# lsquic Documentation

***Release 2.20.0***

**LiteSpeed Technologies**

**Sep 15, 2020**

# Contents

This is the documentation for LSQUIC 2.20.0, last updated Sep 15, 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, ID-28, ID-29, and ID-30. Support for newer versions will be added soon after they are released.

CHAPTER 1

# Features

LSQUIC supports nearly all QUIC and HTTP/3 features, including

- DPLPMTUD
- ECN
- Spin bits (allowing network observer to calculate a connection's RTT)
- Path migration
- NAT rebinding
- Push promises
- TLS Key updates
- Extensions:
- *Datagrams*
- Loss bits extension (allowing network observer to locate source of packet loss)
- Timestamps extension (allowing for one-way delay calculation, improving performance of some congestion controllers)
- Delayed ACKs (this reduces number of ACK frames sent and processed, improving throughput)
- QUIC grease bit to reduce ossification opportunities

CHAPTER 2

# Architecture

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. LSQUIC keeps QUIC connections in several data structures in order to process them efficiently. Connections that need processing are kept in two priority queues: one holds connections that are ready to be processed (or "ticked") and the other orders connections by their next timer value. As a result, no connection is processed needlessly.

CHAPTER 3

Contents

## 3.1 Getting Started

### 3.1.1 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).

### 3.1.2 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.

### 3.1.3 What's in the box

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

### 3.1.4 Building

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

### 3.1.5 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.

### 3.1.6 Next steps

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

*Internals* covers some library internals.

## 3.2 Tutorial

### 3.2.1 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.

#### Engine

An engine manages connections, processes incoming packets, and schedules outgoing packets. It can be instantiated in either server or client mode. If your program needs to have both QUIC client and server functionality, instantiate two engines. (This is what we do in our LiteSpeed ADC server.) In addition, HTTP mode can be turned on for gQUIC and HTTP/3 support.

#### Connection

A connection carries one or more streams, ensures reliable data delivery, and handles the protocol details. In client mode, a connection is created using a function call, which we will cover later in the tutorial. In server mode, by the time the user code gets a hold of the connection object, the handshake has already been completed successfully. This is not the case in client mode.

#### Stream

A connection can have several streams in parallel and many streams during its lifetime. Streams do not exist by themselves; they belong to a connection. Streams are bidirectional and usually correspond to a request/response exchange - depending on the application protocol. Application data is transmitted over streams.

#### HTTP Mode

The HTTP support is included directly into LSQUIC. The library hides the interaction between the HTTP application layer and the QUIC transport layer and presents a simple, unified way of sending and receiving HTTP messages. (By "unified way," we mean between Google QUIC and HTTP/3). Behind the scenes, the library will compress and decompress HTTP headers, add and remove HTTP/3 stream framing, and operate the necessary control streams.

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

In your source files, you need to include a single header, "lsquic.h". It pulls in an auxiliary file "lsquic_types.h".

```
#include "lsquic.h"
```

### 3.2.2 Library Initialization

Before the first engine object is instantiated, 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 */
```

This will initialize the crypto library, gQUIC server certificate cache, and, depending on the platform, monotonic timers. 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);
```

### 3.2.3 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. If you need both server and client functionality in your program, instantiate two engines (or as many as you like).

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.

### 3.2.4 Receiving Packets

UDP datagrams are passed to the engine using the *lsquic_engine_packet_in()* function. This is the only way to do so. A pointer to the UDP payload is passed along with the size of the payload. Local and peer socket addresses are passed in as well. The void "peer ctx" pointer is associated with the peer address. It gets passed to the function that sends outgoing packets and to a few other callbacks. In a standard setup, this is most likely the socket file descriptor, but it could be pointing to something else. The ECN value is in the range of 0 through 3, as in RFC 3168.

```
/*  0: processed by real connection
 *  1: handled
 * -1: error: invalid arguments, malloc failure
```

(continues on next page)

(continued from previous page)

```
 */
int
lsquic_engine_packet_in (lsquic_engine_t *,
    const unsigned char *udp_payload, size_t sz,
    const struct sockaddr *sa_local,
    const struct sockaddr *sa_peer,
    void *peer_ctx, int ecn);
```

#### Why specify local address

The local address is necessary because it becomes the source address of the outgoing packets. This is important in a multihomed configuration, when packets arriving at a socket can have different destination addresses. Changes in local and peer addresses are also used to detect changes in paths, such as path migration during the classic "parking lot" scenario or NAT rebinding. When path change is detected, QUIC connection performs special steps to validate the new path.

### 3.2.5 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: it does not use local address and ECN value specified in *lsquic_out_spec*. These can be set using ancillary data in a platform-dependent way.

#### When an error occurs

When an error occurs, the value of `errno` is examined:

- `EAGAIN` (or `EWOULDBLOCK`) means that the packets could not be sent and to retry later. It is up to the caller to call *`lsquic_engine_send_unsent_packets()`* when sending can resume.
- `EMSGSIZE` means that a packet was too large. This occurs when lsquic send MTU probes. In that case, the engine will retry sending without the offending packet immediately.
- Any other error causes the connection whose packet could not be sent to be terminated.

#### Outgoing Packet Specification

```
struct lsquic_out_spec
{
    struct iovec          *iov;
    size_t                 iovlen;
    const struct sockaddr *local_sa;
    const struct sockaddr *dest_sa;
    void                  *peer_ctx;
    int                    ecn; /* 0 - 3; see RFC 3168 */
};
```

Each packet specification in the array given to the "packets out" function looks like this. In addition to the packet payload, specified via an iovec, the specification contains local and remote addresses, the peer context associated with the connection (which is just a file descriptor in tut.c), and ECN. The reason for using iovec in the specification is that a UDP datagram may contain several QUIC packets. QUIC packets with long headers, which are used during QUIC handshake, can be coalesced and lsquic tries to do that to reduce the number of datagrams needed to be sent. On the incoming side, *`lsquic_engine_packet_in()`* takes care of splitting incoming UDP datagrams into individual packets.

### 3.2.6 When to process connections

Now that we covered how to initialize the library, instantiate an engine, and send and receive packets, it is time to see how to make the engine tick. "LSQUIC" has the concept of "tick," which is a way to describe a connection doing something productive. Other verbs could have been "kick," "prod," "poke," and so on, but we settled on "tick."

There are several ways for a connection to do something productive. When a connection can do any of these things, it is "tickable:"

- There are incoming packets to process
- A user wants to read from a stream and there is data that can be read
- A user wants to write to a stream and the stream is writeable
- A stream has buffered packets generated when a user has written to stream outside of the regular callback mechanism. (This is allowed as an optimization: sometimes data becomes available and it's faster to just write to stream than to buffer it in the user code and wait for the "on write" callback.)
- Internal QUIC protocol or LSQUIC maintenance actions need to be taken, such as sending out a control frame or recycling a stream.

```
/* Returns true if there are connections to be processed, in
 * which case `diff' is set to microseconds from current time.
 */
int
lsquic_engine_earliest_adv_tick (lsquic_engine_t *, int *diff);
```

There is a single function, *lsquic_engine_earliest_adv_tick()*, that can tell the user whether and when there is at least one connection managed by an engine that needs to be ticked. "Adv" in the name of the function stands for "advisory," meaning that you do not have to process connections at that exact moment; it is simply recommended. If there is a connection to be ticked, the function will return a true value and `diff` will be set to a relative time to when the connection is to be ticked. This value may be negative, which means that the best time to tick the connection has passed. The engine keeps all connections in several data structures. It tracks each connection's timers and knows when it needs to fire.

