lsquic Documentation

Release 3.0.2

LiteSpeed Technologies

Jun 29, 2021

Contents

This is the documentation for LSQUIC 3.0.2, last updated Jun 29, 2021.

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 v1, Internet-Draft versions 29, and 27; and the older "Google" QUIC versions Q043, Q046, an Q050.

LSQUIC is licensed under the MIT License; see LICENSE in the source distribution for details.

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:
- *Extensible HTTP Priorities*
- *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=h3-29
```

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 More about LSQUIC

You may be also interested in this presentation about LSQUIC. Slides are available here.

3.1.7 Next steps

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

Library Guts covers some library internals.

3.2 Tutorial

Code for this tutorial is available on GitHub.

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 send 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* (optional in client 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
 */
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.
 */
```

(continues on next page)

(continued from previous page)

```
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,
};
```

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,
                                         const struct sockaddr *local);
  /* (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.

To export TLS secrets, use BoringSSL’s `SSL_CTX_set_keylog_callback()`. Use *`lsquic_ssl_to_conn()`* to get the connection associated with the SSL object.

Key logging example

```
static void *
keylog_open_file (const SSL *ssl)
{
    const 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";

    conn = lsquic_ssl_to_conn(ssl);
    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))
    {
        LOG("WARN: %s: file too long", __func__);
        return NULL;
    }
    fh = fopen(path, "ab");
    if (!fh)
        LOG("WARN: could not open %s for appending: %s", path, strerror(errno));
    return fh;
}

static void
keylog_log_line (const SSL *ssl, const char *line)
{
    file = keylog_open_file(ssl);
    if (file)
    {
        fputs(line, file);
        fputs("\n", file);
```

(continues on next page)

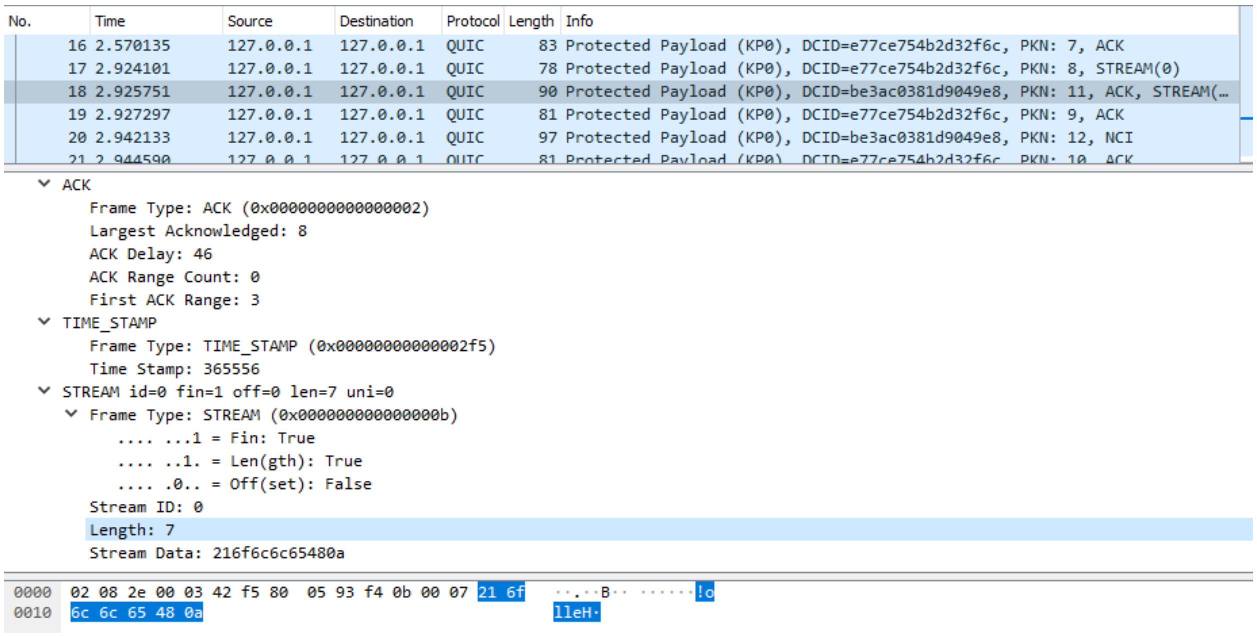
(continued from previous page)

```
        fclose(file);
    }
}

/* ... */

SSL_CTX_set_keylog_callback(ssl, keylog_log_line);
```

The most involved part of this is opening the necessary file, creating it if necessary. The connection 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!



```
No.   Time       Source     Destination  Protocol Length Info
   16 2.570135   127.0.0.1  127.0.0.1    QUIC     83 Protected Payload (KP0), DCID=e77ce754b2d32f6c, PKN: 7, ACK
   17 2.924101   127.0.0.1  127.0.0.1    QUIC     78 Protected Payload (KP0), DCID=e77ce754b2d32f6c, PKN: 8, STREAM(0)
   18 2.925751   127.0.0.1  127.0.0.1    QUIC     90 Protected Payload (KP0), DCID=be3ac0381d9049e8, PKN: 11, ACK, STREAM(…
   19 2.927297   127.0.0.1  127.0.0.1    QUIC     81 Protected Payload (KP0), DCID=e77ce754b2d32f6c, PKN: 9, ACK
   20 2.942133   127.0.0.1  127.0.0.1    QUIC     97 Protected Payload (KP0), DCID=be3ac0381d9049e8, PKN: 12, NCI
   21 2.944590   127.0.0.1  127.0.0.1    QUIC     81 Protected Payload (KP0), DCID=e77ce754b2d32f6c, PKN: 10, ACK

  ACK
      Frame Type: ACK (0x0000000000000002)
      Largest Acknowledged: 8
      ACK Delay: 46
      ACK Range Count: 0
      First ACK Range: 3
  TIME_STAMP
      Frame Type: TIME_STAMP (0x00000000000002f5)
      Time Stamp: 365556
  STREAM id=0 fin=1 off=0 len=7 uni=0
      Frame Type: STREAM (0x000000000000000b)
          .... ...1 = Fin: True
          .... ..1. = Len(gth): True
          .... .0.. = Off(set): False
      Stream ID: 0
      Length: 7
      Stream Data: 216f6c6c65480a

0000  02 08 2e 00 03 42 f5 80  05 93 f4 0b 00 07 21 6f   ··.··B·· ······!o
0010  6c 6c 65 48 0a                                     lleH·
```

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
```

(continues on next page)

(continued from previous page)

```
{
    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

```
#include <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; this version is deprecated.

`LSQVER_ID29`
IETF QUIC version ID 29

`LSQVER_ID34`
IETF QUIC version ID 34

`LSQVER_I001`
IETF QUIC version 1. (This version is disabled by default until the QUIC RFC is released).

`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*, const struct sockaddr **local*)
: Get SSL_CTX associated with a peer context. Mandatory in server mode. This is used 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.

void **(*ea_generate_scid)**(*lsquic_conn_t* *, lsquic_cid_t *, unsigned)
: Optional interface to control the creation of connection IDs.

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_CL* or *LSQUIC_DF_INIT_MAX_STREAM_DATA_BIDI_REMOTE_SERVER*.

IETF QUIC only.

unsigned **es_init_max_stream_data_bidi_local**

Initial max stream data.

This is a transport parameter.

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

IETF QUIC only.

unsigned **es_init_max_stream_data_uni**

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

This is a transport parameter.

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

IETF QUIC only.

unsigned **es_init_max_streams_bidi**

Maximum initial number of bidirectional stream.

This is a transport parameter.

Default value is *LSQUIC_DF_INIT_MAX_STREAMS_BIDI*.

IETF QUIC only.

unsigned **es_init_max_streams_uni**

Maximum initial number of unidirectional stream.

This is a transport parameter.

Default value is *LSQUIC_DF_INIT_MAX_STREAMS_UNI_CLIENT* or *LSQUIC_DF_INIT_MAX_STREAM_DATA_UNI_SERVER*.

IETF QUIC only.

unsigned **es_idle_timeout**

Idle connection timeout.

This is a transport parameter.

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

Default value is *LSQUIC_DF_IDLE_TIMEOUT*.

Maximum value is 600 seconds.

IETF QUIC only.

unsigned **es_ping_period**

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

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

IETF QUIC only.

unsigned **es_scid_len**

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

Default value is *LSQUIC_DF_SCID_LEN*.

IETF QUIC only.

unsigned **es_scid_iss_rate**

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

Default value is *LSQUIC_DF_SCID_ISS_RATE*.

IETF QUIC only.

unsigned **es_qpack_dec_max_size**

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

The default is *LSQUIC_DF_QPACK_DEC_MAX_SIZE*.

IETF QUIC only.

unsigned **es_qpack_dec_max_blocked**

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

The default is *LSQUIC_DF_QPACK_DEC_MAX_BLOCKED*.

IETF QUIC only.

unsigned **es_qpack_enc_max_size**

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

The default is *LSQUIC_DF_QPACK_ENC_MAX_SIZE*.

IETF QUIC only.

unsigned **es_qpack_enc_max_blocked**

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

The default is *LSQUIC_DF_QPACK_ENC_MAX_BLOCKED*.

IETF QUIC only.

int **es_ecn**

Enable ECN support.

The default is *LSQUIC_DF_ECN*

IETF QUIC only.

int **es_allow_migration**

Allow peer to migrate connection.

The default is *LSQUIC_DF_ALLOW_MIGRATION*

IETF QUIC only.

unsigned **es_cc_algo**

Congestion control algorithm to use.

- 0: Use default (*LSQUIC_DF_CC_ALGO*)
- 1: Cubic
- 2: 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.

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. **RFC 8899** 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*

int **es_ext_http_prio**

If set to true, Extensible HTTP Priorities are enabled. This is HTTP/3-only setting.

Default value is *LSQUIC_DF_EXT_HTTP_PRIO*

int **es_qpack_experiment**

If set to 1, QPACK statistics are logged per connection.

If set to 2, QPACK experiments are run. In this mode, encoder and decoder setting values are randomly selected (from the range [0, whatever is specified in es_qpack_(encldec)_*]) and these values along with compression ratio and user agent are logged at NOTICE level when connection is destroyed. The purpose of these experiments is to use compression performance statistics to figure out a good set of default values.

Default value is *LSQUIC_DF_QPACK_EXPERIMENT*

int **es_delay_onclose**

When set to true, *lsquic_stream_if.on_close* will be delayed until the peer acknowledges all data sent on the stream. (Or until the connection is destroyed in some manner – either explicitly closed by the user or as a result of an engine shutdown.) To find out whether all data written to peer has been acknowledged, use *lsquic_stream_has_unacked_data()*.

Default value is *LSQUIC_DF_DELAY_ONCLOSE*

int **es_max_batch_size**

If set to a non-zero value, specifies maximum batch size. (The batch of packets passed to *lsquic_engine_api.ea_packets_out*). Must be no larger than 1024.

Default value is *LSQUIC_DF_MAX_BATCH_SIZE*

int **es_check_tp_sanity**
When true, sanity checks are performed on peer's transport parameter values. If some limits are set suspiciously low, the connection won't be established.

Default value is *LSQUIC_DF_CHECK_TP_SANITY*

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`
The Delayed ACKs extension is on 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.

`LSQUIC_DF_EXT_HTTP_PRIO`
Turn on Extensible HTTP Priorities by default.

`LSQUIC_DF_QPACK_EXPERIMENT`
By default, QPACK experiments are turned off.

`LSQUIC_DF_DELAY_ONCLOSE`
By default, calling `lsquic_stream_if.on_close()` is not delayed.

`LSQUIC_DF_MAX_BATCH_SIZE`
By default, maximum batch size is not specified, leaving it up to the library.

`LSQUIC_DF_CHECK_TP_SANITY`
Transport parameter sanity checks are performed 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_reset)`** (*lsquic_stream_t* **s*, *lsquic_stream_ctx_t* **h*, int *how*)

This callback is called as soon as the peer resets a stream. The argument `how()` is either 0, 1, or 2, meaning “read”, “write”, and “read and write”, respectively (just like in `shutdown(2)`). This signals the user to stop reading, writing, or both.

Note that resets differ in gQUIC and IETF QUIC. In gQUIC, `how()` is always 2; in IETF QUIC, `how()` is either 0 or 1 because one can reset just one direction in IETF QUIC.

This callback is optional. The reset error can still be collected during next “on read” or “on write” event.

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.

  For IETF QUIC, this is called only if `lsquic_engine_api.ea_get_ssl_ctx_st` is *not* set, in which case the library creates its own SSL_CTX.

  Note: this callback will be deprecated when gQUIC support is removed.

  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.

void **lsquic_conn_abort**(*lsquic_conn_t *conn*)

This aborts the connection. The connection and all associated objects will be destroyed (with necessary callbacks called) during the next time *lsquic_engine_process_conns()* is invoked.

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.

const char * **lsquic_conn_get_sni** (*lsquic_conn_t* **conn*)
: Get SNI sent by the client.

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.

int **lsquic_stream_has_unacked_data** (const *lsquic_stream_t* **stream*)
: Return true if peer has not ACKed all data written to the stream. This includes both packetized and buffered data.

3.3.27 Other Functions

lsquic_conn_t **lsquic_ssl_to_conn** (const SSL *)
: Get connection associated with this SSL object.

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*, *lsquic_conn_get_ctx* **conn_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.

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.
- *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 Extensible HTTP Priorities

lsquic supports the Extensible HTTP Priorities Extension. It is enabled by default when HTTP/3 is used. The "urgency" and "incremental" parameters are included into a dedicated type:

struct **`lsquic_ext_http_prio`**

unsigned char **`urgency`**
This value's range is [0, 7], where 0 is the highest and 7 is the lowest urgency.

signed char **`incremental`**
This is a boolean value. The valid range is [0, 1].

Some useful macros are also available:

`LSQUIC_MAX_HTTP_URGENCY`

The maximum value of the "urgency" parameter is 7.

`LSQUIC_DEF_HTTP_URGENCY`

The default value of the "urgency" parameter is 3.

`LSQUIC_DEF_HTTP_INCREMENTAL`

The default value of the "incremental" parameter is 0.

There are two functions to manage a stream's priority:

int **`lsquic_stream_get_http_prio`** (*lsquic_stream_t *stream*, struct *lsquic_ext_http_prio *ehp*)
Get a stream's priority information.

Parameters

- **`stream`** – The stream whose priority informaion we want.

- **ehp** – Structure that is to be populated with the stream's priority information.

Returns Returns zero on success of a negative value on failure. A failure occurs if this is not an HTTP/3 stream or if Extensible HTTP Priorities have not been enabled. See *lsquic_engine_settings.es_ext_http_prio*.

int **lsquic_stream_set_http_prio**(*lsquic_stream_t *stream*, const struct *lsquic_ext_http_prio *ehp*)

Set a stream's priority information.

Parameters

- **stream** – The stream whose priority we want to set.
- **ehp** – Structure containing the stream's new priority information.

Returns Returns zero on success of a negative value on failure. A failure occurs if some internal error occured or if this is not an HTTP/3 stream or if Extensible HTTP Priorities haven't been enabled. See *lsquic_engine_settings.es_ext_http_prio*.

3.3.32 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 Developing lsquic

3.4.1 Generating Tags

Over the years, we have developed a wrapper around Universal Ctags to generate convenient tags so that, for example, `ci_packet_in` will be able to take you to any of its implementations such as `full_conn_ci_packet_in()`, `evanescent_conn_ci_packet_in()`, and others.

Exuberant Ctags will work, too, but the more recent and maintained fork of it, the _Universal_ Ctags, is preferred. (If you are on Ubuntu, you should clone Universal Ctags from GitHub and compiled it yourself. The version that comes in the Ubuntu package – at the time of this writing – is so slow as to be considered broken).

The wrapper is `tools/gen-tags.pl`. Run it in the source directory:

```
sh$ cd lsquic
sh$ ./tools/gen-tags.pl
```

3.4.2 Maintaining Documentation

Documentation – the `*.rst` files under `docs/` should be kept up-to-date with changes in the API in `include/lsquic.h`.

For convenience, tags for the documentation files can be generated by passing the `--docs` argument to `tools/gen-tags.pl`.

3.5 Library Guts

3.5.1 Introduction

lsquic inception dates back to the fall of 2016. Since that time, lsquic underwent several major changes. Some of those had to do with making the library more performant; others were needed to add important new functionality (for example, IETF QUIC and HTTP/3). Throughout this time, one of the main principles we embraced is that **performance trumps everything else**, including code readability and maintainability. This focus drove code design decisions again and again and it explains some of the hairiness that we will come across in this document.

Code Version

The code version under discussion is v2.29.6.

3.5.2 Coding Style

Spacing and Cuddling

lsquic follows the LiteSpeed spacing and cuddling conventions:

- Two empty lines between function definitions
- Four-space indentation
- Ifs and elses are not cuddled

Function Name Alignment

In function definitions, the name is always left-aligned, for example:

```
static void
check_flush_threshold (lsquic_stream_t *stream)
```

Naming Conventions

- Struct members usually have prefixes derived from the struct name. For example members of `struct qpack_dec_hdl` begin with `qdh_`, members of `struct cid_update_batch` begin with `cub_`, and so on. This is done to reduce the need to memorize struct member names as vim's autocomplete (Ctrl-P) functionality makes it easy to fill in the needed identifier.
- Non-static functions all begin with `lsquic_`.

- Functions usually begin with a module name (e.g. `lsquic_engine_` or `stream_`) which is then followed by a verb (e.g. `lsquic_engine_connect` or `stream_activate_hq_frame`). If a function does not begin with a module name, it begins with a verb (e.g. `check_flush_threshold` or `maybe_remove_from_write_q`).
- Underscores are used to separate words (as opposed to, for example, theCamelCase).

Typedefs

Outside of user-facing API, structs, unions, and enums are not typedefed. On the other hand, some integral types are typedefed.

3.5.3 List of Common Terms

- **gQUIC** Google QUIC. The original lsquic supported only Google QUIC. gQUIC is going to become obsolete. (Hopefully soon).
- **HQ** This stands for "HTTP-over-QUIC", the original name of HTTP/3. The code predates the official renaming to HTTP/3 and thus there are many types and names with some variation of `HQ` in them.
- **iQUIC** This stands for IETF QUIC. To differentiate between gQUIC and IETF QUIC, we use `iquic` in some names and types.
- **Public Reset** In the IETF QUIC parlance, this is called the *stateless* reset. Because gQUIC was first to be implemented, this name is still used in the code, even when the IETF QUIC stateless reset is meant. You will see names that contain strings like "prst" and "pubres".

3.5.4 High-Level Structure

At a high level, the lsquic library can be used to instantiate an engine (or several engines). An engine manages connections; each connection has streams. Engine, connection, and stream objects are exposed to the user who interacts with them using the API (see *API Reference*). All other data structures are internal and are hanging off, in one way or another, from the engine, connection, or stream objects.

3.5.5 Engine

Files: lsquic_engine.c, lsquic_engine_public.h, lsquic.h

Data Structures

out_batch

```
/* The batch of outgoing packets grows and shrinks dynamically */
/* Batch sizes do not have to be powers of two */
#define MAX_OUT_BATCH_SIZE 1024
#define MIN_OUT_BATCH_SIZE 4
#define INITIAL_OUT_BATCH_SIZE 32

struct out_batch
{
    lsquic_conn_t           *conns   [MAX_OUT_BATCH_SIZE];
```

(continues on next page)

(continued from previous page)

```
    struct lsquic_out_spec   outs    [MAX_OUT_BATCH_SIZE];
    unsigned                 pack_off[MAX_OUT_BATCH_SIZE];
    lsquic_packet_out_t     *packets[MAX_OUT_BATCH_SIZE * 2];
    struct iovec             iov     [MAX_OUT_BATCH_SIZE * 2];
};
```

The array of struct lsquic_out_specs – outs above – is what gets passed to the user callback `ea_packets_out()`. `conns` array corresponds to the spec elements one to one.

`pack_off` records which packet in `packets` corresponds to which connection in `conns`. Because of coalescing, an element in `outs` can correspond (logically) to more than one packet. (See how the batch is constructed in *Batching packets*.) On the other hand, `packets` and `iov` arrays have one-to-one correspondence.

There is one instance of this structure per engine: the whole thing is allocated as part of *struct lsquic_engine*.

cid_update_batch

```
struct cid_update_batch
{
    lsquic_cids_update_f    cub_update_cids;
    void                   *cub_update_ctx;
    unsigned                cub_count;
    lsquic_cid_t            cub_cids[20];
    void                   *cub_peer_ctxs[20];
};
```

This struct is used to batch CID updates.

There are three user-defined CID liveness callbacks: `ea_new_scids`, `ea_live_scids`, and `ea_old_scids`. These functions all have the same signature, `lsquic_cids_update_f`. When the batch reaches the count of 20 (kept in `cub_count`), the callback is called.

The new SCIDs batch is kept in *struct lsquic_engine*. Other batches are allocated on the stack in different functions as necessary.

20 is an arbitrary number.

lsquic_engine_public

This struct, defined in lsquic_engine_public.h, is the "public" interface to the engine. ("Public" here means accessible by other modules inside lsquic, not that it's a public interface like the *API Reference*.) Because there are many things in the engine object that are accessed by other modules, this struct is used to expose those (`public`) parts of the engine.

`lsquic_engine_struct` is the first member of *lsquic_engine*. The functions declared in lsquic_engine_public.h take a pointer to lsquic_engine_public as the first argument, which is then case to lsquic_engine.

This is somewhat ugly, but it's not too bad, as long as one remembers that the two pointers are interchangeable.

lsquic_engine

This is the central data structure. The engine instance is the root of all other data structures. It contains:

- Pointers to connections in several lists and hashes (see *Connection Management*)

- Memory manager
- Engine settings
- Token generator
- CID Purgatory
- Server certificate cache
- Transport parameter cache
- Packet request queue
- *Outgoing packet batch*
- And several other things

Some of the members above are stored in the `pub` member of type *lsquic_engine_public*. These are accessed directly from other parts of lsquic.

The engine is instantiated via `lsquic_engine_new()` and destroyed via `lsquic_engine_destroy()`

Connection Management

Lifetime

There are several *connection types*. All types of connections begin their life inside the engine module, where their constructors are called. They all also end their life here as well: this is where the destructors are called.

The connection constructors are all different function calls:

- lsquic_ietf_full_conn_client_new
- lsquic_gquic_full_conn_client_new
- lsquic_ietf_full_conn_server_new
- lsquic_gquic_full_conn_server_new
- lsquic_mini_conn_ietf_new
- lsquic_mini_conn_new
- lsquic_prq_new_req
- lsquic_prq_new_req_ext

(See *Evanescent Connection* for information about the last two.)

After a connection is instantiated, all further interactions with it, including destruction, are done via the *Common Connection Interface*.

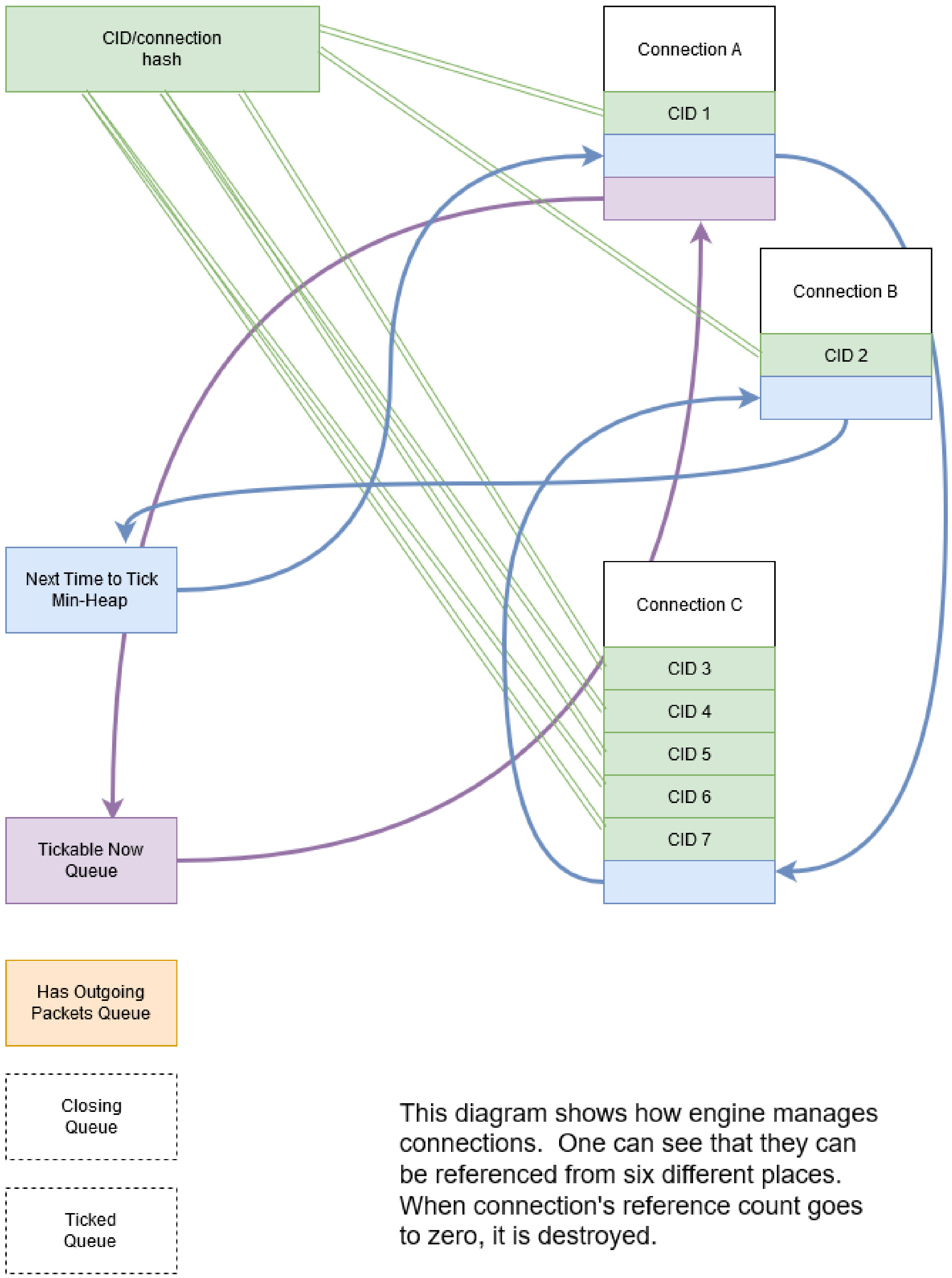
Refcounting Model

Each connection is referenced by at least one of the following data structures:

1. CID-to-connection hash. This hash is used to find connections in order to dispatch an incoming packet. Connections can be hashed by CIDs or by address. In the former case, each connection has one or more mappings in the hash table. IETF QUIC connections have up to eight (in our implementation) source CIDs and each of those would have a mapping. In client mode, depending on QUIC versions and options selected, it is may be necessary to hash connections by address, in which case incoming packets are delivered to connections based on the address.

2. Outgoing queue. This queue holds connections that have packets to send.
3. *Tickable Queue*. This queue holds connections that *can be ticked now*.
4. *Advisory Tick Time Queue*.
5. Closing connections queue. This is a transient queue – it only exists for the duration of *process_connections()* function call.
6. Ticked connections queue. Another transient queue, similar to the above.

The idea is to destroy the connection when it is no longer referenced. For example, a connection tick may return TICK_SEND|TICK_CLOSE. In that case, the connection is referenced from two places: (2) and (5). After its packets are sent, it is only referenced in (5), and at the end of the function call, when it is removed from (5), reference count goes to zero and the connection is destroyed. (See function `destroy_conn`.) If not all packets can be sent, at the end of the function call, the connection is referenced by (2) and will only be removed once all outgoing packets have been sent.



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.

In the diagram above, you can see that the CID-to-connection hash has several links to the same connection. This is because an IETF QUIC connection has more than one Source Connection IDs (SCIDs), any of which can be included by the peer into the packet. See `insert_conn_into_hash` for more details.

References from each of these data structures are tracked inside the connection object by bit flags:

```
#define CONN_REF_FLAGS  (LSCONN_HASHED          \
                        |LSCONN_HAS_OUTGOING    \
                        |LSCONN_TICKABLE        \
                        |LSCONN_TICKED          \
                        |LSCONN_CLOSING         \
                        |LSCONN_ATTQ)
```

Functions `engine_incref_conn` and `engine_decref_conn` manage setting and unsetting of these flags.

Notable Code

Handling incoming packets

Incoming UDP datagrams are handed off to the lsquic library using the function `lsquic_engine_packet_in`. Depending on the engine mode – client or server – the appropriate *packet parsing* function is selected.

Because a UDP datagram can contain more than one QUIC packet, the parsing is done in a loop. If the first part of packet parsing is successful, the internal `process_packet_in` function is called.

There, most complexity is contained in `find_or_create_conn`, which gets called for the server side. Here, parsing of the packet is finished, now via the version-specific call to `pf_parse_packet_in_finish`. If connection is not found, it may need to be created. Before that, the following steps are performed:

- Check that engine is not in the cooldown mode
- Check that the maximum number of mini connections is not exceeded
- Check that the (D)CID specified in the packet is not in the *CID Purgatory*
- Check that the packet can be used to create a mini conn: it contains version information and the version is supported
- Depending on QUIC version, perform token verification, if necessary

Only then does the mini connection constructor is called and the connection is inserted into appropriate structures.

Processing connections

Connections are processed in the internal function `process_connections`. There is the main connection processing loop and logic.

All connections that the iterator passed to this function returns are processed in the first while loop. The `ci_tick()` call is what causes the underlying connection to do all it needs to (most importantly, dispatch user events and generate outgoing packets). The return value dictates what lists – global and local to the function – the connection will be placed upon.

Note that mini connection promotion happens inside this loop. Newly created full connections are processed inside the same while loop. For a short time, a mini and a full connection object exist that are associated with the same logical connection.

After all connections are ticked, outgoing packets, if there are any, *are sent out*.

Then, connections that were closed by the first while loop above are finally closed.

Connections that were ticked (and not closed) are either:

- Put back onto the `tickable` queue;
- Added to the *Advisory Tick Time Queue*; or
- Left unqueued. This can happen when both idle and ping timer are turned off. (This should not happen for the connections that we expect to process, though.)

And lastly, CID liveness updates are reported to the user via the optional SCIDs callbacks: `ea_new_scids` etc.

Tickable Queue Cycle

When a connection is ticked, it is removed from the *Tickable Queue* and placed onto the transient Ticked Queue. After outgoing packets are sent and some connections are closed, the Ticked Queue is examined: the engine queries each remaining connection again whether it's tickable. If it is, back onto the Tickable Queue it goes. This should not happen often, however. It may occur when RTT is low and there are many connections to process. In that case, once all connections have been processed, the pacer now allows to send another packet because some time has passed.

Batching packets

Packet-sending entry point is the function `send_packets_out`. The main idea here is as follows:

Iterate over connections that have packets to send (those are on the Outgoing queue in the engine). For each connection, ask it for the next outgoing packet, encrypt it, and place it into the batch. When the batch is full, *send the batch*.

The outgoing packets from all connections are interleaved. For example, if connections A, B, and C are on the Outgoing queue, the batch will contain packets A1, B1, C1, A2, B2, C2, A3, B3, C3, … and so on. This is done to ensure fairness. When a connection runs out of packets to send, it returns NULL and is removed from the iterator.

The idea is simple, but the devil is in the details. The code may be difficult to read. There are several things going on:

Conns Out Iterator

This iterator, `conns_out_iter`, sends packets from connections on the Outgoing queue and packets on the Packet Request queue. (The latter masquerade as *Evanescent Connections* so that they are simple to use.) First, the Outgoing queue (which is a min-heap) is drained. Then, packets from the Packet Request queue are sent, if there are any. Then, remaining connections from the first pass are returned in the round-robin fashion.

After sending is completed, the connections that still have outgoing packets to send are placed back onto the Outgoing queue.

Packet Coalescing

Some IETF QUIC packets can be coalesced. This reduces the number of UDP datagrams that need to be sent during the handshake. To support this, if a packet matches some parameters, the same connection is queried for another packet, which, if it returns, is added to the current batch slot's iov.

