IETF QUIC only.

unsigned **es_qpack_enc_max_blocked**
Maximum number of blocked streams that the QPACK encoder is willing to risk. The actual number of blocked streams will not exceed the minimum of this value and the value advertized by peer.

The default is *LSQUIC_DF_QPACK_ENC_MAX_BLOCKED*.

IETF QUIC only.

int **es_ecn**
Enable ECN support.

The default is *LSQUIC_DF_ECN*

IETF QUIC only.

int **es_allow_migration**
Allow peer to migrate connection.

The default is *LSQUIC_DF_ALLOW_MIGRATION*

IETF QUIC only.

unsigned **es_cc_algo**
Congestion control algorithm to use.

- 0: Use default (`LSQUIC_DF_CC_ALGO`)
- 1: Cubic
- 2: BBR

IETF QUIC only.

int **es_ql_bits**
Use QL loss bits. Allowed values are:

- 0: Do not use loss bits
- 1: Allow loss bits
- 2: Allow and send loss bits

Default value is *LSQUIC_DF_QL_BITS*

int **es_spin**
Enable spin bit. Allowed values are 0 and 1.

Default value is *LSQUIC_DF_SPIN*

int **es_delayed_acks**
Enable delayed ACKs extension. Allowed values are 0 and 1.

Warning: this is an experimental feature. Using it will most likely lead to degraded performance.

Default value is *LSQUIC_DF_DELAYED_ACKS*

int **es_timestamps**
Enable timestamps extension. Allowed values are 0 and 1.

Default value is @ref LSQUIC_DF_TIMESTAMPS

unsigned short **es_max_udp_payload_size_rx**
Maximum packet size we are willing to receive. This is sent to peer in transport parameters: the library does not enforce this limit for incoming packets.

If set to zero, limit is not set.

Default value is *LSQUIC_DF_MAX_UDP_PAYLOAD_SIZE_RX*

unsigned **es_noprogress_timeout**
No progress timeout.

If connection does not make progress for this number of seconds, the connection is dropped. Here, progress is defined as user streams being written to or read from.

If this value is zero, this timeout is disabled.

Default value is *LSQUIC_DF_NOPROGRESS_TIMEOUT_SERVER* in server mode and *LSQUIC_DF_NOPROGRESS_TIMEOUT_CLIENT* in client mode.

To initialize the settings structure to library defaults, use the following convenience function:

lsquic_engine_init_settings (struct *lsquic_engine_settings* *, unsigned *flags*)
`flags` is a bitmask of `LSENG_SERVER` and `LSENG_HTTP`

After doing this, change just the settings you'd like. To check whether the values are correct, another convenience function is provided:

lsquic_engine_check_settings (const struct *lsquic_engine_settings* *, unsigned *flags*, char **err_buf*, size_t *err_buf_sz*)

Check settings for errors. Return 0 if settings are OK, -1 otherwise.

If `err_buf()` and `err_buf_sz()` are set, an error string is written to the buffers.

The following macros in `lsquic.h` specify default values:

Note that, despite our best efforts, documentation may accidentally get out of date. Please check your :file:'lsquic.h' for actual values.

LSQUIC_MIN_FCW
Minimum flow control window is set to 16 KB for both client and server. This means we can send up to this amount of data before handshake gets completed.

LSQUIC_DF_VERSIONS
By default, deprecated and experimental versions are not included.

LSQUIC_DF_CFCW_SERVER

LSQUIC_DF_CFCW_CLIENT

LSQUIC_DF_SFCW_SERVER

LSQUIC_DF_SFCW_CLIENT

LSQUIC_DF_MAX_STREAMS_IN

LSQUIC_DF_INIT_MAX_DATA_SERVER

LSQUIC_DF_INIT_MAX_DATA_CLIENT

LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_REMOTE_SERVER

LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_LOCAL_SERVER

LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_REMOTE_CLIENT

LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_LOCAL_CLIENT

LSQUIC_DF_INIT_MAX_STREAMS_BIDI

LSQUIC_DF_INIT_MAX_STREAMS_UNI_CLIENT

LSQUIC_DF_INIT_MAX_STREAMS_UNI_SERVER

LSQUIC_DF_INIT_MAX_STREAM_DATA_UNI_CLIENT

LSQUIC_DF_INIT_MAX_STREAM_DATA_UNI_SERVER

LSQUIC_DF_IDLE_TIMEOUT
Default idle connection timeout is 30 seconds.

LSQUIC_DF_PING_PERIOD
Default ping period is 15 seconds.

LSQUIC_DF_HANDSHAKE_TO
Default handshake timeout is 10,000,000 microseconds (10 seconds).

LSQUIC_DF_IDLE_CONN_TO
Default idle connection timeout is 30,000,000 microseconds.

LSQUIC_DF_SILENT_CLOSE
By default, connections are closed silenty when they time out (no CONNECTION_CLOSE frame is sent).

LSQUIC_DF_MAX_HEADER_LIST_SIZE
Default value of maximum header list size. If set to non-zero value, SETTINGS_MAX_HEADER_LIST_SIZE will be sent to peer after handshake is completed (assuming the peer supports this setting frame type).

LSQUIC_DF_UA
Default value of UAID (user-agent ID).

LSQUIC_DF_MAX_INCHOATE
Default is 1,000,000.

LSQUIC_DF_SUPPORT_NSTP
NSTP is not used by default.

LSQUIC_DF_SUPPORT_PUSH
Push promises are supported by default.