#### Example with libev

```
void
process_conns (struct tut *tut)
{
    ev_tstamp timeout;
    int diff;
    ev_timer_stop();
    lsquic_engine_process_conns(engine);
    if (lsquic_engine_earliest_adv_tick(engine, &diff) {
        if (diff > 0)
            timeout = (ev_tstamp) diff / 1000000;   /* To seconds */
        else
            timeout = 0.;
        ev_timer_init(timeout)
        ev_timer_start();
    }
}
```

Here is a simple example that uses the libev library. First, we stop the timer and process connections. Then, we query the engine to tell us when the next advisory tick time is. Based on that, we calculate the timeout to reinitialize the timer with and start the timer. If `diff` is negative, we set timeout to zero. When the timer expires (not shown here), it simply calls this `process_conns()` again.

Note that one could ignore the advisory tick time and simply process connections every few milliseconds and it will still work. This, however, will result in worse performance.

#### Processing Connections

Recap: To process connections, call *lsquic_engine_process_conns()*. This will call necessary callbacks to read from and write to streams and send packets out. Call *lsquic_engine_process_conns()* when advised by *lsquic_engine_earliest_adv_tick()*.

Do not call *lsquic_engine_process_conns()* from inside callbacks, for this function is not reentrant.

Another function that sends packets is *lsquic_engine_send_unsent_packets()*. Call it if there was a previous failure to send out all packets

### 3.2.7 Required Engine Callbacks

Now we continue to initialize our engine instance. We have covered the callback to send out packets. This is one of the required engine callbacks. Other required engine callbacks are a set of stream and connection callbacks that get called on various events in then connections and stream lifecycles and a callback to get the default TLS context.

```
struct lsquic_engine_api engine_api = {
  /* --- 8< --- snip --- 8< --- */
  .ea_stream_if       = &stream_callbacks,
  .ea_stream_if_ctx   = &some_context,
  .ea_get_ssl_ctx     = get_ssl_ctx,  /* Server only */
};
```

#### Optional Callbacks

Here we mention some optional callbacks. While they are not covered by this tutorial, it is good to know that they are available.

- Looking up certificate and TLS context by SNI.
- Callbacks to control memory allocation for outgoing packets. These are useful when sending packets using a custom library. For example, when all packets must be in contiguous memory.
- Callbacks to observe connection ID lifecycle. These are useful in multi-process applications.
- Callbacks that provide access to a shared-memory hash. This is also used in multi-process applications.
- HTTP header set processing. These callbacks may be used in HTTP mode for HTTP/3 and Google QUIC.

Please refer to *Engine Settings* for details.

### 3.2.8 Stream and connection callbacks

Stream and connection callbacks are the way that the library communicates with user code. Some of these callbacks are mandatory; others are optional. They are all collected in `lsquic_stream_if` ("if" here stands for "interface"). The mandatory callbacks include calls when connections and streams are created and destroyed and callbacks when streams can be read from or written to. The optional callbacks are used to observe some events in the connection lifecycle, such as being informed when handshake has succeeded (or failed) or when a goaway signal is received from peer.

```
struct lsquic_stream_if
{
    /* Mandatory callbacks: */
    lsquic_conn_ctx_t *(*on_new_conn)(void *stream_if_ctx,
                                                    lsquic_conn_t *c);
    void (*on_conn_closed)(lsquic_conn_t *c);
    lsquic_stream_ctx_t *
         (*on_new_stream)(void *stream_if_ctx, lsquic_stream_t *s);
    void (*on_read)      (lsquic_stream_t *s, lsquic_stream_ctx_t *h);
    void (*on_write)     (lsquic_stream_t *s, lsquic_stream_ctx_t *h);
    void (*on_close)     (lsquic_stream_t *s, lsquic_stream_ctx_t *h);

    /* Optional callbacks: */
    void (*on_goaway_received)(lsquic_conn_t *c);
    void (*on_hsk_done)(lsquic_conn_t *c, enum lsquic_hsk_status s);
    void (*on_new_token)(lsquic_conn_t *c, const unsigned char *token,
    void (*on_sess_resume_info)(lsquic_conn_t *c, const unsigned char *, size_t);
};
```

### On new connection

When a connection object is created, the "on new connection" callback is called. In server mode, the handshake is already known to have succeeded; in client mode, the connection object is created before the handshake is attempted. The client can tell when handshake succeeds or fails by relying on the optional "handshake is done" callback or the "on connection close" callback.

```
/* Return pointer to per-connection context.  OK to return NULL. */
static lsquic_conn_ctx_t *
my_on_new_conn (void *ea_stream_if_ctx, lsquic_conn_t *conn)
{
    struct some_context *ctx = ea_stream_if_ctx;
    struct my_conn_ctx *my_ctx = my_ctx_new(ctx);
    if (ctx->is_client)
        /* Need a stream to send request */
        lsquic_conn_make_stream(conn);
    return (void *) my_ctx;
}
```

In the made-up example above, a new per-connection context is allocated and returned. This context is then associated with the connection and can be retrieved using a dedicated function. Note that it is OK to return a `NULL` pointer. Note that in client mode, this is a good place to request that the connection make a new stream by calling *`lsquic_conn_make_stream()`*. The connection will create a new stream when handshake succeeds.

### On new stream

QUIC allows either endpoint to create streams and send and receive data on them. There are unidirectional and bidirectional streams. Thus, there are four stream types. In our tutorial, however, we use the familiar paradigm of the client sending requests to the server using bidirectional stream.

On the server, new streams are created when client requests arrive. On the client, streams are created when possible after the user code has requested stream creation by calling *`lsquic_conn_make_stream()`*.

```
/* Return pointer to per-connection context.  OK to return NULL. */
static lsquic_stream_ctx_t *
my_on_new_stream (void *ea_stream_if_ctx, lsquic_stream_t *stream) {
    struct some_context *ctx = ea_stream_if_ctx;
    /* Associate some data with this stream: */
    struct my_stream_ctx *stream_ctx
                    = my_stream_ctx_new(ea_stream_if_ctx);
    stream_ctx->stream = stream;
    if (ctx->is_client)
        lsquic_stream_wantwrite(stream, 1);
    return (void *) stream_ctx;
}
```

In a pattern similar to the "on new connection" callback, a per-stream context can be created at this time. The function returns this context and other stream callbacks - "on read," "on write," and "on close" - will be passed a pointer to it. As before, it is OK to return `NULL`. You can register an interest in reading from or writing to the stream by using a "want read" or "want write" function. Alternatively, you can simply read or write; be prepared that this may fail and you have to try again in the "regular way." We talk about that next.

### On read

When the "on read" callback is called, there is data to be read from stream, end-of-stream has been reached, or there is an error.

```
static void
my_on_read (lsquic_stream_t *stream, lsquic_stream_ctx_t *h) {
    struct my_stream_ctx *my_stream_ctx = (void *) h;
    unsigned char buf[BUFSZ];

    ssize_t nr = lsquic_stream_read(stream, buf, sizeof(buf));
    /* Do something with the data.... */
    if (nr == 0) /* EOF */ {
        lsquic_stream_shutdown(stream, 0);
        lsquic_stream_wantwrite(stream, 1); /* Want to reply */
    }
}
```

To read the data or to collect the error, call *lsquic_stream_read()*. If a negative value is returned, examine `errno`. If it is not `EWOULDBLOCK`, then an error has occurred, and you should close the stream. Here, an error means an application error, such as peer resetting the stream. A protocol error or an internal library error (such as memory allocation failure) lead to the connection being closed outright. To reiterate, the "on read" callback is called only when the user has registered interest in reading from the stream.

### On write

The "on write" callback is called when the stream can be written to. At this point, you should be able to write at least a byte to the stream. As with the "on read" callback, for this callback to be called, the user must have registered interest in writing to stream using *lsquic_stream_wantwrite()*.

```
static void
my_on_write (lsquic_stream_t *stream, lsquic_stream_ctx_t *h) {
    struct my_stream_ctx *my_stream_ctx = (void *) h;
    ssize_t nw = lsquic_stream_write(stream,
        my_stream_ctx->resp, my_stream_ctx->resp_sz);
    if (nw == my_stream_ctx->resp_sz)
        lsquic_stream_close(stream);
}
```

By default, "on read" and "on write" callbacks will be called in a loop as long as there is data to read or the stream can be written to. If you are done reading from or writing to stream, you should either shut down the appropriate end, close the stream, or unregister your interest. The library implements a circuit breaker to stop would-be infinite loops when no reading or writing progress is made. Both loop dispatch and the circuit breaker are configurable (see *lsquic_engine_settings.es_progress_check* and *lsquic_engine_settings.es_rw_once*).