```
if ((conn->cn_flags & LSCONN_IETF)
    && ((1 << packet_out->po_header_type)
      & ((1 << HETY_INITIAL)|(1 << HETY_HANDSHAKE)|(1 << HETY_0RTT)))
    && (engine->flags & ENG_COALESCE)
    && iov < batch->iov + sizeof(batch->iov) / sizeof(batch->iov[0]))
{
    const struct to_coal to_coal = {
```

(continues on next page)

(continued from previous page)

```
        .prev_packet = packet_out,
        .prev_sz_sum = iov_size(packet_iov, iov),
    };
    packet_out = conn->cn_if->ci_next_packet_to_send(conn, &to_coal);
    if (packet_out)
        goto next_coa;
}
batch->outs   [n].iovlen = iov - packet_iov;
```

With some debug code removed for simplicity

Also see the description of the batch in *out_batch*.

Note that packet coalescing is only done during the handshake of an IETF QUIC connection. Non-handshake and gQUIC packets cannot be coalesced.

Sending and Refilling the Batch

When the batch is sent inside the while loop, and the whole batch was sent successfully, the batch pointers are reset, the batch potentially grows larger, and the while loop continues.

Batch Resizing

When all datagrams in the batch are sent successfully, the batch may grow – up to the hardcoded maximum value of `MAX_OUT_BATCH_SIZE`. When not all datagrams are sent, the batch shrinks. The batch size survives the call into the library: when packets are sent again, the same batch size is used to begin the sending.

Deadline Checking

This is a rather old safety check dating back to the summer of 2017, when we first shipped QUIC support. The way we send packets has changed since then – there is high possibility that this code can be removed with no ill effect.

Sending a batch

When the batch is filled, it is handed off to the function `send_batch`, which calls the user-supplied callback to send packets out. The high-level logic is as follows:

- Update each packet's `sent` time
- Call the "send packets out" callback
- For packets that were sent successfully, call `ci_packet_sent`
- For packets that were not sent, call `ci_packet_not_sent`. This is important: all packets returned by `ci_next_packet_to_send` must be returned to the connection via either these two calls above or via `ci_packet_too_large` (see below).
- Return the number of packets sent

Because of support for coalescing, we have to map from outgoing spec to packets via `batch->pack_off`. This is done in several places in this function.

To handle the case when a PMTU probe is too large (stuff happens!), the code checks for EMSGSIZE and returns the packet back to the connection via `ci_packet_too_large`. Because this error is of our own making, this does not count as inability to send. The too-large packet is skipped and sending of the datagrams in the batch continues.

Growing min-heaps

The Outgoing and Tickable connection queues are actually min-heaps. The number of elements in these min-heaps never exceeds the number of connections. As optimization, allocation of the underlying arrays is done not in the min-heap module itself but in the engine module in the function `maybe_grow_conn_heaps`. The engine knows how many connections there are and it grows the arrays as necessary.

As an additional optimization, the two arrays use a single memory region which is allocated once.

The min-heap arrays are never shrunk.

3.5.6 Connection

Files: lsquic_conn.h, lsquic_conn.c – others are covered in dedicated chapters

The connection represents the QUIC connection. Connections are *managed by the engine*. A connection, in turn, manages *streams*.

Connection Types

lsquic supports two different QUIC protocols: Google QUIC and IETF QUIC. Each of these has a separate implementation, which includes connection logic, parsing/generating mechanism, and encryption.

Each of the QUIC connection types on the server begin their life as a `mini` connection. This connection type is used while handshake is proceeding. Once the handshake has completed, the mini connection is `promoted` to a `full` connection. (See *Mini vs Full Connection* for more.)

In addition to the above, an "evanescent" connection type is used to manage replies to incoming packets that do not result in connection creation. These include version negotiation, stateless retry, and stateless reset packets.

Each of the five connection types above are covered in their own dedicated chapters elsewhere in this document:

- *Mini gQUIC Connection*
- *Full gQUIC Connection*
- *Mini IETF QUIC Connection*
- *Full IETF QUIC Connection*
- *Evanescent Connection*

lsquic_conn

All connection types expose the same connection interface via a pointer to `struct lsquic_conn`. (This is the same type pointer to which is exposed to the user, but the user can only treat the connection as an opaque pointer.)

This structure contains the following elements:

Pointers to Crypto Implementation

The crypto session pointer, `cn_enc_session`, points to a type-specific (gQUIC or iQUIC) instance of the encryption session. This session survives *connection promotion*.

The two types of crypto session have a set of common functionality; it is pointed to by `cn_esf_c` (where `c` stands for `common`). Each of them also has its own, type-specific functionality, which is pointed to by `cn_esf.g` and `cn_esf.i`

Pointer to Common Connection Interface

`cn_if` points to the set of functions that implement the Common Connection Interface (*see below*).

Pointer to Parsing Interface

The parsing interface is version-specific. It is pointed to by `cn_pf`.

Various list and heap connectors

A connection may be pointed to by one or several queues and heaps (see "*Connection Management*"). There are several struct members that make it possible: *TAILQ_ENTRYs, `cn_attq_elem`, and `cn_cert_susp_head`.

Version

`cn_version` is used to make some decisions in several parts of the code.

Flags

The flags in `cn_flags` specify which lists the connection is on and some other properties of the connection which need to be accessible by other modules.

Stats

`cn_last_sent` and `cn_last_ticked` are used to determine the connection's place on the outgoing queue (see *Batching Packets*) and on the *Advisory Tick Time Queue*.

List of SCIDs

IETF QUIC connections have one or more SCIDs (Source Connection IDs), any one of which can be used by the peer as the DCID (Destination CID) in the packets it sends. Each of the SCIDs is used to hash the connection so it can be found. `cn_cces` points to an array of size `cn_n_cces` which is allocated internally inside each connection type.

Google QUIC connections use only one CID (same for source and destination). In order not to modify old code, the macro `cn_cid` is used.

Common Connection Interface

The struct `conn_iface` defines the common connection interface. All connection types implement all or some of these functions.

Some of these functions are used by the engine; others by other modules (for example, to abort a connection); yet others are for use by the user, e.g. `lsquic_conn_close` and others in lsquic.h. In that case, these calls are wrapped in lsquic_conn.c.

Tickability

A connection is processed when it is tickable. More precisely, the connection is placed onto the *Tickable Queue*, which is iterated over when *connections are processed*. A connection reports its own tickability via the `ci_is_tickable` method.

In general, a connection is tickable if it has productive user callbacks to dispatch (that is, user wants to read and there is data to read or user wants to write and writing is possible), if there are packets to send or generate, or if its advisory tick time is in the past. (The latter is handled in `lsquic_engine_process_conns()` when expired connections from the *Advisory Tick Time Queue* are added to the Tickable Queue.)

3.5.7 Stream

Files: lsquic_stream.h, lsquic_stream.c

Overview

The lsquic stream is the conduit for data. This object is accessible by the user via any of the `lsquic_stream_*` functions declared in lsquic.h. The stream is bidirectional; in our user code, it represents the HTTP request and response. The client writes its request to the stream and the server reads the request in its corresponding instance of the stream. The server sends its response using the same stream, which the client reads from the stream.

Besides streams exposed to the application, connections use streams internally:

- gQUIC has the HANDSHAKE and HEADERS streams
- IETF QUIC has up to four HANDSHAKE streams
- HTTP/3 has at least three unidirectional streams:
  - Settings stream
  - QPACK encoder stream
  - QPACK decoder stream

In addition, HTTP/3 push promises use unidirectional streams. In the code, we make a unidirectional stream simply by closing one end in the constructor.

All of the use cases above are handled by the single module, lsquic_stream. The differences in behavior – gQUIC vs IETF QUIC, HTTP vs non-HTTP – are handled either by explicit conditionals or via function pointers.

The streams hang off full connections via stream ID-to-stream hashes and in various queues. This is similar to the way the connections hang off the engine.

Streams are only used in the full connections; mini connections use their own, minimalistic, code to handle streams.

Data Structures

stream_hq_frame

This structure is used to keep information about an HTTP/3 frame that is being, or is about to be, written. In our implementation, frame headers can be two or three bytes long: one byte is HTTP/3 frame type and the frame length is encoded in 1 or 2 bytes, giving us the maximum payload size of 2^{14} - 1 bytes. You will find literal `2` or `3` values in code that deals with writing HQ frames.

If the HQ frame's size is known in advance (SHF_FIXED_SIZE) – which is the case for HEADERS and PUSH_PROMISE frames – then the HQ header contents are written immediately. Otherwise, `shf_frame_ptr` points to the bytes in the packet where the HQ header was written, to be filled in later.

See *Writing HTTP/3 Streams* for more information.

hq_filter

This structure is used to read HTTP/3 streams. A single instance of it is stored in the stream in `sm_hq_filter`. The framing is removed transparently (see *Reading HTTP/3 Streams*).

Frame type and length are read into `hqfi_vint2_state`. Due to greasing, the reader must be able to support arbitrary frame types and so the code is pretty generic: varints of any size are supported.

`hqfi_flags` and `hqfi_state` contain information needed to resume parsing the frame header, as only partial data may have arrived.

`hqfi_hist_buf` and `hqfi_hist_idx` are used to record the last few incoming headers. This information is used to check for validity, as some sequences of HTTP/3 frames are invalid.

stream_filter_if

This struct is used to specify functionality required to strip arbitrary framing when reading from the stream. At the moment (and for the foreseeable future) only one mechanism is used: that to strip the HTTP/3 framing. At the time the code was written, however, the idea was to future-proof it in case we needed to support more than one framing format at a time.

lsquic_stream

This struct is the stream object. It contains many members that deal with

- Reading data
- Writing data
- Maintaining stream list memberships
- Enforcing flow control
- Dispatching read and write events
- Calling various user callbacks
- Interacting with HEADERS streams

The stream has an ID (`id`). It is used to hash the stream.

A stream can be on one or more lists: see `next_send_stream`, `next_read_stream`, and so on.

Incoming data is stored in `data_in`. Outgoing data is packetized immediately or buffered in `sm_buf`.

HTTP/3 frames that are being actively written are on the `sm_hq_frames` list.

A note on naming: newer members of the stream begin with `sm_` for simplicity. Originally, the structure members lacked a prefix.

progress

This structure is used to determine whether the user callback has made any progress during an `on_write` or `on_read` event loop. If progress is not made for a number of calls, the callback is interrupted, breaking out of a suspected infinite loop. (See `es_progress_check` setting.)

frame_gen_ctx

This structure holds function pointers to get user data and write it to packets. `fgc_size`, `fgc_fin`, and `fgc_read` are set based on framing requirements. This is a nice abstraction that gets passed to several packetization functions and allows them not to care about how or whether framing is performed.

pwritev_ctx

Used to aid `lsquic_stream_pwritev`. `hq_arr` is used to roll back HTTP/3 framing if necessary. (The rollback is the most complicated part of the `pwritev` functionality).

Event Dispatch

The "on stream read" and "on stream write" callbacks are part of the lsquic API. These callbacks are called when the user has registered interest in reading from or writing to the stream and reading or writing is possible.

Calling `lsquic_stream_wantwrite` and `lsquic_stream_wantread` places the stream on the corresponding "want to write" and "want to read" list. These lists are processed by a connection when it's ticked. For each stream on the list, the internal function `lsquic_stream_dispatch_read_events` or `lsquic_stream_dispatch_write_events`, whichever may be the case.

Dispatching read events is simple. When `es_rw_once` is set, the "on stream read" callback is called once – if the stream is readable. Otherwise, the callback is called in a loop as long as:

- The stream is readable;
- The user wants to read from it; and
- Progress is being made

Dispatching write events is more complicated due to the following factors:

- In addition to calling the "on stream write" callback, the flushing mechanism also works by using the "want to write" list.
- When writing occurs, the stream's position on the list may change

STREAM frames in

The data gets in from the transport into the stream via `lsquic_stream_frame_in` function. The connection calls this function after parsing a STREAM frame.

The data from the STREAM frame is stored in one of the two "data in" modules: `di_nocopy` and `di_hash`. The two are abstracted out behind `stream->data_in`.

The "data in" module is used to store incoming stream data. The data is read from this module using the `di_get_frame` function. See the next section.

Reading Data

There are three user-facing stream-reading functions; two of them are just wrappers around `"lsquic_stream_readf`. This function performs some checks (we will cover HTTP mode separately) and calls `lsquic_stream_readf`, which also performs some checks and calls `read_data_frames`. This is the only function in the stream module where data is actually read from the "data in" module.

Writing Data

There are four user-facing functions to write to stream, and all of them are wrappers around `stream_write`. (`lsquic_stream_pwritev` is a bit more involved than the other three, but it's pretty well-commented – and the complexity is in the rollback, not writing itself.)

Small writes get buffered. If the write size plus whatever is buffered already exceeds the threshold – which is the size of the largest STREAM frame that could be fit into a single outgoing packet – the data is packetized instead by calling `stream_write_to_packets`. See the next section.

Packetization

`stream_write_to_packets` is the only function through which user data makes it into outgoing packets. There are three ways to write STREAM frames:

1. `stream_write_to_packet_hsk`
2. `stream_write_to_packet_std`
3. `stream_write_to_packet_crypto`

The particular function is selected based on connection and stream type when the stream is first created.

stream_write_to_packets

Depending on the need to frame data, a reader is selected. The job of the reader is to copy user data into the outgoing STREAM frame. In HTTP/3 mode, HTTP/3 framing is added transparently – see *Writing HTTP/3 Streams* for more information.

The while loop is entered if there is user data to be copied or if the end of the stream has been reached and FIN needs to be written. Note the threshold check: when writing data from a user call, the threshold is set and frames smaller than the full packet are not generated. This is to allow for usage like "write 8KB", "write 8KB", "write 8KB" not to produce jagged STREAM frames. This way, we utilize the bandwidth most effectively. When flushing data, the threshold is not set, so even a 1-byte data gets packetized.

The call `stream->sm_write_to_packet` writes data to a single packet. This packet is allocated by the *Send Controller*. (Depending on when writing is performed, the returned packet may be placed onto the scheduled queue immediately or it may be a "buffered" packet. The stream code is oblivious to that.) If the send controller does not give us a packet, STOP is returned and the while loop exits. An ERROR should never happen – this indicates a bug or maybe failure to allocate memory – and so the connection is aborted in that case. If everything is OK, the while loop goes on.

The `seen_ok` check is used to place the connection on the tickable list on the first successfully packetized STREAM frame. This is so that if the packet is buffered (meaning that the writing is occurring outside of the callback mechanism), the connection will be processed (ticked) and the packets will be scheduled and sent out.

After the while loop, we conditionally close an outstanding HTTP/3 frame, save any leftover data, schedule STREAM_BLOCKED or BLOCKED frames to be sent out if needed, and return the number of user-provided bytes that were copied into the outgoing packets and into the internal stream buffer (leftovers).

Write a single STREAM frame

We will examine `stream_write_to_packet_std` as it is the most complicated of these three functions.

First, we flush the headers stream if necessary – this is because we want the HTTP (gQUIC or HTTP/3) headers to be sent before the payload.

Then, the number of bytes needed to generate a STREAM frame is calculated. This value depends on the QUIC version, whether we need to generate HTTP/3 framing, and whether the data to write exists (or we just need to write an empty STREAM frame with the FIN bit set).

(Note that the framing check is made to overshoot the estimate for simplicity. For one, we might not need 3 bytes for the DATA frame, but only 2. Secondly, there may already be an open HTTP/3 frame in one of the previous packets and so we don't need to write it at all.)

Then, a packet is allocated and `write_stream_frame` is called. It is in this function that we finally make the call to generate the STREAM frame and to copy the data from the user. The function `pf_gen_stream_frame` returns the number of bytes actually written to the packet: this includes both the STREAM frame header and the payload (which may also include HTTP/3 frame).

The fact that this frame type has been written is added to `po_frame_types` and the STREAM frame location, type, and size are recorded. This information is necessary to be able to elide the frame from the packet in case the stream is reset.

`PO_STREAM_END` is set if the STREAM frame extends to the end of the packet. This is done to prevent this packet from being used again to append frames to it (after, for example, some preceding frames are elided from it). This is because both in gQUIC and IETF QUIC the STREAM frame header is likely to omit the `length` field and instead use the "extends to the end of the packet" field. If frames are shifted, the packet cannot be appended to because it will lead to data loss and corruption.

Writing HTTP/3 Streams

HTTP/3 streams use framing. In most cases, a single HEADERS frame is followed by zero or more DATA frames. The user code does not know this: both gQUIC and IETF QUIC streams appear to behave in exactly the same manner. This makes lsquic simple to use.

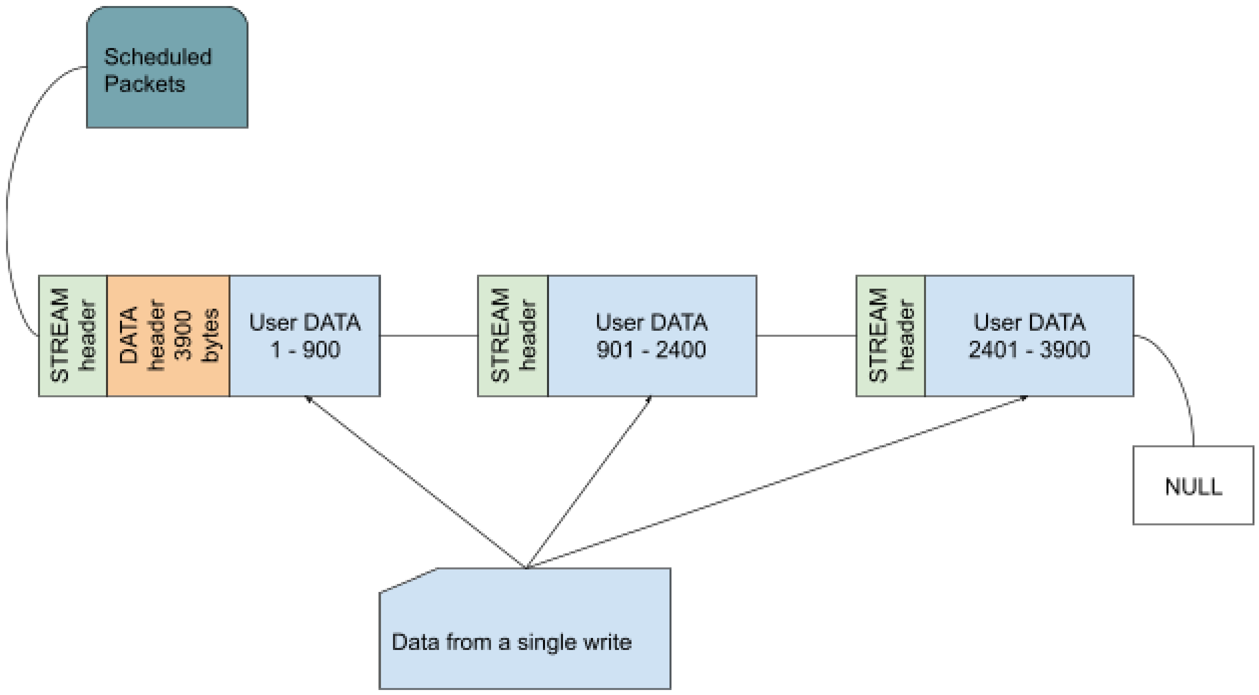
The drawback is internal complexity. To make the code both easy to use and performant, HTTP/3 framing is generated on-the-fly, as data is being written to packets (as opposed to being buffered and then written). (OK, *mostly* on-the-fly: the HEADERS frame payload is generated and then copied.)

On the high level, the way it works is as follows:

- When a write call is made, a variable-size (that is, unknown size; it's called variable-size because the size of the DATA header may be 2 or 3 bytes; it's not the best name in the world) frame is opened/activated.

- When data is written to stream, the DATA header placeholder bytes are written to the stream transparently and a pointer is saved to this location.
- The active frame header is closed when
  - It reaches its maximum size; or
  - The data we are writing runs out.
- When the header is closed, the number of bytes that follows is now written to the location we saved when the header was activated.

This mechanism allows us to create a DATA frame that spans several packets before we know how many packets there will be in a single write. (As outgoing packet allocation is governed by the *Send Controller*.) This is done to minimize the goodput overhead incurred by the DATA frame header.



There are a couple of things that do not fit into this model:

1. The HEADERS frame is fixed size[1]. It is generated separately (written by QPACK encoder into a buffer on the stack) and later copied into the stream. (See the `send_headers_ietf` function.) It can happen that the whole buffer cannot be written. In that case, a rather complicated dance of buffering the unwritten HEADERS frame bytes is performed. Here, the "on stream write" callback is replaced with an internal callback (see the `select_on_write` function) and user interaction is prohibited until the whole of the HEADERS frame is written to the stream.
2. Push promise streams are even weirder. In addition to the HEADERS handling above, the push promise stream must begin with a variable-integer Push ID. To make this fit into the framed stream model, the code makes up the concept of a "phantom" HTTP/3 frame. This type of frame's header is not written. This allows us to treat the Push ID as the payload of a regular HTTP/3 frame.

The framing code has had its share of bugs. Because of that, there is a dedicated unit test program just for the framing code, *tests/test_h3_framing.c*. In addition to manually-written tests, the program has a "fuzzer driver" mode, in which the American Fuzzy Lop fuzzer drives the testing of the HTTP/3 framing mechanism. The advantage of this approach is that AFL tries to explore all the code paths.

[1] This is due to the limitation of the QPACK library: the decoder can read input one byte at a time, but the encoder cannot write output one byte at a time. It could be made to do that, but the effort is not worth it.

Debates regarding DATA framing raged in 2018 on the QUIC mailing list. Some of the discussion is quite interesting: for example, the debate about "optimizing" DATA frames and calculations of the header cost.

Reading HTTP/3 Streams

HTTP/3 frame headers are stripped out transparently – they are never seen by the user. From the user's perspective, the lsquic stream represents the payload of HTTP message; a dedicated call is made first to get at the HTTP headers.

To accomplish this, the stream implements a generic deframing mechanism. The *stream_filter_if* interface allows one to specify functions to a) check whether the stream is readable, b) strip header bytes from a data frame fetched from "data in" module; and c) update byte count in the filter once bytes have been read:

hq_filter_readable

This function tests for availability of non-frame-header data, stripping frame headers from the stream transparently. Note how it calls `read_data_frames` with its own callback, `hq_read`. It is inside this callback that the HEADERS frame is fed to the QPACK decoder.

hq_filter_df

This function's job is to strip framing from data frames returned by the "data in" module inside the `read_data_frames` function. It, too, calls the `hq_read` function. This allows the two functions that read from stream (this one) and the readability-checking function (`hq_filter_readable`) to share the same state. This is crucial: Otherwise this approach is not likely to work well.

hq_decr_left

This function is needed to update the filter state. Once all payload bytes from the frame have been consumed, the filter is readied to strip the next frame header again.

Notable Code

frame_hq_gen_read

This is where HTTP/3 frame headers are generated. Note the use of `shf_frame_ptr` to record the memory location to which the correct frame size will be written by a different function.

3.5.8 Parsing

Files: lsquic_parse.h, lsquic_parse_ietf_v1.c, lsquic_parse_Q050.c, lsquic_parse_Q046.c, lsquic_parse_gquic_be.c, lsquic_parse_common.c, and others

Overview

The two types of QUIC – gQUIC and IETF QUIC – have different packet and frame formats. In addition, different gQUIC version are different among themselves. Functions to parse and generate packets and frames of each type are abstracted out behind the rather large `struct parse_funcs`. When a connection is created, its `cn_pf` member is set to point to the correct set of function pointers via the `select_pf_by_ver()` macro.

Parsing Packets

Before settling on a particular set of parsing function for a connection, the server needs to determine the connection's version. It does so using the function `lsquic_parse_packet_in_server_begin()`.

This function figures out whether the packet has a long or a short header, and which QUIC version it is. Because the server deals with fewer packet types than the client (no version negotiation or stateless retry packets), it can determine the necessary parsing function from the first byte of the incoming packet.

The "begin" in the name of the function refers to the fact that packet parsing is a two-step process[3]. In the first step, the packet version, CID, and some other parameters are parsed out; in the second step, version-specific `pf_parse_packet_in_finish()` is called to parse out the packet number. Between the two calls, the state is saved in `struct packin_parse_state`.

Generating Packets

Packets are generated during encryption using the `pf_gen_reg_pkt_header()` function. The generated header is encrypted together with the *packet payload* and this becomes the QUIC packet that is sent out. (Most of the time, the QUIC packet corresponds to the UDP datagram, but sometimes packets are *coalesced*.

Parsing Frames

There is a parsing function for each frame type. These function generally have names that begin with "pf_parse_" and follow a similar pattern:

- The first argument is the buffer to be parsed;
- The second argument is its size;
- Any additional arguments are outputs: the parsed out values from the frame;
- Number of bytes consumed is returned or a negative value is returned if a parsing error occurred.

For example:

```
int
(*pf_parse_stream_frame) (const unsigned char *buf, size_t rem_packet_sz,
                                                    struct stream_frame *);

int
(*pf_parse_max_data) (const unsigned char *, size_t, uint64_t *);
```

Generating Frames

Functions that generate frames begin with "pf_gen_" and also follow a pattern:

- First argument is the buffer to be written to;
- The second argument is the buffer size;
- Any additional arguments specify the values to include in the frame;
- The size of the resulting frame is returned or a negative value if an error occurred.

For example:

[3] This two-step packet parsing mechanism is left over from the little-endian to big-endian switch in gQUIC several years ago: Before parsing out the packet number, it was necessary to know whether it is little- or big-endian. It should be possible to do away with this, especially once gQUIC is gone.

```
int
(*pf_gen_path_chal_frame) (unsigned char *, size_t, uint64_t chal);

int
(*pf_gen_stream_frame) (unsigned char *buf, size_t bufsz,
                        lsquic_stream_id_t stream_id, uint64_t offset,
                        int fin, size_t size, gsf_read_f, void *stream);
```

Frame Types

Frame types are listed in `enum quic_frame_type`. When frames are parsed, the on-the-wire frame type is translated to the enum value; when frames are generated, the enum is converted to the on-the-wire format. This indirection is convenient, as it limits the range of possible QUIC frame values, making it possible to store a list of frame types as a bitmask. Examples include `po_frame_types` and `sc_retx_frames`.

Some frame types, such as ACK and STREAM, are common to both Google and IETF QUIC. Others, such as STOP_WAITING and RETIRE_CONNECTION_ID, are only used in one of the protocols. The third type is frames that are used by IETF QUIC extensions, such as TIMESTAMP and ACK_FREQUENCY.

Parsing IETF QUIC Frame Types

Most IETF frame types are encoded as a single by on the wire (and all Google QUIC frames are). Some of them are encoded using multiple bytes. This is because, like the vast majority of all integral values in IETF QUIC, the frame type is encoded as a varint. Unlike the other integral values, however, the frame type has the unique property is that it must be encoded using the *minimal representation*: that is, the encoding must use the minimum number of bytes possible. For example, encoding the value 200 must use the two-byte varint, not four- or eight-byte version. This makes it possible to parse frame types once without having to reparse the frame type again in individual frame-parsing routines.

Frame type is parsed out in `ietf_v1_parse_frame_type()`. Because of the minimal encoding requirement, the corresponding frame-parsing functions know the number of bytes to skip for type, for example:

```
static int
ietf_v1_parse_frame_with_varints (const unsigned char *buf, size_t len,
            const uint64_t frame_type, unsigned count, uint64_t *vals[])
{
    /* --- 8< --- code removed */
    vbits = vint_val2bits(frame_type);
    p += 1 << vbits;                    // <=== SKIP FRAME TYPE
    /* --- 8< --- code removed */
}

static int
ietf_v1_parse_timestamp_frame (const unsigned char *buf,
                               size_t buf_len, uint64_t *timestamp)
{
    return ietf_v1_parse_frame_with_varints(buf, buf_len,
            FRAME_TYPE_TIMESTAMP, 1, (uint64_t *[]) { timestamp });
}
```

3.5.9 Mini vs Full Connections

Mini Purpose

The reason for having a mini connection is to conserve resources: a mini connection allocates a much smaller amount of memory. This protects the server from a potential DoS attack. The mini connection's job is to get the handshake to succeed, after which the connection is *promoted*.

Mini/Full Differences

Besides their size, the two connection types differ in the following ways:

Mini connections' lifespan is limited. If the handshake does not succeed within 10 seconds (configurable), the mini connection is destroyed.

A mini connection is only *tickable* if it has unsent packets.

Mini connections do not process packets that carry application (as opposed to handshake) data. The 0-RTT packet processing is deferred; these packets are stashed and handed over to the full connection during promotion.

Connection Promotion

A mini connection is promoted when the handshake succeeds. The mini connection reports this via the return status of `ci_tick` by setting the `TICK_PROMOTE` bit. The engine creates a new connection object and calls the corresponding server constructor. The latter copies all the relevant state information from mini to full connection.

For a time, two connection objects – one mini and one full – exist at the same state. Most of the time, the mini connection is destroyed within the same function call to `process_connections()`. If, however, the mini connection has unsent packets, it will remain live until those packets are sent successfully. Because the mini connection is by then removed from the CID-to-connection hash (`engine->conns_hash`), it will not receive any more incoming packets.

Also see *Connection Processing*.

3.5.10 Mini gQUIC Connection

Files: lsquic_mini_conn.h, lsquic_mini_conn.c

Overview

The original version of `struct mini_conn` fit into paltry 128 bytes. The desire to fit into 128 bytes[2] led to, for example, `mc_largest_recv` – in effect, a 3-byte integer! Since that time, the mini conn has grown to over 512 bytes.

Looking at the struct, we can see that a lot of other data structures are squeezed into small fields:

Received and sent packet history is each packed into a 64-bit integer, `mc_received_packnos` and `mc_sent_packnos`, respectively. The HEADERS stream offsets are handled by the two two-byte integers `mc_read_off` and `mc_write_off`.

[2] Mini conn structs are allocated out of the *Malo Allocator*, which used to be limited to objects whose size is a power of two, so it was either fitting it into 128 bytes or effectively doubling the mini conn size.

Notable Code

continue_handshake

This function constructs a contiguous buffer with all the HANDSHAKE stream chunks in order and passes it to `esf_handle_chlo()`. This is done because the gQUIC crypto module does not buffer anything: it's all or nothing.

The code has been written in a generic way, so that even many small packets can be reconstructed into a CHLO. The lsquic client can be made to split the CHLO by setting the max packet size sufficiently low.

sent/unsent packets

To conserve space, only a single outgoing packet header exists in the mini connection struct, `mc_packets_out`. To differentiate between packets that are to be sent and those that have already been sent, the `PO_SENT` flag is used.

3.5.11 Mini IETF Connection

Files: lsquic_mini_conn_ietf.h, lsquic_mini_conn_ietf.c

Overview

The IETF QUIC mini connection has the same idea as the gQUIC mini connection: use as little memory as possible. This is more difficult to do with the IETF QUIC, however, as there are more moving parts in this version of the protocol.

Data Structures

mini_crypto_stream

This structure is a minimal representation of a stream. The IETF QUIC protocol uses up to four HANDSHAKE streams (one for each encryption level) during the handshake and we need to keep track of them. Even a basic event dispatch mechanism is supported.

packno_set_t

This bitmask is used to keep track of sent, received, and acknowledged packet numbers. It can support up to 64 packet numbers: 0 through 63. We assume that the server will not need to send more than 64 packets to complete the handshake.

imc_recvd_packnos

Because the client is allowed to start its packet number sequence with any number in the [0, 2^{32}-1] range, the received packet history must be able to accommodate numbers larger than 63. To do that, the receive history is a union. If all received packet numbers are 63 or smaller, the packno_set_t bitmask is used. Otherwise, the receive history is kept in *Tiny Receive History* (trechist). The flag `IMC_TRECHIST` indicates which data structure is used.

ietf_mini_conn

This structure is similar to the gQUIC mini conn. It is larger, though, as it needs to keep track of several instances of things based on encryption level or packet number space.

`imc_cces` can hold up to three SCIDs: one for the original DCID from the client, one for SCID generated by the server, and one for when preferred address transport parameter is used. (The preferred address functionality is not compiled by default.)

ietf_mini_rechist

The receive history is in the header file because, in addition to generating the ACK frames in the IETF mini conn, it is used to migrate the receive history during promotion.

Notable Code

Switching to trechist

The switch to the Tiny Receive History happens when the incoming packet number does not fit into the bitmask anymore – see `imico_switch_to_trechist()`. To keep the trechist code exercised, about one in every 16 mini connection uses trechist unconditionally – see `lsquic_mini_conn_ietf_new()`.

crypto_stream_if

A set of functions to drive reading and writing CRYPTO frames to move the handshake along is specified. It is passed to the crypto session. After promotion, the full connection installs its own function pointers.

imico_read_chlo_size

This function reads the first few bytes of the first CRYPTO frame on the first HANDSHAKE stream to figure out the size of ClientHello. The transport parameters will not be read until the full ClientHello is available.

Duplicated Code

Some code has been copied from gQUIC mini connection. This was done on purpose, with the expectation that gQUIC is going away.

ECN Blackhole Detection

ECN blackhole at the beginning of connection is guessed at when none of packets sent in the initial batch were acknowledged. This is done by `imico_get_ecn()`. `lsquic_mini_conn_ietf_ecn_ok()` is also used during promotion to check whether to use ECN.

3.5.12 Connection Public Interface

Files: lsquic_conn_public.h

TODO

3.5.13 Full gQUIC Connection

Files: lsquic_full_conn.h, lsquic_full_conn.c

Overview

The full gQUIC connection implements the Google QUIC protocol, both server and client side. This is where a large part of the gQUIC protocol logic is contained and where everything – engine, streams, sending, event dispatch – is tied together.

Components

In this section, each member of the `full_conn` structure is documented.

fc_conn

The first member of the struct is the common connection object, *lsquic_conn*.

It must be first in the struct because the two pointer are cast to each other, depending on circumstances.

fc_cces

This array holds two connection CID elements.

The reason for having two elements in this array instead of one (even though gQUIC only uses one CID) is for the benefit of the client: In some circumstances, the client connections are hashed by the port number, in which case the second element is used to hash the port value. The relevant code is in lsquic_engine.c

fc_rechist

This member holds the *packet receive history*. It is used to generate ACK frames.

fc_stream_ifs

This three-element array holds pointers to stream callbacks and the stream callback contexts.

From the perspective of lsquic, Google QUIC has three stream types:

1. HANDSHAKE stream;
2. HEADERS stream; and
3. Regular (message, or request/response) streams.

The user provides stream callbacks and the context for the regular streams (3) in `ea_stream_if` and `ea_stream_if_ctx`.

The other two stream types are internal. The full connection specifies internal callbacks for those streams. One set handles the handshake and the other handles reading and writing of HTTP/2 frames: SETTINGS, HEADERS, and so on.

fc_send_ctl

This is the *Send Controller*. It is used to allocate outgoing packets, control sending rate, and process acknowledgements.

fc_pub

This member holds the *Connection Public Interface*.

fc_alset

This is the *Alarm Set*. It is used to set various timers in the connection and the send controller.

fc_closed_stream_ids

The two sets in this array hold the IDs of closed streams.

There are two of them because of the uneven distribution of stream IDs. It is more efficient to hold even and odd stream IDs in separate structures.

fc_settings

Pointer to the engine settings.

This member is superfluous – the settings can be fetched from `fc_enpub->enp_settings`.

fc_enpub

This points to the *engine's public interface*.

fc_max_ack_packno

Recording the maximum packet number that contained an ACK allows us to ignore old ACKs.

fc_max_swf_packno

This is the maximum packet number that contained a STOP_WAITING frame. It is used to ignore old STOP_WAITING frames.

fc_mem_logged_last

This timestamp is used to limit logging the amount of memory used to most once per second.

fc_cfg

This structure holds a few important configuration parameters. (Looks like `max_conn_send` is no longer used…)

fc_flags

The flags hold various boolean indicators associated with the full connections. Some of them, such as `FC_SERVER`, never change, while others change all the time.

fc_n_slack_akbl

This is the number of ackable (or, in the new parlance, *ack-eliciting*) packets received since the last ACK was sent.

This counter is used to decide whether an ACK should be sent (or, more precisely, queued to be sent) immediately or whether to wait.

fc_n_delayed_streams

Count how many streams have been delayed.

When `lsquic_conn_make_stream()` is called, a stream may not be created immediately. It is delayed if creating a stream would go over the maximum number of stream allowed by peer.

fc_n_cons_unretx

Counts how many consecutive unretransmittable packets have been sent.

fc_last_stream_id

ID of the last created stream.

Used to assign ID to streams created by this side of the connection. Clients create odd-numbered streams, while servers initiate even-numbered streams (push promises).

fc_max_peer_stream_id

Maximum value of stream ID created by peer.

fc_goaway_stream_id

Stream ID received in the GOAWAY frame.

This ID is used to reset locally-initiated streams with ID larger than this.

fc_ver_neg

This structure holds the version negotiation state.

This is used by the client to negotiate with the server.

With gQUIC going away, it is probably not very important anymore.

fc_hsk_ctx

Handshake context for the HANDSHAKE stream.

Client and server have different HANDSHAKE stream handlers – and therefore different contexts.

fc_stats

Connection stats

fc_last_stats

Snapshot of connection stats

This is used to log the changes in counters between calls to `ci_log_stats()`. The calculation is straightforward in `lsquic_conn_stats_diff()`.

fc_stream_histories and fc_stream_hist_idx

Rolling log of histories of closed streams

fc_errmsg

Error message associated with connection termination

This is set when the connection is aborted for some reason. This error message is only set once. It is used only to set the error message in the call to `ci_status()`

fc_recent_packets

Dual ring-buffer log of packet history

The first element is for incoming packets, the second is for outgoing packets. Each entry holds received or sent time and frame information.

This can be used for debugging. It is only compiled into debug builds.

fc_stream_ids_to_reset

List of stream ID to send STREAM_RESET for

These STREAM_RESET frames are associated with streams that are not allowed to be created because we sent a GOAWAY frame. (There is a period when GOAWAY is in transit, but the peer keeps on creating streams). To queue the reset frames for such a stream, an element is added to this list.

fc_saved_ack_received

Timestamp of the last received ACK.

This is used for *ACK merging*.

fc_path

The network path – Google QUIC only has one network path.

fc_orig_versions

List (as bitmask) of original versions supplied to the client constructor.

Used for version negotiation. See *fc_ver_neg* for more coverage of this topic.

fc_crypto_enc_level

Latest crypto level

This is for Q050 only, which does away with the HANDSHAKE stream and uses CRYPTO frames instead. (This was part of Google's plan to move Google QUIC protocol closer to IETF QUIC.)

fc_ack

Saved ACK – latest or merged

This ACK structure is used in *ACK merging*.

Instantiation

The largest difference between the server and client mode of the full connection is in the way it is created. The client creates a brand-new connection, performs version negotiation, and runs the handshake before dispatching user events. The server connection, on the other hand, gets created from a mini connection during *connection promotion*. By that time, both version negotiation and handshake have already completed.

Common Initialization

The `new_conn_common()` function contains initialization common to both server and client. Most full connection's internal data structures are initialized or allocated here, among them *Send Controller*, *Receive History*, and *Alarm Set*.

The HEADERS stream is created here, if necessary. (Throughout the code, you can see checks whether the connection is in HTTP mode or not. Even though gQUIC means that HTTP is used, our library supports a non-HTTP mode, in which there is no HEADERS stream. This was done for testing purposes and made possible the echo and md5 client and server programs.)

Server

After initializing the common structures in `new_conn_common()`, server-specific initialization continues in `lsquic_gquic_full_conn_server_new()`.

The HANDSHAKE stream is created. The handler (see `lsquic_server_hsk_stream_if`) simply throws out data that it reads from the client.

Outgoing packets are inherited – they will be sent during the next tick – and deferred incoming packets are processed.

Client

The client's initialization takes place in `lsquic_gquic_full_conn_client_new()`. Crypto session is created and the HANDSHAKE stream is initialized. The handlers in `lsquic_client_hsk_stream_if` drive the handshake process.

Incoming Packets

The entry point for incoming packets is `ci_packet_in()`, which is implemented by `full_conn_ci_packet_in`. Receiving a packet restarts the idle timer.

The function `process_incoming_packet` contains some client-only logic for processing version negotiation and stateless retry packets. In the normal case, `process_regular_packet()` is called. This is where the incoming process is decrypted, the *Receive History* is updated, `parse_regular_packet()` is called, and some post-processing takes place (most importantly, scheduling an ACK to be sent).

The function `parse_regular_packet` is simple: It iterates over the whole decrypted payload of the incoming packet and parses out frames one by one. An error aborts the connection.

ACK Merging

Processing ACKs is *expensive*. When sending data, a batch of incoming packets is likely to contain an ACK frame each. The ACK frame handler, `process_ack_frame()`, merges consecutive ACK frames and stores the result in *fc_ack*. The ACK is processed during the *next tick*. If the two ACK cannot be merged (which is unlikely), the cached ACK is processed immediately and the new ACK is cached.

Caching an ACK has a non-trivial memory cost: the 4KB-plus data structure `ack_info` accounts for more than half of the size of the `full_conn` struct. Nevertheless, the tradeoff is well worth it. ACK merging reduces the number of calls to `lsquic_send_ctl_got_ack()` by a factor of 10 or 20 in some high-throughput scenarios.

Ticking

When a *connection is processed by the engine*, the engine calls the connection's `ci_tick()` method. This is where most of the connection logic is exercised. In the full gQUIC connection, this method is implemented by `full_conn_ci_tick()`.

The following steps are performed:

- A cached ACK, if it exists, is processed
- Expired alarms are rung
- Stream read events are dispatched
- An ACK frame is generated if necessary
- Other control frames are generated if necessary
- Lost packets are rescheduled
- More control frames and stream resets are generated if necessary
- HEADERS stream is flushed
- Outgoing packets that carry stream data are scheduled in four steps:
    a. High-priority *buffered packets* are scheduled
    b. Write events are dispatched for high-priority streams

   c. Non-high-priority buffered packets are scheduled
   d. Write events are dispatched for non-high-priority streams

- Connection close or PING frames are generated if necessary
- Streams are serviced (closed, freed, created)

3.5.14 Full IETF Connection

Files: lsquic_full_conn_ietf.h, lsquic_full_conn_ietf.c

Overview

This module implements IETF QUIC Transport and HTTP/3 logic, plus several QUIC extensions. To attain an overall grasp of the code, at least some familiarity with these protocols is required. To understand the code in detail, especially *why* some things are done, a closer reading of the specification may be in order.

In some places, the code contains comments with references to the specification, e.g.