LSQUIC_DF_SUPPORT_TCID0
Support for TCID=0 is enabled by default.

LSQUIC_DF_HONOR_PRST
By default, LSQUIC ignores Public Reset packets.

LSQUIC_DF_SEND_PRST
By default, LSQUIC will not send Public Reset packets in response to packets that specify unknown connections.

LSQUIC_DF_PROGRESS_CHECK
By default, infinite loop checks are turned on.

LSQUIC_DF_RW_ONCE
By default, read/write events are dispatched in a loop.

LSQUIC_DF_PROC_TIME_THRESH
By default, the threshold is not enabled.

LSQUIC_DF_PACE_PACKETS
By default, packets are paced

LSQUIC_DF_CLOCK_GRANULARITY
Default clock granularity is 1000 microseconds.

LSQUIC_DF_SCID_LEN **8**
The default value is 8 for simplicity and speed.

LSQUIC_DF_SCID_ISS_RATE
The default value is 60 CIDs per minute.

LSQUIC_DF_QPACK_DEC_MAX_BLOCKED
Default value is 100.

LSQUIC_DF_QPACK_DEC_MAX_SIZE
Default value is 4,096 bytes.

LSQUIC_DF_QPACK_ENC_MAX_BLOCKED
Default value is 100.

LSQUIC_DF_QPACK_ENC_MAX_SIZE
Default value is 4,096 bytes.

LSQUIC_DF_ECN
ECN is disabled by default.

LSQUIC_DF_ALLOW_MIGRATION
Allow migration by default.

LSQUIC_DF_QL_BITS
Use QL loss bits by default.

LSQUIC_DF_SPIN
Turn spin bit on by default.

LSQUIC_DF_CC_ALGO
Use Cubic by default.

LSQUIC_DF_DELAYED_ACKS
Delayed ACKs are off by default.

LSQUIC_DF_MAX_UDP_PAYLOAD_SIZE_RX
By default, incoming packet size is not limited.

LSQUIC_DF_NOPROGRESS_TIMEOUT_SERVER
By default, drop no-progress connections after 60 seconds on the server.

LSQUIC_DF_NOPROGRESS_TIMEOUT_CLIENT
By default, do not use no-progress timeout on the client.

1.3.10 Receiving Packets

Incoming packets are supplied to the engine using `lsquic_engine_packet_in()`. It is up to the engine to decide what do to with the packet. It can find an existing connection and dispatch the packet there, create a new connection (in server mode), or schedule a version negotiation or stateless reset packet.

int **lsquic_engine_packet_in**(*lsquic_engine_t* **engine*, const unsigned char **data*, size_t *size*, const struct sockaddr **local*, const struct sockaddr **peer*, void **peer_ctx*, int *ecn*)
Pass incoming packet to the QUIC engine. This function can be called more than once in a row. After you add one or more packets, call `lsquic_engine_process_conns()` to schedule outgoing packets, if any.

Parameters

- **engine** – Engine instance.
- **data** – Pointer to UDP datagram payload.
- **size** – Size of UDP datagram.
- **local** – Local address.
- **peer** – Peer address.
- **peer_ctx** – Peer context.
- **ecn** – ECN marking associated with this UDP datagram.

Returns

- `0`: Packet was processed by a real connection.
- `1`: Packet was handled successfully, but not by a connection. This may happen with version negotiation and public reset packets as well as some packets that may be ignored.
- `-1`: Some error occurred. Possible reasons are invalid packet size or failure to allocate memory.

int **lsquic_engine_earliest_adv_tick**(*lsquic_engine_t* **engine*, int **diff*)
Returns true if there are connections to be processed, false otherwise.

Parameters

- **engine** – Engine instance.

- **diff** – If the function returns a true value, the pointed to integer is set to the difference between the earliest advisory tick time and now. If the former is in the past, this difference is negative.

Returns True if there are connections to be processed, false otherwise.

1.3.11 Sending Packets

User specifies a callback `lsquic_packets_out_f` in *lsquic_engine_api* that the library uses to send packets.

struct **lsquic_out_spec**
This structure describes an outgoing packet.

struct iovec ***iov**
A vector with payload.

size_t **iovlen**
Vector length.

const struct sockaddr ***local_sa**
Local address.

const struct sockaddr ***dest_sa**
Destination address.

void ***peer_ctx**
Peer context associated with the local address.

int **ecn**
ECN: Valid values are 0 - 3. See **RFC 3168**.

ECN may be set by IETF QUIC connections if `es_ecn` is set.

void **lsquic_engine_process_conns** (*lsquic_engine_t *engine*)
Process tickable connections. This function must be called often enough so that packets and connections do not expire. The preferred method of doing so is by using *lsquic_engine_earliest_adv_tick()*.

int **lsquic_engine_has_unsent_packets** (*lsquic_engine_t *engine*)
Returns true if engine has some unsent packets. This happens if `ea_packets_out()` could not send everything out.

void **lsquic_engine_send_unsent_packets** (*lsquic_engine_t *engine*)
Send out as many unsent packets as possibe: until we are out of unsent packets or until `ea_packets_out()` fails.

If `ea_packets_out()` cannot send all packets, this function must be called to signify that sending of packets is possible again.