### On stream close

When reading and writing ends of the stream have been closed, the "on close" callback is called. After this function returns, pointers to the stream become invalid. (The library destroys the stream object when it deems proper.) This is a good place to perform necessary cleanup.

```
static void
my_on_close (lsquic_stream_t *stream, lsquic_stream_ctx_t *h) {
    lsquic_conn_t *conn = lsquic_stream_conn(stream);
    struct my_conn_ctx *my_ctx = lsquic_conn_get_ctx(conn);
    if (!has_more_reqs_to_send(my_ctx)) /* For example */
        lsquic_conn_close(conn);
    free(h);
}
```

In the made-up example above, we free the per-stream context allocated in the "on new stream" callback and we may close the connection.

#### On connection close

When either *lsquic_conn_close()* has been called; or the peer has closed the connection; or an error has occurred, the "on connection close" callback is called. At this point, it is time to free the per-connection context, if any.

```
static void
my_on_conn_closed (lsquic_conn_t *conn) {
    struct my_conn_ctx *my_ctx = lsquic_conn_get_ctx(conn);
    struct some_context *ctx = my_ctx->some_context;

    --ctx->n_conns;
    if (0 == ctx->n_conn && (ctx->flags & CLOSING))
        exit_event_loop(ctx);

    free(my_ctx);
}
```

In the example above, you see the call to *lsquic_conn_get_ctx()*. This returns the pointer returned by the "on new connection" callback.

### 3.2.9 Using Streams

To reduce buffering, most of the time bytes written to stream are written into packets directly. Bytes are buffered in the stream until a full packet can be created. Alternatively, one could flush the data by calling *lsquic_stream_flush()*. It is impossible to write more data than the congestion window. This prevents excessive buffering inside the library. Inside the "on read" and "on write" callbacks, reading and writing should succeed. The exception is error collection inside the "on read" callback. Outside of the callbacks, be ready to handle errors. For reading, it is -1 with `EWOULDBLOCK` errno. For writing, it is the return value of 0.

#### More stream functions

Here are a few more useful stream functions.

```
/* Flush any buffered data.  This triggers packetizing even a single
 * byte into a separate frame.
 */
int
lsquic_stream_flush (lsquic_stream_t *);

/* Possible values for how are 0, 1, and 2.  See shutdown(2). */
int
lsquic_stream_shutdown (lsquic_stream_t *, int how);

int
lsquic_stream_close (lsquic_stream_t *);
```

As mentioned before, calling *lsquic_stream_flush()* will cause the stream to packetize the buffered data. Note that it may not happen immediately, as there may be higher-priority writes pending or there may not be sufficient congestion window to do so. Calling "flush" only schedules writing to packets.

*lsquic_stream_shutdown()* and *lsquic_stream_close()* mimic the interface of the "shutdown" and "close" socket functions. After both read and write ends of a stream are closed, the "on stream close" callback will soon be called.

### Stream return values

The stream read and write functions are modeled on the standard UNIX read and write functions, including the use of the `errno`. The most important of these error codes are `EWOULDBLOCK` and `ECONNRESET` because you may encounter these even if you structure your code correctly. Other errors typically occur when the user code does something unexpected.

Return value of 0 is different for reads and writes. For reads, it means that EOF has been reached and you need to stop reading from the stream. For writes, it means that you should try writing later.

If writing to stream returns an error, it may mean an internal error. If the error is not recoverable, the library will abort the connection; if it is recoverable (the only recoverable error is failure to allocate memory), attempting to write later may succeed.

### Scatter/gather stream functions

There is the scatter/gather way to read from and write to stream and the interface is similar to the usual "readv" and "writev" functions. All return values and error codes are the same as in the stream read and write functions we have just discussed. Those are actually just wrappers around the scatter/gather versions.

```
ssize_t
lsquic_stream_readv (lsquic_stream_t *, const struct iovec *,
                                                    int iovcnt);
ssize_t
lsquic_stream_writev (lsquic_stream_t *, const struct iovec *,
                                                      int count);
```

### Read using a callback

The scatter/gather functions themselves are also wrappers. LSQUIC provides stream functions that skip intermediate buffering. They are used for zero-copy stream processing.

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

The second argument to *lsquic_stream_readf()* is a callback that returns the number of bytes processed. The callback is passed:

- Pointer to user-supplied context;
- Pointer to the data;
- Data size (can be zero); and
- Indicator whether the FIN follows the data.

If callback returns 0 or value smaller than `len()`, reading stops.

### Read with callback: Example 1

Here is the first example of reading from stream using a callback. Now the process of reading from stream is split into two functions.

```
static void
tut_client_on_read_v1 (lsquic_stream_t *stream, lsquic_stream_ctx_t *h)
{
  struct tut *tut = (struct tut *) h;
  size_t nread = lsquic_stream_readf(stream, tut_client_readf_v1, NULL);
  if (nread == 0)
  {
      LOG("read to end-of-stream: close and read from stdin again");
      lsquic_stream_shutdown(stream, 0);
      ev_io_start(tut->tut_loop, &tut->tut_u.c.stdin_w);
  }
  /* ... */
}
```

Here, we see the *lsquic_stream_readf()* call. The return value is the same as the other read functions. Because in this example there is no extra information to pass to the callback (we simply print data to stdout), the third argument is NULL.

```
static size_t
tut_client_readf_v1 (void *ctx, const unsigned char *data,
                                                  size_t len, int fin)
{
    if (len)
    {
        fwrite(data, 1, len, stdout);
        fflush(stdout);
    }
    return len;
}
```

Here is the callback itself. You can see it is very simple. If there is data to be processed, it is printed to stdout.

Note that the data size (`len` above) can be anything. It is not limited by UDP datagram size. This is because when incoming STREAM frames pass some fragmentation threshold, LSQUIC begins to copy incoming STREAM data to a data structure that is impervious to stream fragmentation attacks. Thus, it is possible for the callback to pass a pointer to data that is over 3KB in size. The implementation may change, so again, no guarantees. When the fourth argument, `fin`, is true, this indicates that the incoming data ends after `len` bytes have been read.

### Read with callback: Example 2: Use FIN

The FIN indicator passed to the callback gives us yet another way to detect end-of-stream. The previous version checked the return value of *lsquic_stream_readf()* to check for EOS. Instead, we can use `fin` in the callback.

The second zero-copy read example is a little more efficient as it saves us an extra call to `tut_client_on_read_v2`. Here, we package pointers to the tut struct and stream into a special struct and pass it to `lsquic_stream_readf()`.

```
struct client_read_v2_ctx { struct tut *tut; lsquic_stream_t *stream; };

static void
tut_client_on_read_v2 (lsquic_stream_t *stream,
```

(continues on next page)

(continued from previous page)

```
                                                lsquic_stream_ctx_t *h)
{
  struct tut *tut = (struct tut *) h;
  struct client_read_v2_ctx v2ctx = { tut, stream, };
  ssize_t nread = lsquic_stream_readf(stream, tut_client_readf_v2,
                                                                  &v2ctx);
  if (nread < 0)
    /* ERROR */
}
```

Now the callback becomes more complicated, as we moved the logic to stop reading from stream into it. We need pointer to both stream and user context when "fin" is true. In that case, we call *lsquic_stream_shutdown()* and begin reading from stdin again to grab the next line of input.

```
static size_t
tut_client_readf_v2 (void *ctx, const unsigned char *data,
                                                 size_t len, int fin)
{
  struct client_read_v2_ctx *v2ctx = ctx;
  if (len)
    fwrite(data, 1, len, stdout);
  if (fin)
  {
    fflush(stdout);
    LOG("read to end-of-stream: close and read from stdin again");
    lsquic_stream_shutdown(v2ctx->stream, 0);
    ev_io_start(v2ctx->tut->tut_loop, &v2ctx->tut->tut_u.c.stdin_w);
  }
  return len;
}
```

### Writing to stream: Example 1

Now let's consider writing to stream.

```
static void
tut_server_on_write_v0 (lsquic_stream_t *stream, lsquic_stream_ctx_t *h)
{
  struct tut_server_stream_ctx *const tssc = (void *) h;
  ssize_t nw = lsquic_stream_write(stream,
      tssc->tssc_buf + tssc->tssc_off, tssc->tssc_sz - tssc->tssc_off);
  if (nw > 0)
  {
      tssc->tssc_off += nw;
      if (tssc->tssc_off == tssc->tssc_sz)
          lsquic_stream_close(stream);
  /* ... */
}
```

Here, we call *lsquic_stream_write()* directly. If writing succeeds and we reached the end of the buffer we wanted to write, we close the stream.