1.3.12 Stream Callback Interface

The stream callback interface structure lists the callbacks used by the engine to communicate with the user code:

struct **lsquic_stream_if**

```
lsquic_conn_ctx_t *(*on_new_conn)(void *stream_if_ctx,
```

lsquic_conn_t *);
Called when a new connection has been created. In server mode, this means that the handshake has been successful. In client mode, on the other hand, this callback is called as soon as connection object is created inside the engine, but before the handshake is done.

The return value is the connection context associated with this connection. Use *lsquic_conn_get_ctx()* to get back this context. It is OK for this function to return NULL.

This callback is mandatory.

void **(*on_conn_closed)** (*lsquic_conn_t* *)
Connection is closed.

This callback is mandatory.

lsquic_stream_ctx_t * **(*on_new_stream)** (void *stream_if_ctx, *lsquic_stream_t* *)
If you need to initiate a connection, call lsquic_conn_make_stream(). This will cause `on_new_stream()` callback to be called when appropriate (this operation is delayed when maximum number of outgoing streams is reached).

If connection is going away, this callback may be called with the second parameter set to NULL.

The return value is the stream context associated with the stream. A pointer to it is passed to `on_read()`, `on_write()`, and `on_close()` callbacks. It is OK for this function to return NULL.

This callback is mandatory.

void **(*on_read)** (*lsquic_stream_t* *s, *lsquic_stream_ctx_t* *h)
Stream is readable: either there are bytes to be read or an error is ready to be collected.

This callback is mandatory.

void **(*on_write)** (*lsquic_stream_t* *s, *lsquic_stream_ctx_t* *h)
Stream is writeable.

This callback is mandatory.

void **(*on_close)** (*lsquic_stream_t* *s, *lsquic_stream_ctx_t* *h)
After this callback returns, the stream is no longer accessible. This is a good time to clean up the stream context.

This callback is mandatory.

void **(*on_hsk_done)** (*lsquic_conn_t* *c, enum lsquic_hsk_status *s*)
When handshake is completed, this callback is called.

This callback is optional.

void **(*on_goaway_received)** (*lsquic_conn_t* *)
This is called when our side received GOAWAY frame. After this, new streams should not be created.

This callback is optional.

void **(*on_new_token)** (*lsquic_conn_t* *c, const unsigned char *token, size_t *token_size*)
When client receives a token in NEW_TOKEN frame, this callback is called.

This callback is optional.

void **(*on_zero_rtt_info)** (*lsquic_conn_t* *c, const unsigned char *, size_t)
This callback lets client record information needed to perform a zero-RTT handshake next time around.

This callback is optional.

1.3.13 Creating Connections

In server mode, the connections are created by the library based on incoming packets. After handshake is completed, the library calls `on_new_conn()` callback.

In client mode, a new connection is created by

lsquic_conn_t * **lsquic_engine_connect** (*lsquic_engine_t* **engine*, enum *lsquic_version* *version*, const struct sockaddr **local_sa*, const struct sockaddr **peer_sa*, void **peer_ctx*, *lsquic_conn_ctx_t* **conn_ctx*, const char **sni*, unsigned short *max_udp_payload_size*, const unsigned char **zero_rtt*, size_t *zero_rtt_len*, const unsigned char **token*, size_t *token_sz*)

Parameters

- **engine** – Engine to use.
- **version** – To let the engine specify QUIC version, use N_LSQVER. If zero-rtt info is supplied, version is picked from there instead.
- **local_sa** – Local address.
- **peer_sa** – Address of the server.
- **peer_ctx** – Context associated with the connection. This is what gets passed to TODO.
- **conn_ctx** – Connection context can be set early using this parameter. Useful if you need the connection context to be available in `on_conn_new()`. Note that that callback's return value replaces the connection context set here.
- **sni** – The SNI is required for Google QUIC connections; it is optional for IETF QUIC and may be set to NULL.
- **max_udp_payload_size** – Maximum packet size. If set to zero, it is inferred based on `peer_sa()` and `version()`.
- **zero_rtt** – Pointer to previously saved zero-RTT data needed for TLS resumption. May be NULL.
- **zero_rtt_len** – Size of zero-RTT data.
- **token** – Pointer to previously received token to include in the Initial packet. Tokens are used by IETF QUIC to pre-validate client connections, potentially avoiding a retry.

  See `on_new_token` callback in *lsquic_stream_if*:

  May be NULL.
- **token_sz** – Size of data pointed to by `token`.

1.3.14 Closing Connections

void **lsquic_conn_going_away** (*lsquic_conn_t* **conn*)

Mark connection as going away: send GOAWAY frame and do not accept any more incoming streams, nor generate streams of our own.

Only applicable to HTTP/3 and GQUIC connections. Otherwise a no-op.

void **lsquic_conn_close** (*lsquic_conn_t* **conn*)

This closes the connection. `on_conn_closed()` and `on_close()` callbacks will be called.

1.3.15 Creating Streams

Similar to connections, streams are created by the library in server mode; they correspond to requests. In client mode, a new stream is created by

void **lsquic_conn_make_stream**(*lsquic_conn_t* *)

Create a new request stream. This causes `on_new_stream()` callback to be called. If creating more requests is not permitted at the moment (due to number of concurrent streams limit), stream creation is registered as "pending" and the stream is created later when number of streams dips under the limit again. Any number of pending streams can be created. Use *`lsquic_conn_n_pending_streams()`* and *`lsquic_conn_cancel_pending_streams()`* to manage pending streams.

If connection is going away, `on_new_stream()` is called with the stream parameter set to NULL.

1.3.16 Stream Events

To register or unregister an interest in a read or write event, use the following functions:

int **lsquic_stream_wantread**(*lsquic_stream_t* *stream, int *want*)

Parameters

- **stream** – Stream to read from.
- **want** – Boolean value indicating whether the caller wants to read from stream.

Returns Previous value of `want` or `-1` if the stream has already been closed for reading.

A stream becomes readable if there is was an error: for example, the peer may have reset the stream. In this case, reading from the stream will return an error.

int **lsquic_stream_wantwrite**(*lsquic_stream_t* *stream, int *want*)

Parameters

- **stream** – Stream to write to.
- **want** – Boolean value indicating whether the caller wants to write to stream.

Returns Previous value of `want` or `-1` if the stream has already been closed for writing.

1.3.17 Reading From Streams

ssize_t **lsquic_stream_read**(*lsquic_stream_t* *stream, unsigned char *buf, size_t *sz*)

Parameters

- **stream** – Stream to read from.
- **buf** – Buffer to copy data to.
- **sz** – Size of the buffer.

Returns Number of bytes read, zero if EOS has been reached, or -1 on error.

Read up to `sz` bytes from `stream` into buffer `buf`.

`-1` is returned on error, in which case `errno` is set:

- `EBADF`: The stream is closed.
- `ECONNRESET`: The stream has been reset.
- `EWOULDBLOCK`: There is no data to be read.

ssize_t **lsquic_stream_readv** (*lsquic_stream_t* **stream*, const struct iovec **vec*, int *iovcnt*)

Parameters

- **stream** – Stream to read from.
- **vec** – Array of `iovec` structures.
- **iovcnt** – Number of elements in `vec`.

Returns Number of bytes read, zero if EOS has been reached, or -1 on error.

Similar to *lsquic_stream_read()*, but reads data into a vector.

ssize_t **lsquic_stream_readf** (*lsquic_stream_t* **stream*, size_t (**readf*)(void *ctx, const unsigned char **buf*, size_t len, int fin), void **ctx*)

Parameters

- **stream** – Stream to read from.
- **readf** – The callback takes four parameters:
  - Pointer to user-supplied context;
  - Pointer to the data;
  - Data size (can be zero); and
  - Indicator whether the FIN follows the data.

  The callback returns number of bytes processed. If this number is zero or is smaller than `len`, reading from stream stops.
- **ctx** – Context pointer passed to `readf`.

This function allows user-supplied callback to read the stream contents. It is meant to be used for zero-copy stream processing.

Return value and errors are same as in *lsquic_stream_read()*.

1.3.18 Writing To Streams

ssize_t **lsquic_stream_write** (*lsquic_stream_t* **stream*, const void **buf*, size_t *len*)

Parameters

- **stream** – Stream to write to.
- **buf** – Buffer to copy data from.
- **len** – Number of bytes to copy.

Returns Number of bytes written – which may be smaller than `len` – or a negative value when an error occurs.

Write `len` bytes to the stream. Returns number of bytes written, which may be smaller that `len`.

A negative return value indicates a serious error (the library is likely to have aborted the connection because of it).

ssize_t **lsquic_stream_writev** (*lsquic_stream_t* **s*, const struct iovec **vec*, int *count*)
Like *lsquic_stream_write()*, but read data from a vector.

struct **lsquic_reader**
Used as argument to *lsquic_stream_writef()*.

size_t **(*lsqr_read)** (void *lsqr_ctx*, void *buf*, size_t *count*)

Parameters

- **lsqr_ctx** – Pointer to user-specified context.
- **buf** – Memory location to write to.
- **count** – Size of available memory pointed to by `buf`.

Returns Number of bytes written. This is not a `ssize_t` because the read function is not supposed to return an error. If an error occurs in the read function (for example, when reading from a file fails), it is supposed to deal with the error itself.

size_t **(*lsqr_size)** (void *lsqr_ctx*)
Return number of bytes remaining in the reader.

void ***lsqr_ctx**
Context pointer passed both to `lsqr_read()` and to `lsqr_size()`.

ssize_t **lsquic_stream_writef** (*lsquic_stream_t* *stream, struct *lsquic_reader* *reader)

Parameters

- **stream** – Stream to write to.
- **reader** – Reader to read from.

Returns Number of bytes written or -1 on error.

Write to stream using *`lsquic_reader`*. This is the most generic of the write functions – *`lsquic_stream_write()`* and *`lsquic_stream_writev()`* utilize the same mechanism.

int **lsquic_stream_flush** (*lsquic_stream_t* *stream)

Parameters

- **stream** – Stream to flush.

Returns 0 on success and -1 on failure.

Flush any buffered data. This triggers packetizing even a single byte into a separate frame. Flushing a closed stream is an error.

1.3.19 Closing Streams

Streams can be closed for reading, writing, or both. `on_close()` callback is called at some point after a stream is closed for both reading and writing,

int **lsquic_stream_shutdown** (*lsquic_stream_t* *stream, int *how*)

Parameters

- **stream** – Stream to shut down.
- **how** – This parameter specifies what do to. Allowed values are:
  - 0: Stop reading.
  - 1: Stop writing.
  - 2: Stop both reading and writing.

Returns 0 on success or -1 on failure.

int **lsquic_stream_close** (*lsquic_stream_t* *stream)

Parameters

- **stream** – Stream to close.

Returns 0 on success or -1 on failure.