### Write using callbacks

To write using a callback, we need to use *lsquic_stream_writef()*.

```
struct lsquic_reader {
  /* Return number of bytes written to buf */
  size_t (*lsqr_read) (void *lsqr_ctx, void *buf, size_t count);
  /* Return number of bytes remaining in the reader.  */
  size_t (*lsqr_size) (void *lsqr_ctx);
  void    *lsqr_ctx;
};

/* Return umber of bytes written or -1 on error. */
ssize_t
lsquic_stream_writef (lsquic_stream_t *, struct lsquic_reader *);
```

We must specify not only the function that will perform the copy, but also the function that will return the number of bytes remaining. This is useful in situations where the size of the data source may change. For example, an underlying file may change size. The *lsquic_reader.lsqr_read* callback will be called in a loop until stream can write no more or until *lsquic_reader.lsqr_size* returns zero. The return value of `lsquic_stream_writef` is the same as *lsquic_stream_write()* and *lsquic_stream_writev()*, which are just wrappers around the "writef" version.

### Writing to stream: Example 2

Here is the second version of the "on write" callback. It uses *lsquic_stream_writef()*.

```
static void
tut_server_on_write_v1 (lsquic_stream_t *stream, lsquic_stream_ctx_t *h)
{
    struct tut_server_stream_ctx *const tssc = (void *) h;
    struct lsquic_reader reader = { tssc_read, tssc_size, tssc, };
    ssize_t nw = lsquic_stream_writef(stream, &reader);
    if (nw > 0 && tssc->tssc_off == tssc->tssc_sz)
        lsquic_stream_close(stream);
    /* ... */
}
```

The reader struct is initialized with pointers to read and size functions and this struct is passed to the "writef" function.

```
static size_t
tssc_size (void *ctx)
{
  struct tut_server_stream_ctx *tssc = ctx;
  return tssc->tssc_sz - tssc->tssc_off;
}
```

The size callback simply returns the number of bytes left.

```
static size_t
tssc_read (void *ctx, void *buf, size_t count)
{
  struct tut_server_stream_ctx *tssc = ctx;

  if (count > tssc->tssc_sz - tssc->tssc_off)
    count = tssc->tssc_sz - tssc->tssc_off;
  memcpy(buf, tssc->tssc_buf + tssc->tssc_off, count);
  tssc->tssc_off += count;
  return count;
}
```

The read callback (so called because you *read* data from the source) writes no more than `count` bytes to memory location pointed by `buf` and returns the number of bytes copied. In our case, `count` is never larger than the number of bytes still left to write. This is because the caller - the LSQUIC library - gets the value of `count` from the `lsqr_size()` callback. When reading from a file descriptor, on the other hand, this can very well happen that you don't have as much data to write as you thought you had.

### 3.2.10 Client: making connection

We now switch our attention to making a QUIC connection. The function *`lsquic_engine_connect()`* does that. This function has twelve arguments. (These arguments have accreted over time.)

```
lsquic_conn_t *
lsquic_engine_connect (lsquic_engine_t *,
      enum lsquic_version, /* Set to N_LSQVER for default */
      const struct sockaddr *local_sa,
      const struct sockaddr *peer_sa,
      void *peer_ctx,
      lsquic_conn_ctx_t *conn_ctx,
      const char *hostname,          /* Used for SNI */
      unsigned short base_plpmtu, /* 0 means default */
      const unsigned char *sess_resume, size_t sess_resume_len,
      const unsigned char *token, size_t token_sz);
```

- The first argument is the pointer to the engine instance.
- The second argument is the QUIC version to use.
- The third and fourth arguments specify local and destination addresses, respectively.
- The fifth argument is the so-called "peer context."
- The sixth argument is the connection context. This is used if you need to pass a pointer to the "on new connection" callback. This context is overwritten by the return value of the "on new connection" callback.
- The argument "hostname," which is the seventh argument, is used for SNI. This argument is optional, just as the rest of the arguments that follow.
- The eighth argument is the initial maximum size of the UDP payload. This will be the base PLPMTU if DPLPMTUD is enabled. Specifying zero, or default, is the safe way to go: lsquic will pick a good starting value.
- The next two arguments allow one to specify a session resumption information to establish a connection faster. In the case of IETF QUIC, this is the TLS Session Ticket. To get this ticket, specify the *`lsquic_stream_if.on_sess_resume_info`* callback.
- The last pair of arguments is for specifying a token to try to prevent a potential stateless retry from the server. The token is learned in a previous session. See the optional callback *`lsquic_stream_if.on_new_token`*.

```
tut.tut_u.c.conn = lsquic_engine_connect(
    tut.tut_engine, N_LSQVER,
    (struct sockaddr *) &tut.tut_local_sas, &addr.sa,
    (void *) (uintptr_t) tut.tut_sock_fd,  /* Peer ctx */
    NULL, NULL, 0, NULL, 0, NULL, 0);
if (!tut.tut_u.c.conn)
{
    LOG("cannot create connection");
    exit(EXIT_FAILURE);
}
tut_process_conns(&tut);
```

Here is an example from a tutorial program. The connect call is a lot less intimidating in real life, as half the arguments are set to zero. We pass a pointer to the engine instance, N_LSQVER to let the engine pick the version to use and the two socket addresses. The peer context is simply the socket file descriptor cast to a pointer. This is what is passed to the "send packets out" callback.

### 3.2.11 Specifying QUIC version

QUIC versions in LSQUIC are gathered in an enum, *lsquic_version*, and have an arbitrary value.

```
enum lsquic_version {
    LSQVER_043, LSQVER_046, LSQVER_050,     /* Google QUIC */
    LSQVER_ID27, LSQVER_ID28, LSQVER_ID29,  /* IETF QUIC */
    /* ...some special entries skipped */
    N_LSQVER    /* <====================== Special value */
};
```

The special value "N_LSQVER" is used to let the engine pick the QUIC version. It picks the latest non-experimental version, so in this case it picks ID-29. (Experimental from the point of view of the library.)

Because version enum values are small – and that is by design – a list of versions can be passed around as bitmasks.

```
/* This allows list of versions to be specified as bitmask: */
es_versions = (1 << LSQVER_ID28) | (1 << LSQVER_ID29);
```

This is done, for example, when specifying list of versions to enable in engine settings using `lsquic_engine_api.ea_versions`. There are a couple of more places in the API where this technique is used.

### 3.2.12 Server callbacks

The server requires SSL callbacks to be present. The basic required callback is *lsquic_engine_api.ea_get_ssl_ctx*. It is used to get a pointer to an initialized `SSL_CTX`.

```
typedef struct ssl_ctx_st * (*lsquic_lookup_cert_f)(
    void *lsquic_cert_lookup_ctx, const struct sockaddr *local,
    const char *sni);

struct lsquic_engine_api {
  lsquic_lookup_cert_f   ea_lookup_cert;
  void                  *ea_cert_lu_ctx;
  struct ssl_ctx_st *  (*ea_get_ssl_ctx)(void *peer_ctx);
  /* (Other members of the struct are not shown) */
};
```

In case SNI is used, LSQUIC will call *lsquic_engine_api.ea_lookup_cert*. For example, SNI is required in HTTP/3. In our web server, each virtual host has its own SSL context. Note that besides the SNI string, the callback is also given the local socket address. This makes it possible to implement a flexible lookup mechanism.