1.3.20 Sending HTTP Headers

struct **lsxpack_header**

This type is defined in _lsxpack_header.h_. See that header file for more information.

char ***buf**
the buffer for headers

const char ***name_ptr**
the name pointer can be optionally set for encoding

uint32_t **name_hash**
hash value for name

uint32_t **nameval_hash**
hash value for name + value

lsxpack_strlen_t **name_offset**
the offset for name in the buffer

lsxpack_strlen_t **name_len**
the length of name

lsxpack_strlen_t **val_offset**
the offset for value in the buffer

lsxpack_strlen_t **val_len**
the length of value

uint16_t **chain_next_idx**
mainly for cookie value chain

uint8_t **hpack_index**
HPACK static table index

uint8_t **qpack_index**
QPACK static table index

uint8_t **app_index**
APP header index

enum lsxpack_flag **flags:8**
combination of lsxpack_flag

uint8_t **indexed_type**
control to disable index or not

uint8_t **dec_overhead**
num of extra bytes written to decoded buffer

lsquic_http_headers_t

int **count**
Number of headers in `headers`.

struct *lsxpack_header* ***headers**
Pointer to an array of HTTP headers.

HTTP header list structure. Contains a list of HTTP headers.

int **lsquic_stream_send_headers**(*lsquic_stream_t* *stream, const *lsquic_http_headers_t* *headers, int *eos*)

Parameters

- **stream** – Stream to send headers on.
- **headers** – Headers to send.
- **eos** – Boolean value to indicate whether these headers constitute the whole HTTP message.

Returns 0 on success or -1 on error.

1.3.21 Receiving HTTP Headers

If `ea_hsi_if` is not set in *lsquic_engine_api*, the library will translate HPACK- and QPACK-encoded headers into HTTP/1.x-like headers and prepend them to the stream. To the stream-reading function, it will look as if a standard HTTP/1.x message.

Alternatively, you can specify header-processing set of functions and manage header fields yourself. In that case, the header set must be "read" from the stream via *lsquic_stream_get_hset()*.

struct **lsquic_hset_if**

void * **(*hsi_create_header_set)**(void *hsi_ctx, *lsquic_stream_t* *stream, int *is_push_promise*)

Parameters

- **hsi_ctx** – User context. This is the pointer specifed in `ea_hsi_ctx`.
- **stream** – Stream with which the header set is associated. May be set to NULL in server mode.
- **is_push_promise** – Boolean value indicating whether this header set is for a push promise.

Returns Pointer to user-defined header set object.

Create a new header set. This object is (and must be) fetched from a stream by calling *lsquic_stream_get_hset()* before the stream can be read.

struct *lsxpack_header* * **(*hsi_prepare_decode)**(void *hdr_set, struct *lsxpack_header* *hdr, size_t *space*)

Return a header set prepared for decoding. If `hdr` is NULL, this means return a new structure with at least `space` bytes available in the decoder buffer. On success, a newly prepared header is returned.

If `hdr` is not NULL, it means there was not enough decoder buffer and it must be increased to at least `space` bytes. `buf`, `val_len`, and `name_offset` member of the `hdr` structure may change. On success, the return value is the same as `hdr`.

If NULL is returned, the space cannot be allocated.

int **(*hsi_process_header)**(void *hdr_set, struct *lsxpack_header* *hdr)
Process new header.

Parameters

- **hdr_set** – Header set to add the new header field to. This is the object returned by `hsi_create_header_set()`.
- **hdr** – The header returned by @ref `hsi_prepare_decode()`.

Returns Return 0 on success, a positive value if a header error occured, or a negative value on any other error. A positive return value will result in cancellation of associated stream. A negative return value will result in connection being aborted.

void **(*hsi_discard_header_set)** (void **hdr_set*)

Parameters

- **hdr_set** – Header set to discard.

Discard header set. This is called for unclaimed header sets and header sets that had an error.

enum *lsquic_hsi_flag* **hsi_flags**

These flags specify properties of decoded headers passed to `hsi_process_header()`. This is only applicable to QPACK headers; HPACK library header properties are based on compilation, not run-time, options.

void * **lsquic_stream_get_hset** (*lsquic_stream_t *stream*)

Parameters

- **stream** – Stream to fetch header set from.

Returns Header set associated with the stream.

Get header set associated with the stream. The header set is created by `hsi_create_header_set()` callback. After this call, the ownership of the header set is transferred to the caller.

This call must precede calls to *`lsquic_stream_read()`*, *`lsquic_stream_readv()`*, and *`lsquic_stream_readf()`*.

If the optional header set interface is not specified, this function returns NULL.

1.3.22 Push Promises

int **lsquic_conn_push_stream** (*lsquic_conn_t *conn*, void **hdr_set*, *lsquic_stream_t *stream*, const *lsquic_http_headers_t *headers*)

Returns

- 0: Stream pushed successfully.
- **1: Stream push failed because it is disabled or because we hit** stream limit or connection is going away.
- -1: Stream push failed because of an internal error.

A server may push a stream. This call creates a new stream in reference to stream `stream`. It will behave as if the client made a request: it will trigger `on_new_stream()` event and it can be used as a regular client-initiated stream.

`hdr_set` must be set. It is passed as-is to *`lsquic_stream_get_hset()`*.

int **lsquic_conn_is_push_enabled** (*lsquic_conn_t *conn*)

Returns Boolean value indicating whether push promises are enabled.

Only makes sense in server mode: the client cannot push a stream and this function always returns false in client mode.

int **lsquic_stream_refuse_push**(*lsquic_stream_t *stream*)
Refuse pushed stream. Call it from `on_new_stream()`. No need to call *`lsquic_stream_close()`* after this. `on_close()` will be called.

int **lsquic_stream_push_info**(const *lsquic_stream_t *stream*, *lsquic_stream_id_t *ref_stream_id*, void ***hdr_set*)
Get information associated with pushed stream

Parameters

- **ref_stream_id** – Stream ID in response to which push promise was sent.
- **hdr_set** – Header set. This object was passed to or generated by *`lsquic_conn_push_stream()`*.

Returns 0 on success and -1 if this is not a pushed stream.

1.3.23 Stream Priorities

unsigned **lsquic_stream_priority**(const *lsquic_stream_t *stream*)
Return current priority of the stream.

int **lsquic_stream_set_priority**(*lsquic_stream_t *stream*, unsigned *priority*)
Set stream priority. Valid priority values are 1 through 256, inclusive.

Returns 0 on success of -1 on failure (this happens if priority value is invalid).

1.3.24 Miscellaneous Engine Functions

unsigned **lsquic_engine_quic_versions**(const *lsquic_engine_t *engine*)
Return the list of QUIC versions (as bitmask) this engine instance supports.

unsigned **lsquic_engine_count_attq**(*lsquic_engine_t *engine*, int *from_now*)
Return number of connections whose advisory tick time is before current time plus `from_now` microseconds from now. `from_now` can be negative.

1.3.25 Miscellaneous Connection Functions

enum *lsquic_version* **lsquic_conn_quic_version**(const *lsquic_conn_t *conn*)
Get QUIC version used by the connection.

If version has not yet been negotiated (can happen in client mode), `-1` is returned.

const lsquic_cid_t * **lsquic_conn_id**(const *lsquic_conn_t *conn*)
Get connection ID.

lsquic_engine_t * **lsquic_conn_get_engine**(*lsquic_conn_t *conn*)
Get pointer to the engine.

int **lsquic_conn_get_sockaddr**(*lsquic_conn_t *conn*, const struct sockaddr ***local*, const struct sockaddr ***peer*)
Get current (last used) addresses associated with the current path used by the connection.

struct stack_st_X509 * **lsquic_conn_get_server_cert_chain**(*lsquic_conn_t *conn*)
Get certificate chain returned by the server. This can be used for server certificate verification.

The caller releases the stack using sk_X509_free().

lsquic_conn_ctx_t * **lsquic_conn_get_ctx** (const *lsquic_conn_t* **conn*)
Get user-supplied context associated with the connection.

void **lsquic_conn_set_ctx** (*lsquic_conn_t* **conn*, *lsquic_conn_ctx_t* **ctx*)
Set user-supplied context associated with the connection.

void * **lsquic_conn_get_peer_ctx** (*lsquic_conn_t* **conn*, const struct sockaddr **local_sa*)
Get peer context associated with the connection and local address.

enum *LSQUIC_CONN_STATUS* **lsquic_conn_status** (*lsquic_conn_t* **conn*, char **errbuf*, size_t *bufsz*)
Get connection status.

1.3.26 Miscellaneous Stream Functions

unsigned **lsquic_conn_n_avail_streams** (const *lsquic_conn_t* **conn*)
Return max allowed outbound streams less current outbound streams.

unsigned **lsquic_conn_n_pending_streams** (const *lsquic_conn_t* **conn*)
Return number of delayed streams currently pending.

unsigned **lsquic_conn_cancel_pending_streams** (*lsquic_conn_t* *, unsigned *n*)
Cancel `n` pending streams. Returns new number of pending streams.

lsquic_conn_t * **lsquic_stream_conn** (const *lsquic_stream_t* **stream*)
Get a pointer to the connection object. Use it with connection functions.

int **lsquic_stream_is_rejected** (const *lsquic_stream_t* **stream*)
Returns true if this stream was rejected, false otherwise. Use this as an aid to distinguish between errors.

1.3.27 Other Functions

enum *lsquic_version* **lsquic_str2ver** (const char **str*, size_t *len*)
Translate string QUIC version to LSQUIC QUIC version representation.

enum *lsquic_version* **lsquic_alpn2ver** (const char **alpn*, size_t *len*)
Translate ALPN (e.g. “h3”, “h3-23”, “h3-Q046”) to LSQUIC enum.

1.3.28 Miscellaneous Types

struct **lsquic_shared_hash_if**
The shared hash interface is used to share data between multiple LSQUIC instances.

int **(*shi_insert)** (void **shi_ctx*, void **key*, unsigned *key_sz*, void **data*, unsigned *data_sz*, time_t *expiry*)

Parameters

- **shi_ctx** – Shared memory context pointer
- **key** – Key data.
- **key_sz** – Key size.
- **data** – Pointer to the data to store.
- **data_sz** – Data size.
- **expiry** – When this item expires. If you want your item to never expire, set this to zero.

Returns 0 on success, -1 on failure.

If inserted successfully, `free()` will be called on `data` and `key` pointer when the element is deleted, whether due to expiration or explicit deletion.

int **(*shi_delete)**(void **shi_ctx*, const void **key*, unsigned *key_sz*)
Delete item from shared hash

Returns 0 on success, -1 on failure.

int **(*shi_lookup)**(void **shi_ctx*, const void **key*, unsigned *key_sz*, void ***data*, unsigned **data_sz*)

Parameters

- **shi_ctx** – Shared memory context pointer
- **key** – Key data.
- **key_sz** – Key size.
- **data** – Pointer to set to the result.
- **data_sz** – Pointer to the data size.