### 3.2.13 Engine settings

Besides the engine API struct passed to the engine constructor, there is also an engine settings struct, *lsquic_engine_settings*. *lsquic_engine_api.ea_settings* in the engine API struct can be pointed to a custom settings struct. By default, this pointer is `NULL`. In that case, the engine uses default settings.

There are many settings, controlling everything from flow control windows to the number of times an "on read" callback can be called in a loop before it is deemed an infinite loop and the circuit breaker is tripped. To make changing default settings values easier, the library provides functions to initialize the settings struct to defaults and then to check these values for sanity.

### Settings helper functions

```
/* Initialize `settings' to default values */
void
lsquic_engine_init_settings (struct lsquic_engine_settings *,
  /* Bitmask of LSENG_SERVER and LSENG_HTTP */
                             unsigned lsquic_engine_flags);

/* Check settings for errors, return 0 on success, -1 on failure. */
int
lsquic_engine_check_settings (const struct lsquic_engine_settings *,
                              unsigned lsquic_engine_flags,
                              /* Optional, can be NULL: */
                              char *err_buf, size_t err_buf_sz);
```

The first function is *lsquic_engine_init_settings()*, which does just that. The second argument is a bitmask to specify whether the engine is in server mode and whether HTTP mode is turned on. These should be the same flags as those passed to the engine constructor.

Once you have initialized the settings struct in this manner, change the setting or settings you want and then call *lsquic_engine_check_settings()*. The first two arguments are the same as in the initializer. The third and fourth argument are used to pass a pointer to a buffer into which a human-readable error string can be placed.

The checker function does only the basic sanity checks. If you really set out to misconfigure LSQUIC, you can. On the bright side, each setting is clearly documented (see *Engine Settings*). Most settings are standalone; when there is interplay between them, it is also documented. Test before deploying!

### Settings example

The example is adapted from a tutorial program. Here, command-line options are processed and appropriate options is set. The first time the `-o` flag is encountered, the settings struct is initialized. Then the argument is parsed to see which setting to alter.

```
while (/* getopt */)
{
    case 'o':   /* For example: -o version=h3-27 -o cc_algo=2 */
      if (!settings_initialized) {
        lsquic_engine_init_settings(&settings,
                        cert_file || key_file ? LSENG_SERVER : 0);
        settings_initialized = 1;
      }
      /* ... */
      else if (0 == strncmp(optarg, "cc_algo=", val - optarg))
        settings.es_cc_algo = atoi(val);
    /* ... */
}

/* Check settings */
if (0 != lsquic_engine_check_settings(&settings,
                tut.tut_flags & TUT_SERVER ? LSENG_SERVER : 0,
```

(continues on next page)

(continued from previous page)

```
                errbuf, sizeof(errbuf)))
{
  LOG("invalid settings: %s", errbuf);
  exit(EXIT_FAILURE);
}

/* ... */
eapi.ea_settings = &settings;
```

After option processing is completed, the settings are checked. The error buffer is used to log a configuration error.

Finally, the settings struct is pointed to by the engine API struct before the engine constructor is called.

### 3.2.14 Logging

LSQUIC provides a simple logging interface using a single callback function. By default, no messages are logged. This can be changed by calling *lsquic_logger_init()*. This will set a library-wide logger callback function.

```
void lsquic_logger_init(const struct lsquic_logger_if *,
    void *logger_ctx, enum lsquic_logger_timestamp_style);

struct lsquic_logger_if {
  int (*log_buf)(void *logger_ctx, const char *buf, size_t len);
};

enum lsquic_logger_timestamp_style { LLTS_NONE, LLTS_HHMMSSMS,
    LLTS_YYYYMMDD_HHMMSSMS, LLTS_CHROMELIKE, LLTS_HHMMSSUS,
    LLTS_YYYYMMDD_HHMMSSUS, N_LLTS };
```

You can instruct the library to generate a timestamp and include it as part of the message. Several timestamp formats are available. Some display microseconds, some do not; some display the date, some do not. One of the most useful formats is "chromelike," which matches the somewhat weird timestamp format used by Chromium. This makes it easy to compare the two logs side by side.

There are eight log levels in LSQUIC: debug, info, notice, warning, error, alert, emerg, and crit. These correspond to the usual log levels. (For example, see `syslog(3)`). Of these, only five are used: debug, info, notice, warning, and error. Usually, warning and error messages are printed when there is a bug in the library or something very unusual has occurred. Memory allocation failures might elicit a warning as well, to give the operator a heads up.

LSQUIC possesses about 40 logging modules. Each module usually corresponds to a single piece of functionality in the library. The exception is the "event" module, which logs events of note in many modules. There are two functions to manipulate which log messages will be generated.

```
/* Set log level for all modules */
int
lsquic_set_log_level (const char *log_level);

/* Set log level per module "event=debug" */
int
lsquic_logger_lopt (const char *optarg);
```

The first is *lsquic_set_log_level()*. It sets the same log level for each module. The second is *lsquic_logger_lopt()*. This function takes a comma-separated list of name-value pairs. For example, "event=debug."

### Logging Example

The following example is adapted from a tutorial program. In the program, log messages are written to a file handle. By default, this is the standard error. One can change that by using the "-f" command-line option and specify the log file.

```
static int
tut_log_buf (void *ctx, const char *buf, size_t len) {
  FILE *out = ctx;
  fwrite(buf, 1, len, out);
  fflush(out);
  return 0;
}
static const struct lsquic_logger_if logger_if = { tut_log_buf, };

lsquic_logger_init(&logger_if, s_log_fh, LLTS_HHMMSSUS);
```

`tut_log_buf()` returns 0, but the truth is that the return value is ignored. There is just nothing for the library to do when the user-supplied log function fails!

```
case 'l':   /* e.g. -l event=debug,cubic=info */
  if (0 != lsquic_logger_lopt(optarg)) {
      fprintf(stderr, "error processing -l option\n");
      exit(EXIT_FAILURE);
  }
  break;
case 'L':   /* e.g. -L debug */
  if (0 != lsquic_set_log_level(optarg)) {
      fprintf(stderr, "error processing -L option\n");
      exit(EXIT_FAILURE);
  }
  break;
```

Here you can see how we use `-l` and `-L` command-line options to call one of the two log level functions. These functions can fail if the incorrect log level or module name is passed. Both log level and module name are treated in case-insensitive manner.

### Sample log messages

When log messages are turned on, you may see something like this in your log file (timestamps and log levels are elided for brevity):

```
[QUIC:B508E8AA234E0421] event: generated STREAM frame: stream 0, offset: 0, size: 3,␣
↪fin: 1
[QUIC:B508E8AA234E0421-0] stream: flushed to or past required offset 3
[QUIC:B508E8AA234E0421] event: sent packet 13, type Short, crypto: forw-secure, size␣
↪32, frame types: STREAM, ecn: 0, spin: 0; kp: 0, path: 0, flags: 9470472
[QUIC:B508E8AA234E0421] event: packet in: 15, type: Short, size: 44; ecn: 0, spin: 0;␣
↪path: 0
[QUIC:B508E8AA234E0421] rechist: received 15
[QUIC:B508E8AA234E0421] event: ACK frame in: [13-9]
[QUIC:B508E8AA234E0421] conn: about to process QUIC_FRAME_STREAM frame
[QUIC:B508E8AA234E0421] event: STREAM frame in: stream 0; offset 0; size 3; fin: 1
[QUIC:B508E8AA234E0421-0] stream: received stream frame, offset 0x0, len 3; fin: 1
[QUIC:B508E8AA234E0421-0] di: FIN set at 3
```

Here we see the connection ID, `B508E8AA234E0421`, and logging for modules "event", "stream", "rechist" (that stands for "receive history"), "conn", and "di" (the "data in" module). When the connection ID is followed by a dash and that number, the number is the stream ID. Note that stream ID is logged not just for the stream, but for some other modules as well.