Returns

- `1`: found.
- `0`: not found.
- `-1`: error (perhaps not enough room in `data` if copy was attempted).

The implementation may choose to copy the object into buffer pointed to by `data`, so you should have it ready.

struct **lsquic_packout_mem_if**
The packet out memory interface is used by LSQUIC to get buffers to which outgoing packets will be written before they are passed to *lsquic_engine_api.ea_packets_out* callback.

If not specified, malloc() and free() are used.

void * **(*pmi_allocate)**(void **pmi_ctx*, void **conn_ctx*, unsigned short *sz*, char *is_ipv6*)
Allocate buffer for sending.

void **(*pmi_release)**(void **pmi_ctx*, void **conn_ctx*, void **buf*, char *is_ipv6*)
This function is used to release the allocated buffer after it is sent via `ea_packets_out()`.

void **(*pmi_return)**(void **pmi_ctx*, void **conn_ctx*, void **buf*, char *is_ipv6*)
If allocated buffer is not going to be sent, return it to the caller using this function.

typedef void **(*lsquic_cids_update_f)**(void **ctx*, void ***peer_ctx*, const lsquic_cid_t **cids*, unsigned *n_cids*)

Parameters

- **ctx** – Context associated with the CID lifecycle callbacks (ea_cids_update_ctx).
- **peer_ctx** – Array of peer context pointers.
- **cids** – Array of connection IDs.
- **n_cids** – Number of elements in the peer context pointer and connection ID arrays.

struct **lsquic_keylog_if**
SSL keylog interface.

void * **(*kli_open)**(void **keylog_ctx*, *lsquic_conn_t* **conn*)
Return keylog handle or NULL if no key logging is desired.

void **(*kli_log_line)**(void *handle, const char *line)
Log line. The first argument is the pointer returned by `kli_open()`.

void **(*kli_close)**(void *handle)
Close handle.

enum **lsquic_logger_timestamp_style**
Enumerate timestamp styles supported by LSQUIC logger mechanism.

LLTS_NONE
No timestamp is generated.

LLTS_HHMMSSMS
The timestamp consists of 24 hours, minutes, seconds, and milliseconds. Example: 13:43:46.671

LLTS_YYYYMMDD_HHMMSSMS
Like above, plus date, e.g: 2017-03-21 13:43:46.671

LLTS_CHROMELIKE
This is Chrome-like timestamp used by proto-quic. The timestamp includes month, date, hours, minutes, seconds, and microseconds.

Example: 1223/104613.946956 (instead of 12/23 10:46:13.946956).

This is to facilitate reading two logs side-by-side.

LLTS_HHMMSSUS
The timestamp consists of 24 hours, minutes, seconds, and microseconds. Example: 13:43:46.671123

LLTS_YYYYMMDD_HHMMSSUS
Date and time using microsecond resolution, e.g: 2017-03-21 13:43:46.671123

enum **LSQUIC_CONN_STATUS**

LSCONN_ST_HSK_IN_PROGRESS

LSCONN_ST_CONNECTED

LSCONN_ST_HSK_FAILURE

LSCONN_ST_GOING_AWAY

LSCONN_ST_TIMED_OUT

LSCONN_ST_RESET
If es_honor_prst is not set, the connection will never get public reset packets and this flag will not be set.

LSCONN_ST_USER_ABORTED

LSCONN_ST_ERROR

LSCONN_ST_CLOSED

LSCONN_ST_PEER_GOING_AWAY

enum **lsquic_hsi_flag**
These flags are ORed together to specify properties of *lsxpack_header* passed to *lsquic_hset_if.hsi_process_header*.

LSQUIC_HSI_HTTP1X
Turn HTTP/1.x mode on or off. In this mode, decoded name and value pair are separated by `": "` and `"\r\n"` is appended to the end of the string. By default, this mode is off.

LSQUIC_HSI_HASH_NAME
Include name hash into lsxpack_header.

LSQUIC_HSI_HASH_NAMEVAL
Include nameval hash into lsxpack_header.

1.3.29 Global Variables

const char *const lsquic_ver2str[N_LSQVER]
Convert LSQUIC version to human-readable string

1.3.30 List of Log Modules

The following log modules are defined:

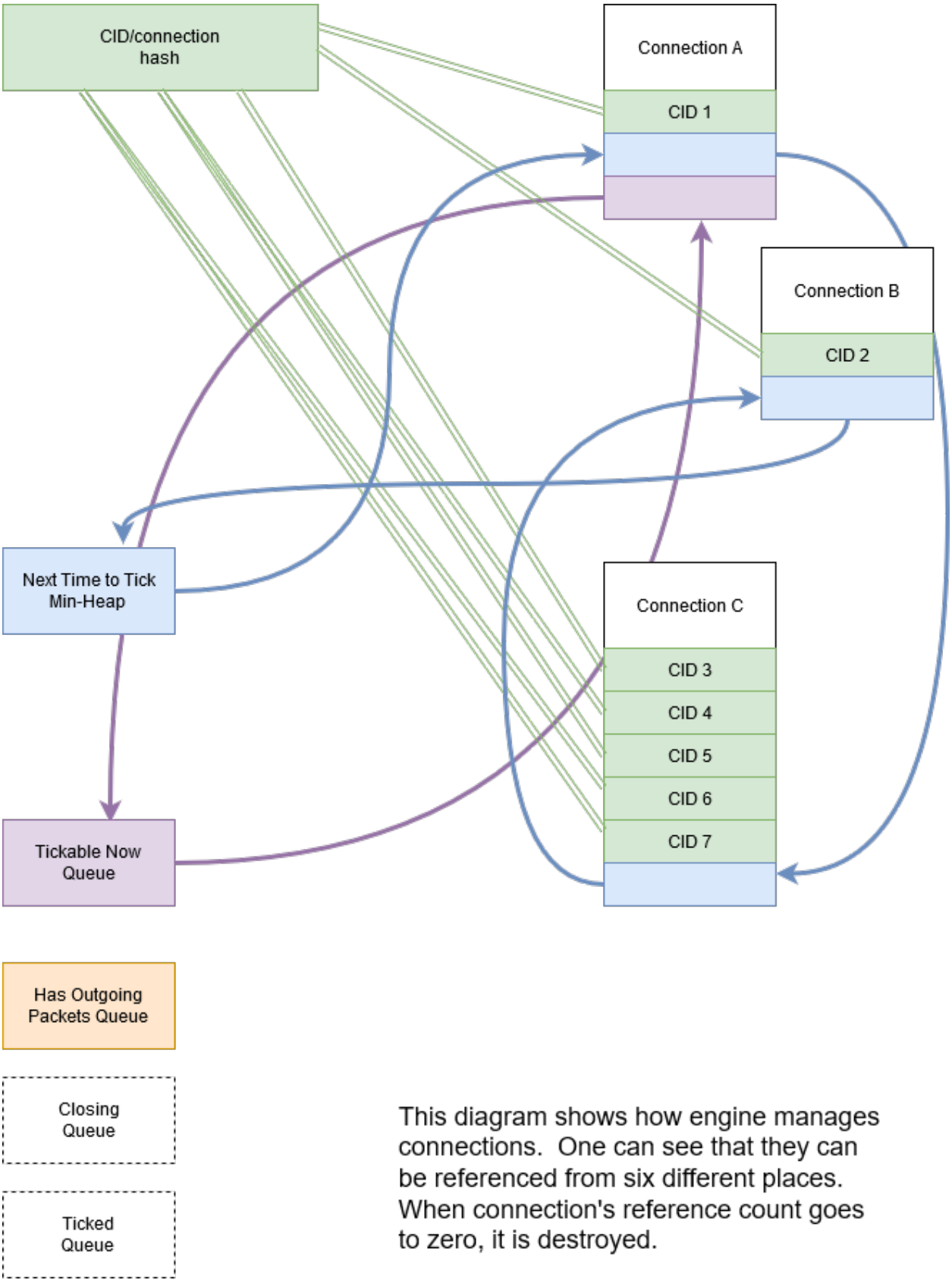
- *alarmset*: Alarm processing.
- *bbr*: BBR congestion controller.
- *bw-sampler*: Bandwidth sampler (used by BBR).
- *cfcw*: Connection flow control window.
- *conn*: Connection.
- *crypto*: Low-level Google QUIC cryptography tracing.
- *cubic*: Cubic congestion controller.
- *di*: "Data In" handler (storing incoming data before it is read).
- *eng-hist*: Engine history.
- *engine*: Engine.
- *event*: Cross-module significant events.
- *frame-reader*: Reader of the HEADERS stream in Google QUIC.
- *frame-writer*: Writer of the HEADERS stream in Google QUIC.
- *handshake*: Handshake and packet encryption and decryption.
- *hcsi-reader*: Reader of the HTTP/3 control stream.
- *hcso-writer*: Writer of the HTTP/3 control stream.
- *headers*: HEADERS stream (Google QUIC).
- *hsk-adapter*:
- *http1x*: Header conversion to HTTP/1.x.
- *logger*: Logger.
- *mini-conn*: Mini connection.
- *pacer*: Pacer.
- *parse*: Parsing.
- *prq*: PRQ stands for Packet Request Queue. This logs scheduling and sending packets not associated with a connection: version negotiation and stateless resets.
- *purga*: CID purgatory.
- *qdec-hdl*: QPACK decoder stream handler.
- *qenc-hdl*: QPACK encoder stream handler.

- *qlog*: QLOG output. At the moment, it is out of date.
- *qpack-dec*: QPACK decoder.
- *qpack-enc*: QPACK encoder.
- *rechist*: Receive history.
- *sendctl*: Send controller.
- *sfcw*: Stream flow control window.
- *spi*: Stream priority iterator.
- *stream*: Stream operation.
- *tokgen*: Token generation and validation.
- *trapa*: Transport parameter processing.

1.4 Internals

1.4.1 Connection Management

References to connections can exist in six different places in an engine.


CID/connection hash
Connection A
CID 1
Connection B
CID 2
Connection C
CID 3
CID 4
CID 5
CID 6
CID 7
Next Time to Tick Min-Heap
Tickable Now Queue
Has Outgoing Packets Queue
Closing Queue
Ticked Queue
This diagram shows how engine manages connections. One can see that they can be referenced from six different places. When connection's reference count goes to zero, it is destroyed.

CHAPTER 2

Indices and tables

- genindex
- search

Index

Symbols

A

B

C

D

F

H

I

L

N

Q

R

V