### 3.2.15 Key logging and Wireshark

Wireshark supports IETF QUIC. The developers have been very good at keeping up with latest versions. You will need version 3.3 of Wireshark to support Internet-Draft 29. Support for HTTP/3 is in progress.

LSQUIC supports exporting TLS secrets. For that, you need to specify a set of function pointers via `lsquic_engine_api.ea_keylog_if`.

```
/* Secrets are logged per connection.  Interface to open file (handle),
 * log lines, and close file.
 */
struct lsquic_keylog_if {
    void * (*kli_open) (void *keylog_ctx, lsquic_conn_t *);
    void   (*kli_log_line) (void *handle, const char *line);
    void   (*kli_close) (void *handle);
};

struct lsquic_engine_api {
  /* --- 8< --- snip --- 8< --- */
  const struct lsquic_keylog_if       *ea_keylog_if;
  void                                *ea_keylog_ctx;
};
```

There are three functions: one to open a file, one to write a line into the file, and one to close the file. The lines are not interpreted. In the engine API struct, there are two members to set: one is the pointer to the struct with the function pointers, and the other is the context passed to "kli_open" function.

#### Key logging example

```
static void *
keylog_open (void *ctx, lsquic_conn_t *conn)
{
    const lsquic_cid_t *cid;
    FILE *fh;
    int sz;
    unsigned i;
    char id_str[MAX_CID_LEN * 2 + 1];
    char path[PATH_MAX];
    static const char b2c[16] = "0123456789ABCDEF";

    cid = lsquic_conn_id(conn);
    for (i = 0; i < cid->len; ++i)
    {
        id_str[i * 2 + 0] = b2c[ cid->idbuf[i] >> 4 ];
        id_str[i * 2 + 1] = b2c[ cid->idbuf[i] & 0xF ];
    }
    id_str[i * 2] = '\0';
    sz = snprintf(path, sizeof(path), "/secret_dir/%s.keys", id_str);
    if ((size_t) sz >= sizeof(path))
    {
```

(continues on next page)

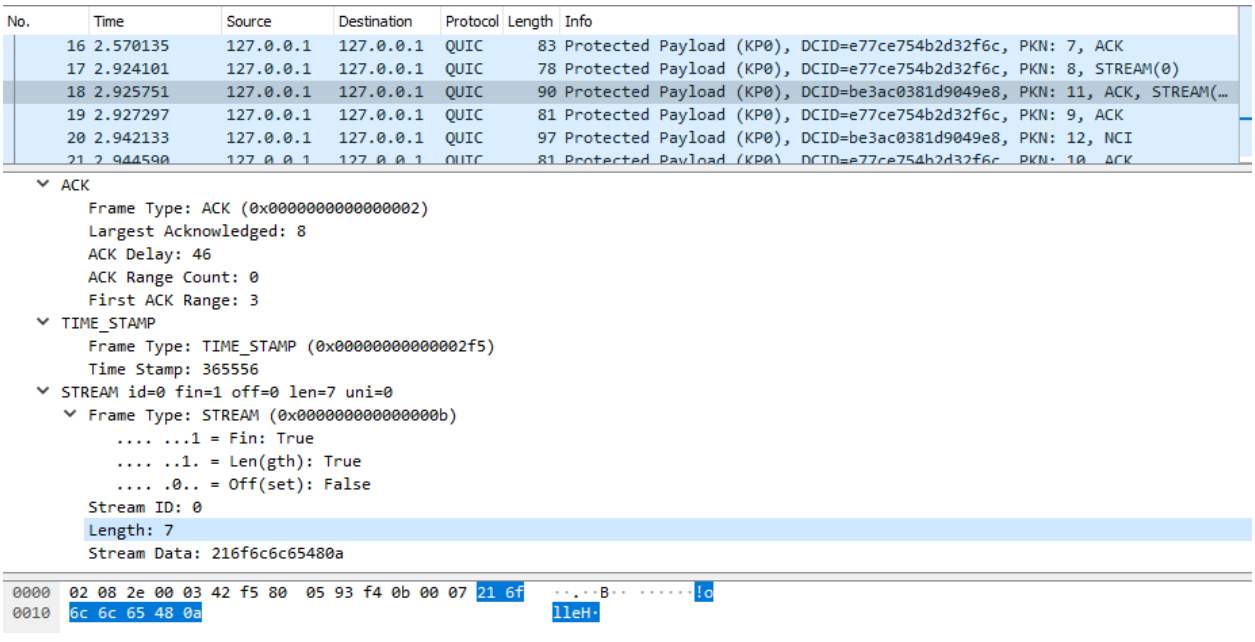
(continued from previous page)

```
        LOG("WARN: %s: file too long", __func__);
        return NULL;
    }
    fh = fopen(path, "wb");
    if (!fh)
        LOG("WARN: could not open %s for writing: %s", path, strerror(errno));
    return fh;
}

static void
keylog_log_line (void *handle, const char *line)
{
    fputs(line, handle);
    fputs("\n", handle);
    fflush(handle);
}

static void
keylog_close (void *handle)
{
    fclose(handle);
}
```

The function to open the file is passed the connection object. It can be used to generate a filename based on the connection ID. We see that the line logger simply writes the passed C string to the filehandle and appends a newline.

### Wireshark screenshot

After jumping through those hoops, our reward is a decoded QUIC trace in Wireshark!




Here, we highlighted the STREAM frame payload. Other frames in view are ACK and TIMESTAMP frames. In the top panel with the packet list, you can see that frames are listed after the packet number. Another interesting item is the DCID. This stands for "Destination Connection ID," and you can see that there are two different values there. This is because the two peers of the QUIC connection place different connection IDs in the packets!

### 3.2.16 Connection IDs

A QUIC connection has two sets of connection IDs: source connection IDs and destination connection IDs. The source connection IDs set is what the peer uses to place in QUIC packets; the destination connection IDs is what this endpoint uses to include in the packets it sends to the peer. One's source CIDs is the other's destination CIDs and vice versa. What interesting is that either side of the QUIC connection may change the DCID. Use CIDs with care.

```
#define MAX_CID_LEN 20

typedef struct lsquic_cid
{
    uint_fast8_t    len;
    union {
        uint8_t     buf[MAX_CID_LEN];
        uint64_t    id;
    }               u_cid;
#define idbuf u_cid.buf
} lsquic_cid_t;

#define LSQUIC_CIDS_EQ(a, b) ((a)->len == 8 ? \
    (b)->len == 8 && (a)->u_cid.id == (b)->u_cid.id : \
    (a)->len == (b)->len && 0 == memcmp((a)->idbuf, (b)->idbuf, (a)->len))
```

The LSQUIC representation of a CID is the struct above. The CID can be up to 20 bytes in length. By default, LSQUIC uses 8-byte CIDs to speed up comparisons.

### 3.2.17 Get this-and-that API

Here are a few functions to get different LSQUIC objects from other objects.

```
const lsquic_cid_t *
lsquic_conn_id (const lsquic_conn_t *);

lsquic_conn_t *
lsquic_stream_conn (const lsquic_stream_t *);

lsquic_engine_t *
lsquic_conn_get_engine (lsquic_conn_t *);

int lsquic_conn_get_sockaddr (lsquic_conn_t *,
      const struct sockaddr **local, const struct sockaddr **peer);
```

The CID returned by *lsquic_conn_id()* is that used for logging: server and client should return the same CID. As noted earlier, you should not rely on this value to identify a connection! You can get a pointer to the connection from a stream and a pointer to the engine from a connection. Calling *lsquic_conn_get_sockaddr()* will point `local` and `peer` to the socket addressess of the current path. QUIC supports multiple paths during migration, but access to those paths has not been exposed via an API yet. This may change when or if QUIC adds true multipath support.

## 3.3 API Reference

### 3.3.1 Preliminaries

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

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

in each source file.

### 3.3.2 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.

### 3.3.3 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

**LSQVER_ID29**
IETF QUIC version ID 29

**LSQVER_ID30**
IETF QUIC version ID 30

**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.

**LSQUIC_DEPRECATED_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.

### 3.3.4 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 *lsquic_stream_if.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

### 3.3.5 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();
```

### 3.3.6 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*

### 3.3.7 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.

**LSENG_SERVER**
One of possible bitmask values passed as first argument to *lsquic_engine_new*. When set, the engine instance will be in the server mode.

**LSENG_HTTP**
One of possible bitmask values passed as first argument to *lsquic_engine_new*. When set, the engine instance will enable HTTP functionality.

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.

### 3.3.8 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.

const char ***ea_alpn**
The optional ALPN string is used by the client if *`LSENG_HTTP`* is not set.

### 3.3.9 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 CFCW 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.

This setting is applicable to both gQUIC and IETF QUIC.

See *lsquic_engine_settings.es_cfcw*, *lsquic_engine_settings.es_init_max_data*.

unsigned **es_max_sfcw**
This value is used to specify the maximum value stream flow control window is allowed to reach due to auto-tuning. By default, this value is zero, meaning that auto-tuning is turned off.

This setting is applicable to both gQUIC and IETF QUIC.

See *lsquic_engine_settings.es_sfcw*, *lsquic_engine_settings.es_init_max_stream_data_bidi_local*, *lsquic_engine_settings.es_init_max_stream_data_bidi_remote*.

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**
When true, `CONNECTION_CLOSE` is not sent when connection times out. The server will also not send a reply to client's `CONNECTION_CLOSE`.

Corresponds to SCLS (silent close) gQUIC option.

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 and Q050, as the server never includes CIDs in the short packets.

: This setting is applicable to gQUIC only.

: 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.

: The default is *LSQUIC_DF_HONOR_PRST*.

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 the 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_CI* 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: BBRv1

- 3: Adaptive congestion control.

Adaptive congestion control adapts to the environment. It figures out whether to use Cubic or BBRv1 based on the RTT.

unsigned **es_cc_rtt_thresh**
Congestion controller RTT threshold in microseconds.

Adaptive congestion control uses BBRv1 until RTT is determined. At that point a permanent choice of congestion controller is made. If RTT is smaller than or equal to *lsquic_engine_settings.es_cc_rtt_thresh*, congestion controller is switched to Cubic; otherwise, BBRv1 is picked.

The default value is *LSQUIC_DF_CC_RTT_THRESH*

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*

int **es_dplpmtud**
If set to true value, enable DPLPMTUD – Datagram Packetization Layer Path MTU Discovery.

Default value is *LSQUIC_DF_DPLPMTUD*

unsigned short **es_base_plpmtu**
PLPMTU size expected to work for most paths.

If set to zero, this value is calculated based on QUIC and IP versions.

Default value is *LSQUIC_DF_BASE_PLPMTU*

unsigned short **es_max_plpmtu**
Largest PLPMTU size the engine will try.

If set to zero, picking this value is left to the engine.

Default value is *LSQUIC_DF_MAX_PLPMTU*

unsigned **es_mtu_probe_timer**
: This value specifies how long the DPLPMTUD probe timer is, in milliseconds. [draft-ietf-tsvwg-datagram-plpmtud-22] says:

> PROBE_TIMER: The PROBE_TIMER is configured to expire after a period longer than the maximum time to receive an acknowledgment to a probe packet. This value MUST NOT be smaller than 1 second, and SHOULD be larger than 15 seconds. Guidance on selection of the timer value are provided in section 3.1.1 of the UDP Usage Guidelines **RFC 8085#section-3.1**.

If set to zero, the default is used.

Default value is *LSQUIC_DF_MTU_PROBE_TIMER*

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.

int **es_grease_quic_bit**
: Enable the "QUIC bit grease" extension. When set to a true value, lsquic will grease the QUIC bit on the outgoing QUIC packets if the peer sent the "grease_quic_bit" transport parameter.

Default value is *LSQUIC_DF_GREASE_QUIC_BIT*

int **es_datagrams**
: Enable datagrams extension. Allowed values are 0 and 1.

Default value is *LSQUIC_DF_DATAGRAMS*

int **es_optimistic_nat**
: If set to true, changes in peer port are assumed to be due to a benign NAT rebinding and path characteristics – MTU, RTT, and CC state – are not reset.

Default value is *LSQUIC_DF_OPTIMISTIC_NAT*

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) and the server does not reply with own `CONNECTION_CLOSE` after it receives one.

**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**
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 Adaptive Congestion Controller by default.

**LSQUIC_DF_CC_RTT_THRESH**
Default value of the CC RTT threshold is 1500 microseconds

**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_DPLPMTUD**
By default, DPLPMTUD is enabled

**LSQUIC_DF_BASE_PLPMTU**
By default, this value is left up to the engine.

**LSQUIC_DF_MAX_PLPMTU**
By default, this value is left up to the engine.

**LSQUIC_DF_MTU_PROBE_TIMER**
By default, we use the minimum timer of 1000 milliseconds.

**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.

**LSQUIC_DF_GREASE_QUIC_BIT**
By default, greasing the QUIC bit is enabled (if peer sent the "grease_quic_bit" transport parameter).

**LSQUIC_DF_TIMESTAMPS**
Timestamps are on by default.

**LSQUIC_DF_DATAGRAMS**
Datagrams are off by default.

**LSQUIC_DF_OPTIMISTIC_NAT**
Assume optimistic NAT by default.

### 3.3.10 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.

### 3.3.11 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.

typedef int **(*lsquic_packets_out_f)**(void **packets_out_ctx*, const struct *lsquic_out_spec* **out_spec*, unsigned *n_packets_out*)
Returns number of packets successfully sent out or -1 on error. -1 should only be returned if no packets were sent out. If -1 is returned or if the return value is smaller than `n_packets_out`, this indicates that sending of packets is not possible.

If not all packets could be sent out, then:

- errno is examined. If it is not EAGAIN or EWOULDBLOCK, the connection whose packet caused the error is closed forthwith.
- No packets are attempted to be sent out until *lsquic_engine_send_unsent_packets()* is called.

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 `lsquic_engine_api.ea_packets_out` could not send everything out or if processing deadline was exceeded (see `lsquic_engine_settings.es_proc_time_thresh`).

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.

### 3.3.12 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_sess_resume_info)** (*lsquic_conn_t* **c*, const unsigned char *, size_t)
This callback lets client record information needed to perform session resumption next time around.

This callback is optional.

ssize_t **(*on_dg_write)** (*lsquic_conn_t* **c*, void **buf*, size_t *buf_sz*)
Called when datagram is ready to be written. Write at most `buf_sz` bytes to `buf` and return number of bytes written.

void **(*on_datagram)** (*lsquic_conn_t* **c*, const void **buf*, size_t *sz*)
Called when datagram is read from a packet. This callback is required when *`lsquic_engine_settings.es_datagrams`* is true. Take care to process it quickly, as this is called during *`lsquic_engine_packet_in()`*.

### 3.3.13 Creating Connections

In server mode, the connections are created by the library based on incoming packets. After handshake is completed, the library calls *`lsquic_stream_if.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 *base_plpmtu*, const unsigned char **sess_resume*, size_t *sess_resume_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 session resumption information is supplied, version is picked from there instead.
- **local_sa** – Local address.
- **peer_sa** – Address of the server.
- **peer_ctx** – Context associated with the peer. 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.

- **base_plpmtu** – Base PLPMTU. If set to zero, it is selected based on the engine settings (see *lsquic_engine_settings.es_base_plpmtu*), QUIC version, and IP version.
- **sess_resume** – Pointer to previously saved session resumption data needed for TLS resumption. May be NULL.
- **sess_resume_len** – Size of session resumption 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 *lsquic_stream_if.on_new_token* callback.

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

### 3.3.14 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. *lsquic_stream_if.on_conn_closed* and *lsquic_stream_if.on_close* callbacks will be called.

### 3.3.15 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.

### 3.3.16 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.

### 3.3.17 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()*.

### 3.3.18 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.

ssize_t **lsquic_stream_pwritev** (struct lsquic_stream **stream*, ssize_t (**preadv*)(void *user_data, const struct iovec *iov, int iovcnt), void **user_data*, size_t *n_to_write*)

**Parameters**

- **stream** – Stream to write to.
- **preadv** – Pointer to a custom `preadv(2)`-like function.
- **user_data** – Data to pass to `preadv` function.
- **n_to_write** – Number of bytes to write.

**Returns** Number of bytes written or -1 on error.

Write to stream using user-supplied `preadv()` function. The stream allocates one or more packets and calls `preadv()`, which then fills the array of buffers. This is a good way to minimize the number of `read(2)` system calls; the user can call `preadv(2)` instead.

The number of bytes available in the `iov` vector passed back to the user callback may be smaller than `n_to_write`. The expected use pattern is to pass the number of bytes remaining in the file and keep on calling `preadv(2)`.

Note that, unlike other stream-writing functions above, `lsquic_stream_pwritev()` does *not* buffer bytes inside the stream; it only writes to packets. That means the caller must be prepared for this function to return 0 even inside the "on write" stream callback. In that case, the caller should fall back to using another write function.

It is OK for the `preadv` callback to write fewer bytes that `n_to_write`. (This can happen if the underlying data source is truncated.)

```
/*
 * For example, the return value of zero can be handled as follows:
 */
nw = lsquic_stream_pwritev(stream, my_readv, some_ctx, n_to_write);
if (nw == 0)
    nw = lsquic_stream_write(stream, rem_bytes_buf, rem_bytes_len);
```

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.

### 3.3.19 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.

### 3.3.20 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

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.

### 3.3.21 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.

### 3.3.22 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_is_pushed**(const *lsquic_stream_t *stream*)

**Returns** Boolean value indicating whether this is a pushed stream.

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.

### 3.3.23 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. Lower value means higher priority.

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

### 3.3.24 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.

### 3.3.25 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.

### 3.3.26 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.

### 3.3.27 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.

### 3.3.28 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 **peer_ctx*, unsigned short *sz*, char *is_ipv6*)
Allocate buffer for sending.

void **(*pmi_release)**(void **pmi_ctx*, void **peer_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 **peer_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.

### 3.3.29 Global Variables

**const char *const lsquic_ver2str[N_LSQVER]**

Convert LSQUIC version to human-readable string

### 3.3.30 List of Log Modules

The following log modules are defined:

- *alarmset*: Alarm processing.
- *bbr*: BBRv1 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.

### 3.3.31 Datagrams

lsquic supports the Unreliable Datagram Extension. To enable datagrams, set `lsquic_engine_settings.es_datagrams` to true and specify `lsquic_stream_if.on_datagram` and `lsquic_stream_if.on_dg_write` callbacks.

int **lsquic_conn_want_datagram_write**(*lsquic_conn_t* *conn, int *want*)
Indicate desire (or lack thereof) to write a datagram.

**Parameters**

- **conn** – Connection on which to send a datagram.
- **want** – Boolean value indicating whether the caller wants to write a datagram.

**Returns** Previous value of `want` or `-1` if the datagrams cannot be written.

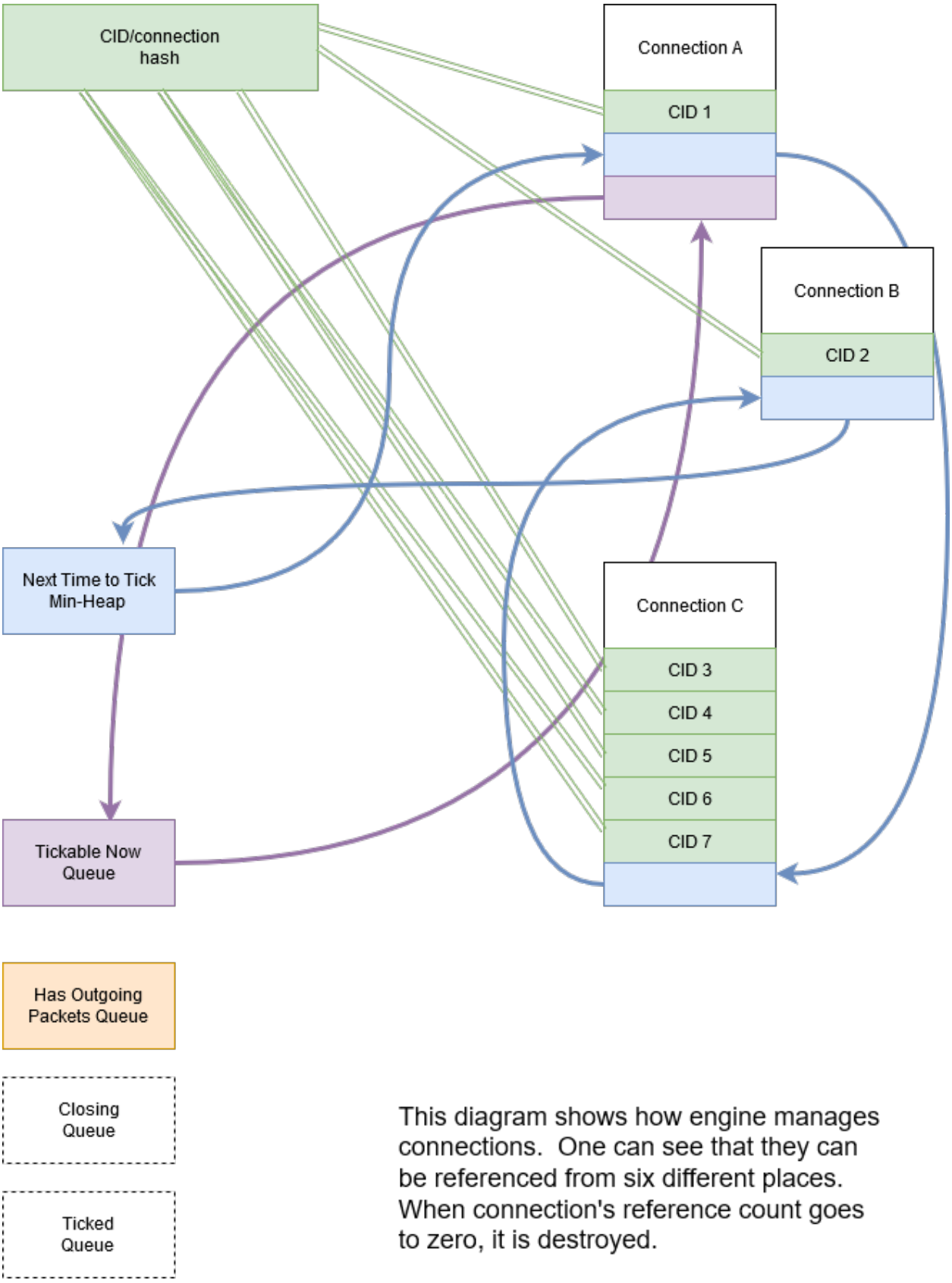
size_t **lsquic_conn_get_min_datagram_size**(*lsquic_conn_t* *conn)
Get minimum datagram size. By default, this value is zero.

int **lsquic_conn_set_min_datagram_size**(*lsquic_conn_t* *conn, size_t *sz*)
Set minimum datagram size. This is the minumum value of the buffer passed to the `lsquic_stream_if.on_dg_write` callback. Returns 0 on success and -1 on error.

## 3.4 Internals

### 3.4.1 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 4

# Indices and tables

- genindex
- search

# Index

## N



## Q



## R



## V