```
if (conn->ifc_flags & IFC_SERVER)
{   /* [draft-ietf-quic-transport-34] Section 19.7 */
    ABORT_QUIETLY(0, TEC_PROTOCOL_VIOLATION,
                        "received unexpected NEW_TOKEN frame");
    return 0;
}
```

(A search for "[draft-ietf" will reveal over one hundred places in the code thus commented.)

The Full IETF Connection module is similar in structure to the *Full gQUIC Connection* module, from which it originated. Some code is quite similar as well, including logic for *ACK Merging* and *Ticking*.

Components

In this section, each member of `ietf_full_conn` is documented.

ifc_conn

The first member of the struct is the common connection object, *lsquic_conn*.

It must be first in the struct because the two pointer are cast to each other, depending on circumstances.

ifc_cces

This array holds eight connection CID elements. See *Managing SCIDs*.

ifc_rechist

This member holds the *packet receive history*. The receive history is used to generate ACK frames.

ifc_max_ackable_packno_in

This value is used to detect holes in incoming packet number sequence. This information is used to queue ACK frames.

ifc_send_ctl

This is the *Send Controller*. It is used to allocate outgoing packets, control sending rate, and process acknowledgements.

ifc_pub

This member holds the *Connection Public Interface*

ifc_alset

This is the *Alarm Set*. It is used to set various timers in the connection and the send controller.

ifc_closed_stream_ids

The two sets in this array hold the IDs of closed streams.

There are two of them because of the uneven distribution of stream IDs. The set data structure is meant to hold sequences without gaps. It is more efficient to hold stream IDs for each stream type in separate structures.

ifc_n_created_streams

Counters for locally initiated streams. Used to generate next stream ID.

ifc_max_allowed_stream_id

Maximum allowed stream ID for each of the four (`N_SITS`) stream types. This is used all over the place.

ifc_closed_peer_streams

Counts how many remotely-initiated streams have been closed. Because the protocol mandates that the stream IDs be assigned in order, this allows us to make some logical inferences in the code.

ifc_max_streams_in

Maximum number of open streams the peer is allowed to initiate.

ifc_max_stream_data_uni

Initial value of the maximum amount of data locally-initiated unidirectional stream is allowed to send.

ifc_flags

All kinds of flags.

ifc_mflags

More flags!

ifc_send_flags

The send flags keep track of which control frames are queued to be sent.

ifc_delayed_send

Some send flags are delayed.

We stop issuing streams credits if peer stops opening QPACK decoder window. This addresses a potential attack whereby client can cause the server to keep allocating memory. See Security Considerations in the QPACK Internet-Draft.

ifc_send

This is the *Send Controller*. It is used to allocate outgoing packets, control sending rate, and process acknowledgements.

ifc_error

This struct records which type of error has occurred (transport or application)' and the error code.

ifc_n_delayed_streams

Count how many streams have been delayed.

When `lsquic_conn_make_stream()` is called, a stream may not be created immediately. It is delayed if creating a stream would go over the maximum number of stream allowed by peer.

ifc_n_cons_unretx

Counts how many consecutive unretransmittable packets have been sent.

Enough unretransittable sent packets in a row causes a PING frame to be sent. This forces the peer to send an ACK.

ifc_pii

Points to the selected priority iterator.

The IETF Full Connection supports two priority mechanisms: the original Google QUIC priority mechanism and the HTTP/3 Extensible Priorities.

ifc_errmsg

Holds dynamically generated error message string.

Once set, the error string does not change until the connection is destroyed.

ifc_enpub

This points to the *engine's public interface*.

ifc_settings

Pointer to the engine settings.

This member is superfluous – the settings can be fetched from `ifc_enpub->enp_settings`.

ifc_stream_ids_to_ss

Holds a queue of STOP_SENDING frames to send as response to remotely initiated streams that came in after we sent a GOAWAY frame.

ifc_created

Time when the connection was created. This is used for the Timestamp and Delayed ACKs extensions.

ifc_saved_ack_received

Time when cached ACK frame was received. See *ACK Merging*.

ifc_max_ack_packno

Holding the maximum packet number containing an ACK frame allows us to ignore old ACK frames. One value per Packet Number Space is kept.

ifc_max_non_probing

Maximum packet number of a received non-probing packets. This is used for path migration.

ifc_cfg

Local copy of a couple of transport parameters. We could get at them with a function call, but these are used often enough to optimize fetching them.

ifc_process_incoming_packet

The client goes through version negotiation and the switches to the fast function. The server begins to use the fast function immediately.

ifc_n_slack_akbl

Number ackable packets received since last ACK was sent. A count is kept for each Packet Number Space.

ifc_n_slack_all

Count of all packets received since last ACK was sent. This is only used in the Application PNS (Packet Number Space). (This is regular PNS after the handshake completes).

ifc_max_retx_since_last_ack

This number is the maximum number of ack-eliciting packets to receive before an ACK must be sent.

The default value is 2. When the Delayed ACKs extension is used, this value gets modified by peer's ACK_FREQUENCY frames.

ifc_max_ack_delay

Maximum amount of allowed after before an ACK is sent if the threshold defined by *ifc_max_retx_since_last_ack* has not yet been reached.

The default value is 25 ms. When the Delayed ACKs extension is used, this value gets modified by peer's ACK_FREQUENCY frames.

ifc_ecn_counts_in

Incoming ECN counts in each of the Packet Number Spaces. These counts are used to generate ACK frames.

ifc_max_req_id

Keeps track of the maximum ID of bidirectional stream ID initiated by the peers. It is used to construct the GOAWAY frame.

ifc_hcso

State for outgoing HTTP/3 control stream.

ifc_hcsi

State for incoming HTTP/3 control stream.

ifc_qeh

QPACK encoder streams handler.

The handler owns two unidirectional streams: a) locally-initiated QPACK encoder stream, to which it writes; and b) peer-initiated QPACK decoder stream, from which it reads.

ifc_qdh

QPACK decoder streams handler.

The handler owns two unidirectional streams: a) peer-initiated QPACK encoder stream, from which it reads; and b) locally-initiated QPACK decoder stream, to which it writes.

ifc_peer_hq_settings

Peer's HTTP/3 settings.

ifc_dces

List of destination connection ID elements (DCEs). Each holds a DCID and the associated stateless reset token. When lsquic uses a DCID, it inserts the stateless reset token into a hash so that stateless resets can be found.

Outside of the initial migration, the lsquic client code does not switch DCIDs. One idea (suggested in the drafts somewhere) is to switch DCIDs after a period of inactivity.

ifc_to_retire

List of DCIDs to retire.

ifc_scid_seqno

Sequence generator for SCIDs generated by the endpoint.

ifc_scid_timestamp

List of timestamps for the generated SCIDs.

This list is used in the SCID rate-limiting mechanism.

ifc_incoming_ecn

History indicating presence of ECN markings on most recent incoming packets.

ifc_cur_path_id

Current path ID – indexes *ifc_paths*.

ifc_used_paths

Bitmask of which paths in *ifc_paths* are being used.

ifc_mig_path_id

Path ID of the path being migrated to.

ifc_active_cids_limit

This is the maximum number of CIDs at any one time this endpoint is allowed to issue to peer. If the TP value exceeds `cn_n_cces`, it is reduced to it.

ifc_active_cids_count

This value tracks how many CIDs have been issued. It is decremented each time a CID is retired.

ifc_first_active_cid_seqno

Another piece of the SCID rate-limiting mechanism.

ifc_ping_unretx_thresh

Once the number consecutively sent non-ack-elicing packets (*ifc_n_cons_unretx*) exceeds this value, this endpoint will send a PING frame to force the peer to respond with an ACK.

The threshold begins at 20 and then made to fluctuate randomly between 12 and 27.

ifc_last_retire_prior_to

Records the maximum value of `Retire Prior To` value of the NEW_CONNECTION_ID frame.

ifc_ack_freq_seqno

Sequence number generator for ACK_FREQUENCY frames generated by this endpoint.

ifc_last_pack_tol

Last value of the `Packet Tolerance` field sent in the last `ACK_FREQUENCY` frame generated by this endpoint.

ifc_last_calc_pack_tol

Last *calculated* value of the `Packet Tolerance` field.

ifc_min_pack_tol_sent

Minimum value of the `Packet Tolerance` field sent. Only used for statistics display.

ifc_max_pack_tol_sent

Maximum value of the `Packet Tolerance` field sent. Only used for statistics display.

ifc_max_ack_freq_seqno

Maximum seen sequence number of incoming `ACK_FREQUENCY` frame. Used to discard old frames.

ifc_max_udp_payload

Maximum UDP payload. This is the cached value of the transport parameter.

ifc_last_live_update

Last time `ea_live_scids()` was called.

ifc_paths

Array of connection paths. Most of the time, only one path is used; more are used during *migration*. The array has four elements as a safe upper limit.

The elements are of type `struct conn_path`. Besides the network path, which stores socket addresses and is associated with each outgoing packet (via `po_path`), the connection path keeps track of the following information:

- Outgoing path challenges. See *Sending Path Challenges*.
- Incoming path challenge.
- Spin bit (`cop_max_packno`, `cop_spin_bit`, and `COP_SPIN_BIT`).
- DPLPMTUD state.

ifc_u.cli

Client-specific state. This is where pointers to "crypto streams" are stored; they are not in the `ifc_pub.all_streams` hash.

ifc_u.ser

The server-specific state is only about push promises.

ifc_idle_to

Idle timeout.

ifc_ping_period

Ping period.

ifc_bpus

A hash of buffered priority updates. It is used when a priority update (part of the Extensible HTTP Priorities extension) arrives before the stream it is prioritizing.

ifc_last_max_data_off_sent

Value of the last MAX_DATA frame sent. This is used to limit the number of times we send the MAX_DATA frame in response to a DATA_BLOCKED frame.

ifc_min_dg_sz

Minimum size of the DATAGRAM frame. Used by the eponymous extension.

ifc_max_dg_sz

Maximum size of the DATAGRAM frame. Used by the eponymous extension.

ifc_pts

PTS stands for "Packet Tolerance Stats". Information collected here is used to calculate updates to the packet tolerance advertised to the peer via ACK_FREQUENCY frames. Part of the Delayed ACKs extension.

ifc_stats

Cumulative connection stats.

ifc_last_stats

Copy of *ifc_stats* last time `ci_log_stats()` was called. Used to calculate the difference.

ifc_ack

One or more cached incoming ACK frames. Used for *ACK merging*.

Managing SCIDs

Source Connection IDs – or SCIDs for short – are stored in the *ifc_cces* array.

Each of `struct conn_cid_elem` contains the CID itself, the CID's port or sequence number, and flags:

- CCE_USED means that this Connection ID has been used by the peer. This information is used to check whether the peer's incoming packet is using a new DCID or reusing an old one when the packet's DCID does not match this path's current DCID.
- CCE_REG signifies that the CID has been registered with the user-defined ea_new_scids() callback.
- CCE_SEQNO means that the connection has been issued by this endpoint and cce_seqno contains a valid value. Most of SCIDs are issued by either endpoint, with one exception: The DCID included in the first few packets sent by the client becomes an interim SCID for the server and it does not have a sequence number. This "original" SCID gets retired 2 seconds after the handshake succeeds, see the AL_RET_CIDS alarm.
- CCE_PORT is used to mark the special case of hashing connections by port number. In client mode, the lsquic engine may, under some circumstances, hash the connections by local port number instead of connection ID. In that case, cce_port contains the port number used to hash the connection.

Each CIDs is hashed in the of the "CID-to-connection" mapping that the engine maintains. If it is not in the hash, incoming packets that use this CID as DCID will not be dispatched to the connection (because the connection will not be found).

Path Migration

What follows assumes familiarity with Section 9 of the Transport I-D.

Server

The server handles two types of path migration. In the first type, the client performs probing by sending path challenges; in the second type, the migration is due to a NAT rebinding.

The connection keeps track of different paths in *ifc_paths*. Path objects are allocated out of the ifc_paths array. They are of type struct conn_path; one of the members is cop_path, which is the network path object used to send packets (via po_path).

Each incoming packet is fed to the engine using the lsquic_engine_packet_in() function. Along with the UDP datagram, the local and peer socket addresses are passed to it. These addresses are eventually passed to the connection via the ci_record_addrs() call. The first of these calls – for the first incoming packet – determines the *current path*. When the address pair, which is a four-tuple of local and remote IP addresses and port numbers, does not match that of the current path, a new path object is created, triggering migration logic.

ci_record_addrs() returns a *path ID*, which is simply the index of the corresponding element in the ifc_paths array. The current path ID is stored in ifc_cur_path_id. The engine assigns this value to the newly created incoming packet (in pi_path_id). The packet is then passed to ci_packet_in().

The first part of the path-switching logic is in process_regular_packet():

```
case REC_ST_OK:
    /* --- 8< --- some code elided... */
    saved_path_id = conn->ifc_cur_path_id;
    parse_regular_packet(conn, packet_in);
    if (saved_path_id == conn->ifc_cur_path_id)
    {
        if (conn->ifc_cur_path_id != packet_in->pi_path_id)
        {
            if (0 != on_new_or_unconfirmed_path(conn, packet_in))
            {
                LSQ_DEBUG("path %hhu invalid, cancel any path response "
                    "on it", packet_in->pi_path_id);
                conn->ifc_send_flags &= ~(SF_SEND_PATH_RESP
```

(continues on next page)

(continued from previous page)

```
                                                    << packet_in->pi_path_id);
            }
```

The above means: if the current path has not changed after the packet was processed, but the packet came in on a different path, then invoke the "on new or unconfirmed path" logic. This is done this way because the current path may be have been already changed if the packet contained a PATH_RESPONSE frame.

First time a packet is received on a new path, a PATH_CHALLENGE frame is scheduled.

If more than one packet received on the new path contain non-probing frames, the current path is switched: it is assumed that the path change is due to NAT rebinding.

Client

Path migration is controlled by the client. When the client receives a packet from an unknown server address, it drops the packet on the floor (per spec). This code is in `process_regular_packet()`.

The client can migrate if `es_allow_migration` is on (it is in the default configuration) and the server provides the "preferred_address" transport parameter. The migration process begins once the handshake is confirmed; see the `maybe_start_migration()` function. The SCID provided by the server as part of the "preferred_address" transport parameter is used as the destination CID and path #1 is picked:

```
copath = &conn->ifc_paths[1];
migra_begin(conn, copath, dce, (struct sockaddr *) &sockaddr, params);
return BM_MIGRATING;
```

In `migra_begin`, migration state is initiated and sending of a PATH_CHALLENGE frame is scheduled:

```
conn->ifc_mig_path_id = copath - conn->ifc_paths;
conn->ifc_used_paths |= 1 << conn->ifc_mig_path_id;
conn->ifc_send_flags |= SF_SEND_PATH_CHAL << conn->ifc_mig_path_id;
LSQ_DEBUG("Schedule migration to path %hhu: will send PATH_CHALLENGE",
    conn->ifc_mig_path_id);
```

Sending Path Challenges

To send a path challenge, a packet is allocated to be sent on that path, a new challenge is generated, the PATH_CHALLENGE is written to the packet, and the packet is scheduled. All this happens in the `generate_path_chal_frame()` function.

```
need = conn->ifc_conn.cn_pf->pf_path_chal_frame_size();
packet_out = get_writeable_packet_on_path(conn, need, &copath->cop_path, 1);
/* --- 8< --- some code elided... */
w = conn->ifc_conn.cn_pf->pf_gen_path_chal_frame(
        packet_out->po_data + packet_out->po_data_sz,
        lsquic_packet_out_avail(packet_out),
        copath->cop_path_chals[copath->cop_n_chals]);
/* --- 8< --- some code elided... */
lsquic_alarmset_set(&conn->ifc_alset, AL_PATH_CHAL + path_id,
                now + (INITIAL_CHAL_TIMEOUT << (copath->cop_n_chals - 1)));
```

If the path response is not received before a timeout, another path challenge is sent, up to the number of elements in `cop_path_chals`. The timeout uses exponential back-off; it is not based on RTT, because the RTT of the new path is unknown.

Receiving Path Responses

When a PATH_RESPONSE frame is received, the path on which the corresponding challenge was sent may become the new current path. See `process_path_response_frame()`.

Note that the path ID of the incoming packet with the PATH_RESPONSE frame is not taken into account. This is by design: see Section 8.2.2 of the Transport I-D.

Stream Priority Iterators

Creating Streams on the Server

Calculating Packet Tolerance

When the Delayed ACKs extension is used, we advertise our `Packet Tolerance` to peer. This is the number of packets the peer can receive before having to send an acknowledgement. By default – without the extension – the packet tolerance is 2.

Because we *merge ACKs*, receiving more than one ACK between ticks is wasteful. Another consideration is that a packet is not declared lost until at least one RTT passes – the time to send a packet and receive the acknowledgement from peer.

To calculate the packet tolerance, we use a feedback mechanism: when number of ACKs per RTT is too high, we increase packet tolerance; when number of ACKs per RTT is too low, we decrease packet tolerance. The feedback is implemented with a PID Controller: the target is the number of ACKs per RTT, normalized to 1.0.

See the function `packet_tolerance_alarm_expired()` as well as comments in `lsquic.h` that explain the normalization as well as the knobs available for tuning.

The pre-normalized target is a function of RTT. It was obtained empirically using netem. This function together with the default PID controller parameters give good performance in the lab and in some limited interop testing.

3.5.15 Anatomy of Outgoing Packet

Overview

The outgoing packet is represented by `struct lsquic_packet_out`. An outgoing packet always lives on one – and only one – of the *Send Controller*'s *Packet Queues*. For that, `po_next` is used.

Beyond the packet number, stored in `po_packno`, the packet has several properties: sent time (`po_sent`), frame information, encryption level, network path, and others. Several properties are encoded into one or more bits in the bitmasks `po_flags` and `po_lflags`. Multibit properties are usually accessed and modified by a special macro.

The packet has a pointer to the packetized data in `po_data`. If the packet has been encrypted but not yet sent, the encrypted buffer is pointed to `po_enc_data`.

Packet Payload

The payload consists of the various frames – STREAM, ACK, and others – written, one after another, to `po_data`. The header, consisting of the type byte, (optional) connection ID, and the packet number is constructed when the packet is just about to be sent, during encryption. This buffer – header and the encrypted payload are stored in a buffer pointed to by `po_enc_data`.

Because stream data is written directly to the outgoing packet, the packet is not destroyed when it is declared lost by the *loss detection logic*. Instead, it is repackaged and sent out again as a new packet. Besides assigning the packet a new number, packet retransmission involves removing non-retransmittable frames from the packet. (See `lsquic_packet_out_chop_regen()`.)

Historically, some places in the code assumed that the frames to be dropped are always located at the beginning of the `po_data` buffer. (This was before a *frame record* was created for each frame). The cumulative size of the frames to be removed is in `po_regen_sz`; this size can be zero. Code that generates non-retransmittable frames still writes them only to the beginning of the packet.

The goal is to drop `po_regen_sz` and to begin to write ACK and other non-retransmittable frames anywhere. This should be possible to do now (see `lsquic_packet_out_chop_regen()`, which can support such use after removing the assertion), but we haven't pulled the trigger on it yet. Making this change will allow other code to become simpler: for example, the opportunistic ACKs logic.

Frame Records

Each frame written to `po_data` has an associated *frame record* stored in `po_frecs`:

```
struct frame_rec {
    union {
        struct lsquic_stream   *stream;
        uintptr_t               data;
    }                       fe_u;
    unsigned short          fe_off,
                            fe_len;
    enum quic_frame_type    fe_frame_type;
};
```

Frame records are primarily used to keep track of the number of unacknowledged stream frames for a stream. When a packet is acknowledged, the frame records are iterated over and `lsquic_stream_acked()` is called. The second purpose is to speed up packet resizing, as frame records record the type, position, and size of a frame.

Most of the time, a packet will contain a single frame: STREAM on the sender of data and ACK on the receiver. This use case is optimized: `po_frecs` is a union and when there is only one frame per packets, the frame record is stored in the packet struct directly.

3.5.16 Evanescent Connection

Files: lsquic_pr_queue.h, lsquic_pr_queue.c

"PR Queue" stands for "Packet Request Queue." This and the Evanescent Connection object types are explaned below in this section.

Overview

Some packets need to be replied to outside of context of existing mini or full connections:

1. A version negotiation packet needs to be sent when a packet arrives that specifies QUIC version that we do not support.
2. A stateless reset packet needs to be sent when we receive a packet that does not belong to a known QUIC connection.

The replies cannot be sent immediately. They share outgoing socket with existing connections and must be scheduled according to prioritization rules.

The information needed to generate reply packet – connection ID, connection context, and the peer address – is saved in the Packet Request Queue.

When it is time to send packets, the connection iterator knows to call prq_next_conn() when appropriate. What is returned is an evanescent connection object that disappears as soon as the reply packet is successfully sent out.

There are two limits associated with Packet Request Queue:

1. Maximum number of packet requests that are allowed to be pending at any one time. This is simply to prevent memory blowout.
2. Maximum verneg connection objects to be allocated at any one time. This number is the same as the maximum batch size in the engine, because the packet (and, therefore, the connection) is returned to the Packet Request Queue when it could not be sent.

We call this a "request" queue because it describes what we do with QUIC packets whose version we do not support or those packets that do not belong to an existing connection: we send a reply for each of these packets, which effectively makes them "requests."

Packet Requests

When an incoming packet requires a non-connection response, it is added to the Packet Request Queue. There is a single `struct pr_queue` per engine – it is instantiated if the engine is in the server mode.

The packet request is recorded in `struct packet_req`, which are kept inside a hash in the PR Queue. The reason for keeping the requests in a hash is to minimize duplicate responses: If a client hello message is spread over several incoming packets, only one response carrying the version negotiation packet (for example) will be sent.

```
struct packet_req
{
    struct lsquic_hash_elem     pr_hash_el;
    lsquic_cid_t                pr_scid;
    lsquic_cid_t                pr_dcid;
    enum packet_req_type        pr_type;
    enum pr_flags {
        PR_GQUIC    = 1 << 0,
    }                           pr_flags;
    enum lsquic_version         pr_version;
    unsigned                    pr_rst_sz;
    struct network_path         pr_path;
};
```

Responses are created on demand. Until that time, everything that is necessary to generate the response is stored in `packet_req`.

Sending Responses

To make these packets fit into the usual packet-sending loop, each response is made to resemble a packet sent by a connecteion. For that, the PR Queue creates a connection object that only lives for the duration of batching of the packet. (Hence the connection's name: *evanescent* connection.) This connection is returned by the `lsquic_prq_next_conn()` by the connection iterator during the *batching process*

For simplicity, the response packet is generated in this function as well. The call to `ci_next_packet_to_send()` only returns the pointer to it.

3.5.17 Send Controller

Files: lsquic_send_ctl.h, lsquic_send_ctl.c

Overview

The Send Controller manages outgoing packets and the sending rate:

- It decides whether packets can be sent
- It figures out what the congestion window is
- It processes acknowledgements and detects packet losses
- It allocates packets
- It maintains sent packet history

The controller allocates, manages, splits, coalesces, and destroys outgoing packets. It owns these packets.

The send controller services two modules:

- Full connection. gQUIC and IETF full connections use the send controller to allocate packets and delegate packet-sending decisions to it.
- Stream. The stream uses the stream controller as the source of outgoing packets to write STREAM frames to.

Packet Life Cycle

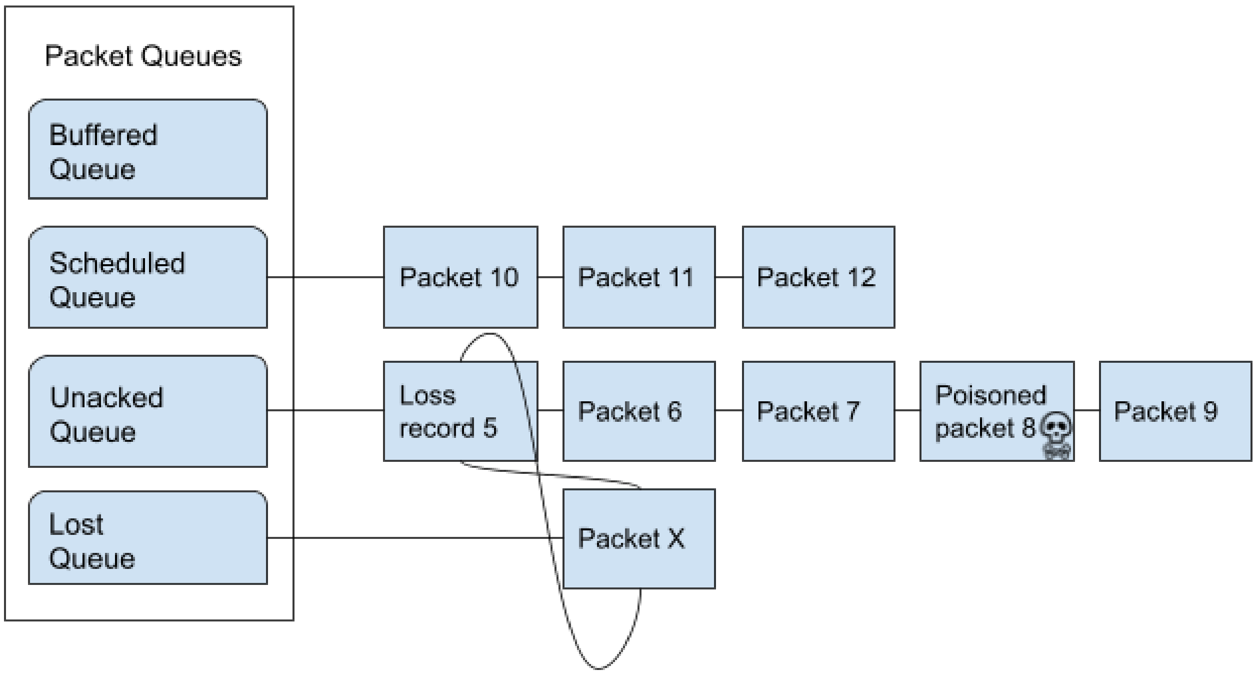
A new outgoing packet is allocated and returned to the connection or the stream. Around this time (just before or just after, depending on the particular function call to get the packet), the packet is placed on the Scheduled Queue.

When the engine is creating a batch of packets to send, it calls `ci_next_packet_to_send()`. The connection removes the next packet from its Scheduled Queue and returns it. The engine now owns the outgoing packet, but only while the batch is being sent. The engine *always returns the packet* after it tries to send it.

If the packet was sent successfully, it is returned via the `ci_packet_sent` call, after which it is appended to the Unacked Queue. If the packet could not be sent, `ci_packet_not_sent()` is called, at which point it is prepended back to the Schedule Queue to be tried later.

There are two ways to get off the Unacked Queue: being acknowledged or being lost. When a packet is acknowledged, it is destroyed. On the other hand, if it is deemed lost, it is placed onto the Lost Queue, where it will await being rescheduled.

Packet Queues



Buffered Queue

The Buffered Queue is a special case. When writing to the stream occurs outside of the event dispatch loop, new outgoing packets begin their life in the Buffered Queue. They get scheduled during a connection tick, making their way onto the Scheduled Queue.

There are two buffered queues: one for packets holding STREAM frames from the highest-priority streams and one for packets for streams with lower priority.

Scheduled Queue

Packets on the Scheduled Queue have packet numbers assigned to them. In rare cases, packets may be removed from this queue before being sent out. (For example, a stream may be cancelled, in which case packets that carry its STREAM frames may end up empty.) In that case, they are marked with a special flag to generate the packet number just before they are sent.

Unacked Queue

This queue holds packets that have been sent but are yet to be acknowledged. When a packet on this queue is acknowledged, it is destroyed.

The loss detection code runs on this queue when ACKs are received or when the retransmission timer expires.

This queue is actually three queues: one for each of the IETF QUIC's Packet Number Spaces, or PNSs. The PNS_APP queue is what is used by gQUIC and IETF QUIC server code. PNS_INIT and PNS_HSK are only used by the IETF QUIC client. (IETF QUIC server handles those packet number spaces in its mini conn module.)

In addition to regular packets, the Unacked Queue holds *loss records* and *poisoned packets*.

Lost Queue

This queue holds lost packets. These packets are removed from the Unacked Queue when it is decided that they have been lost. Packets on this queue get rescheduled after connection schedules a packet with control frames, as those have higher priority.

0-RTT Stash Queue

This queue is used by the client to retransmit packets that carry 0-RTT data.

Handling ACKs

Acknowledgements are processed in the function `lsquic_send_ctl_got_ack`.

One of the first things that is done is ACK validation. We confirm that the ACK does not contain any packet numbers that we did not send. Because of the way we *generate packet numbers*, this check is a simple comparison.

The nested loops work as follows. The outer loop iterates over the packets in the Unacked Queue in order – meaning packet numbers increase. In other words, older packets are examined first. The inner loop processes ACK ranges in the ACK *backwards*, meaning that both loops follow packets in increasing packet number order. It is done this way as an optimization. The (previous) alternative design of looking up a packet number in the ACK frame, even if using binary search, is slower.

The code is optimized: the inner loop has a minimum possible number of branches. These optimizations predate the more-recent, higher-level optimization. The latest ACK-handling optimization added to the code combines incoming ACKs into a single ACK (at the connection level), thus reducing the number of times this loop needs to be used by a lot, sometimes by a significant factor (when lots of data is being sent). This makes some of the code-level optimizations, such as the use of `__builtin_prefetch`, an overkill.

Loss Records

A loss record is a special type of outgoing packet. It marks a place in the Unacked Queue where a lost packet had been – the lost packet itself having since moved on to the Lost Queue or further. The loss record and the lost packet form a circular linked list called the "loss chain." This list contains one real packet and zero or more loss records. The real packet can move from the Unacked Queue to the Lost Queue to the Scheduled Queue and back to the Unacked Queue; its loss records live only on the Unacked Queue.

We need loss records to be able to handle late acknowledgements – those that acknowledge a packet *after* it has been deemed lost. When an acknowledgment for any of the packet numbers associated with this packet comes in, the packet is acknowledged and the whole loss chain is destroyed.

Poisoned Packets

A poisoned packet is used to thwart opportunistic ACK attacks. The opportunistic ACK attack works as follows:

- The client requests a large resource
- The server begins sending the response
- The client sends ACKs for packet number before it sees these packets, tricking the server into sending packets faster than it would otherwise

The poisoned packet is placed onto the Unacked Queue. If the peer lies about packet numbers it received, it will acknowledge the poisoned packet, in which case it will be discovered during ACK processing.

Poisoned packets cycle in and out of the Unacked Queue. A maximum of one poisoned packet is outstanding at any one time for simplicity. (And we don't need more).

Packet Numbers

The Send Controller aims to send out packets without any gaps in the packet number sequence. (The only exception to this rule is the handling of poisoned packets, where the gap is what we want.) Not having gaps in the packet number sequence is beneficial:

- ACK verification is cheap
- Send history updates are fast
- Send history uses very little memory

The downside is code complexity and having to renumber packets when they are removed from the Scheduled Queue (due to, for example, STREAM frame elision or loss chain destruction) or resized (due to a path or MTU change, for instance).

Some scenarios when gaps can be produced inadvertently are difficult to test or foresee. To cope with that, a special warning in the send history code is added when the next packet produces a gap. This warning is limited to once per connection. Having a gap does not break functionality other than ACK verification, but that's minor. On the other hand, we want to fix such bugs when they crop up – that's why the warning is there.

Loss Detection and Retransmission

The loss detection and retransmission logic in the Send Controller was taken from the Chromium code in the fall of 2016, in the beginning of the lsquic project. This logic has not changed much since then – only some bugs have been fixed here and there. The code bears no resemblance to what is described in the QUIC Recovery Internet Draft. Instead, the much earlier document, describing gQUIC, could be looked to for reference.

Congestions Controllers

The Send Controller has a choice of two congestion controllers: Cubic and BBRv1. The latter was translated from Chromium into C. BBRv1 does not work well for very small RTTs.

To cope with that, lsquic puts the Send Controller into the "adaptive CC" mode by default. The CC is selected after RTT is determined: below a certain threshold (configurable; 1.5 ms by default), Cubic is used. Until Cubic or BBRv1 is selected, *both* CC controllers are used – because we won't have the necessary state to instantiate a controller when the decision is made.

Buffered Packet Handling

Buffered packets require quite a bit of special handling. Because they are created outside of the regular event dispatch, a lot of things are unknown:

- Congestion window
- Whether more incoming packets will arrive before the next tick
- The optimal packet number size

The Send Controller tries its best to accommodate the buffered packets usage scenario.

ACKs

When buffered packets are created, we want to generate an ACK, if possible. This can be seen in `send_ctl_get_buffered_packet`, which calls `ci_write_ack()`

This ACK should be in the first buffered packet to be scheduled. Because the Send Controller does not dictate the order of buffered packet creation – high-priority versus low-priority – it may need to move (or steal) the ACK frame from a packet on the low-priority queue to a packet on the high-priority queue.

When buffered packets are finally scheduled, we have to remove ACKs from them if another ACK has already been sent. This is because Chrome errors out if out-of-order ACKs come in.

Flushing QPACK Decoder

The priority-based write events dispatch is emulated when the first buffered packet is allocated: the QPACK decoder is flushed. Without it, QPACK updates are delayed, which may negatively affect compression ratio.

Snapshot and Rollback

The Send Controller snapshot and rollback functionality was implemented exclusively for the benefit of the optimized `lsquic_stream_pwritev` call.

Complexity Woes

The Send Controller is complicated. Because we write stream data to packets directly and packets need to be resized, a lot of complexity resides in the code to resize packets, be it due to repathing, STREAM frame elision, or MTU changes. This is the price to be paid for efficiency in the normal case.

3.5.18 Alarm Set

Files: lsquic_alarmset.h, lsquic_alarmset.c, test_alarmset.c

The alarm set, `struct lsquic_alarmset`, is an array of callbacks and expiry times. To speed up operations, setting and unsetting alarms is done via macros.

The functions to ring[4] the alarms and to calculate the next alarm time use a loop. It would be possible to maintain a different data structure, such as a min-heap, to keep the alarm, and that would obviate the need to loop in `lsquic_alarmset_mintime()`. It is not worth it: the function is not called often and a speed win here would be offset by the necessity to maintain the min-heap ordering.

3.5.19 Tickable Queue

Files: lsquic_engine.c, lsquic_min_heap.h, lsquic_min_heap.c

The Tickable Queue is a min-heap used as a priority queue. Connections on this queue are in line to be processed. Connections that were last ticked a longer time ago have higher priority than those ticked recently. (`cn_last_ticked` is used for ordering.) This is intended to prevent starvation as multiple connections vye for the ability to send packets.

The way the min-heap grows is described in *Growing Min-Heaps*.

[4] This term was picked consciously: alarms *ring*, while timers do other things, such as "fire" and so on.

3.5.20 Advisory Tick Time Queue

Files: lsquic_attq.h, lsquic_attq.c

This data structure is a mini-heap. Connections are ordered by the value of the next time they should be processed (ticked). (Because this is not a hard limit, this value is advisory – hence its name.)

This particular min-heap implementation has two properties worth highlighting:

Removal of Arbitrary Elements

When a connection's next tick time is updated (or when the connection is destroyed), the connection is removed from the ATTQ. At that time, it may be at any position in the min-heap. The position is recorded in the heap element, `attq_elem->ae_heap_idx` and is updated when elements are swapped. This makes it unnecessary to search for the entry in the min-heap.

Swapping Speed

To make swapping faster, the array that underlies the min-heap is an array of *pointers* to `attq_elem`. This makes it unnecessary to update each connection's `cn_attq_elem` as array elements are swapped: the memory that stores `attq_elem` stays put. This is why there are both `aq_elem_malo` and `aq_heap`.

3.5.21 CID Purgatory

Files: lsquic_purga.h, lsquic_purga.c

Overview

This module keeps a set of CIDs that should be ignored for a period of time. It is used when a connection is closed: this way, late packets will not create a new connection.

A connection may have been deleted, retired, or closed. In the latter case, it enters the Draining State. In this state, the connection is to ignore incoming packets.

Structure

The purgatory keeps a list of 16-KB pages. A page looks like this:

```
#define PURGA_ELS_PER_PAGE 273

struct purga_page
{
    TAILQ_ENTRY(purga_page)     pupa_next;
    lsquic_time_t               pupa_last;
    unsigned                    pupa_count;
    bloom_mask_el_t             pupa_mask[BLOOM_N_MASK_ELS];
    lsquic_cid_t                pupa_cids[PURGA_ELS_PER_PAGE];
    void *                      pupa_peer_ctx[PURGA_ELS_PER_PAGE];
    struct purga_el             pupa_els[PURGA_ELS_PER_PAGE];
};
```

The reason for having CIDs and peer contexts in separate arrays is to be able to call the `ea_old_scids()` callback when a page expires. A page is expired when it is full and the last added element is more than `pur_min_life` microseconds ago. The minimum CID life is hardcoded as 30 seconds in lsquic_engine.c (see the `lsquic_purga_new()` call).

To avoid scannig the whole array of CIDs in `lsquic_purga_contains()`, we use a Bloom filter.

The Bloom filter is constructed using a 8192-bit bit field and 6 hash functions. With 273 elements per page, this gives us 0.004% possibility of a false positive. In other words, when we do have to search a page for a particular CID, the chance of finding the CID is 99.99%.

Quick calculation:

```
perl -E '$k=6;$m=1<<13;$n=273;printf("%f\n", (1-exp(1)**-($k*$n/$m))**$k)'
```

To extract 6 13-bit values from a 64-bit integer, they are overlapped:

```
 0         10        20        30        40        50        60
+----------------------------------------------------------------+
|                                                                |
+----------------------------------------------------------------+
 1111111111111
           2222222222222
                     3333333333333
                               4444444444444
                                         5555555555555
                                                   6666666666666
```

This is not 100% kosher, but having 6 functions gives a better guarantee and it happens to work in practice.

3.5.22 Memory Manager

Files: lsquic_mm.h, lsquic_mm.c

The memory manager allocates several types of objects that are used by different parts of the library:

- Incoming packet objects and associated buffers
- Outgoing packet objects and associated buffers
- Stream frames
- Frame records
- Mini connections, both Google and IETF QUIC
- DCID elements
- HTTP/3 (a.k.a. "HQ") frames
- Four- and sixteen-kilobyte pages

These objects are either stored on linked list or in *malo* pools and are shared among all connections. (Full connections allocate outgoing packets from per-connection malo allocators: this is done to speed up *ACK processing*.)

The list of cached outgoing packet buffers is shrunk once in a while (see the "poolst_*" functions). Other object types are kept in the cache until the engine is destroyed. One Memory Manager object is allocated per engine instance.

3.5.23 Malo Allocator

Files: lsquic_malo.h, lsquic_malo.c

Overview

The malo allocator is a pool of objects of fixed size. It tries to allocate and deallocate objects as fast as possible. To do so, it does the following:

1. Allocations occur 4 KB at a time.
2. No division or multiplication operations are performed for appropriately sized objects. (More on this below.)

(In recent testing, malo was about 2.7 times faster than malloc for 64-byte objects.)

Besides speed, the allocator provides a convenient API: To free (put) an object, one does not need a pointer to the malo object.

To gain all these advantages, there are trade-offs:

1. There are two memory penalties:
    a. Per object overhead. If an object is at least ROUNDUP_THRESH in size as the next power of two, the allocator uses that power of two value as the object size. This is done to avoid using division and multiplication. For example, a 104-byte object will have a 24-byte overhead.
    b. Per page overhead. Page links occupy some bytes in the page. To keep things fast, at least one slot per page is always occupied, independent of object size. Thus, for a 1 KB object size, 25% of the page is used for the page header.
2. 4 KB pages are not freed until the malo allocator is destroyed. This is something to keep in mind.

Internal Structure

The malo allocator allocates objects out of 4 KB pages. Each page is aligned on a 4-KB memory boundary. This makes it possible for the `lsquic_malo_put()` function only to take on argument – the object to free – and to find the malo allocator object itself.

Each page begins with a header followed by a number of slots – up to the 4-KB limit. Two lists of pages are maintained: all pages and free pages. A “free” page is a page with at least one free slot in it.

The malo allocator (`struct malo`) stores itself in the first page, occupying some slots.

3.5.24 Receive History

Files: lsquic_rechist.h, lsquic_rechist.c, test_rechist.c

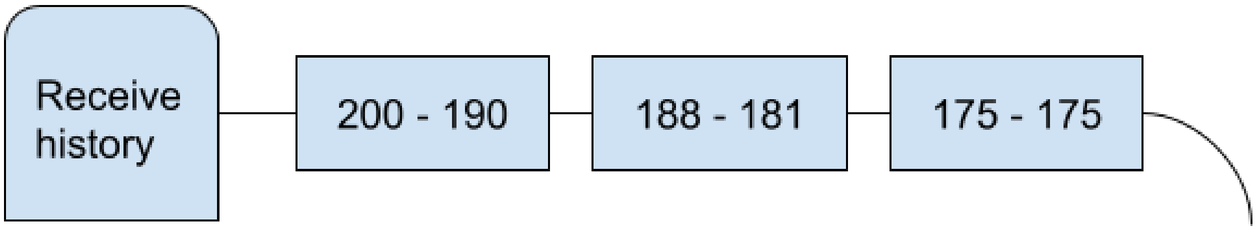
Overview

The reason for keeping the history of received packets is to generate ACK frames. The Receive History module provides functionality to add packet numbers, truncate history, and iterate over the received packet number ranges.

Data Structures

Overview

The receive history is a singly-linked list of packet number ranges, ordered from high to low:



The ordering is maintained as an invariant with each addition to the list and each truncation. This makes it trivial to iterate over the ranges.

To limit the amount of memory this data structure can allocate, the maximum number of elements is specified when Receive History is initialized. In the unlikely case that that number is reached, new elements will push out the elements at the tail of the linked list.

Memory Layout

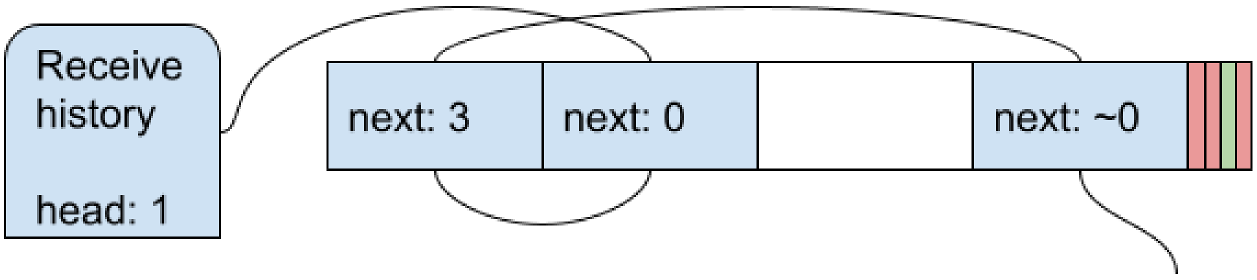
In memory, the linked list elements are stored in an array. Placing them into contiguous memory achieves three goals:

- Receive history manipulation is fast because the elements are all close together.
- Memory usage is reduced because each element does not use pointers to other memory locations.
- Memory fragmentation is reduced.

The array grows as necessary as the number of elements increases.

The elements are allocated from and returned to the array with the aid of an auxiliary data structure. An array of bitmasks is kept where each bit corresponds to an array element. A set bit means that the element is allocated; a cleared bit indicates that the corresponding element is free.

To take memory savings and speed further, the element array and the array of bitmasks are allocated in a single span of memory.



rechist_elem

`re_low` and `re_count` define the packet range. To save memory, we assume that the range will not contain more than 4 billion entries and use a four-byte integer instead of a second `lsquic_packno_t`.

`re_next` is the index of the next element. Again, we assume that there will be no more than 4 billion elements. The NULL pointer is represented by `UINT_MAX`.

This struct is just 16 bytes in size, which is a nice number.

lsquic_rechist

`rh_elems` and `rh_masks` are the element array and the bitmask array, respectively, as described above. The two use the same memory chunk.

`rh_head` is the index of the first element of the linked list.

The iterator state, `rh_iter`, is embedded into the main object itself, as there is no expectation that more than one iterations will need to be active at once.

Notable Code

Inserting Elements

Elements may be inserted into the list when a new packet number is added to history via `lsquic_rechist_received()`. If the new packet number requires a new range (e.g. it does not expand one of the existing ranges), a new element is allocated and linked.

There are four cases to consider:

1. Inserting the new element at the head of the list, with it becoming the new head. (This is the most common scenario.) The code that does it is labeled `first_elem`.
2. Appending the new element to the list, with it becoming the new tail. This code is located right after the `while` loop.
3. Inserting the new element between two existing elements. This code is labeled `insert_before`.
4. Like (3), but when the insertion is between the last and the next-to-last elements and the maximum number of elements has been reached. In this case, the last element's packet number information can simply be replaced. This code is labeled `replace_last_el`.

Growing the Array

When all allocated elements in `rh_elems` are in use (`rh_n_used >= rh_n_alloced`), the element array needs to be expanded. This is handled by the function `rechist_grow`.

Note how, after realloc, the bitmask array is moved to its new location on the right side of the array.

Handling Element Overflow

When the array has grown to its maximum allowed size, allocating a new element occurs via reusing the last element on the list, effectively pushing it out. This happens in `rechist_reuse_last_elem`.

The first loop finds the last linked list element: that's the element whose `re_next` is equal to `UINT_MAX`.

Then, the second loop finds the element that points to the last element. This is the next-to-last (penultimate) element. This element's next pointer will now be set to NULL, effectively dropping the last element, which can now be reused.

Iterating Over Ranges

Iteration is performed by the `lsquic_rechist_first` and `lsquic_rechist_next` pair of functions. The former resets the internal iterator. Only one iteration at a time is possible.

These functions have a specific signature: they and the pointer to the receive history are passed to the `pf_gen_ack_frame` function, which generates an ACK frame.

Clone Functionality

The Receive History can be initialized from another instance of a receive history. This is done by `lsquic_rechist_copy_ranges`. This functionality is used during connection promotion, when *Tiny Receive History* that is used by the *IETF mini connection* is converted to Receive History.

3.5.25 Tiny Receive History

Files: lsquic_trechist.h, lsquic_trechist.c, test_trechist.c

Overview

The Tiny Receive History is similar to *Receive History*, but it is even more frugal with memory. It is used in the *IETF mini connection* as a more costly *alternative to using bare bitmasks*.

Because it is so similar to Receive History, only differences are covered in this section.

Less Memory

No Trechist Type

There is no `lsquic_trechist`. The history is just a single 32-bit bitmask and a pointer to the array of elements. The bitmask and the pointer are passed to all `lsquic_trechist_*` functions.

This gives the user of Tiny Receive History some flexibility and saves memory.

Element

The linked list element, `trechist_elem`, is just 6 bytes in size. The assumptions are:

- No packet number is larger than 2^{32} - 1
- No packet range contains more than 255 packets
- Linked list is limited to 256 elements

Head Does Not Move

Because of memory optimizations described above, the head element is always at index 0. The NULL pointer `te_next` is indicated by the value 0 (because nothing points to the first element).

Array Does Not Grow

The size of the element array is limited by the 32-bit bitmask. As a further optimization, the number of ranges is limited to 16 via the `TRECHIST_MAX_RANGES` macro.

Insertion Range Check

A packet range spanning more than 255 (UCHAR_MAX) packets cannot be represented. This will cause a failure, as it is checked for in the code.

This many packets are unlikely to even be required to complete the handshake. If this limit is hit, it is perhaps good to abort the mini connection.

3.5.26 Set64

Files: lsquic_set.h, lsquic_set.h, test_set.c

This data structure (along with *Set32*, which is not currently used anywhere in the code) is meant to keep track of a set of numbers that are always increasing and are not expected to contain many gaps. Stream IDs fit that description, and `lsquic_set64` is used in both gQUIC and IETF QUIC full connections.

Because one or two low bits in stream IDs contain stream type, the stream IDs of different types are stored in different set structures; otherwise, there would be gaps. For example, see the `conn_is_stream_closed()` functions (there is one in each gQUIC and IETF QUIC full connection code).

3.5.27 Appendix A: List of Data Structures

The majority of data structures employed by lsquic are linked lists and, to a lesser extent, arrays. This makes the code simple and fast (assuming a smart memory layout).

Nevertheless, a few places in the code called for more involved and, at times, customized data structures. This appendix catalogues them.

This is the list of non-trivial data structures implemented in lsquic:

Ring Buffer Linked Lists

- *Receive History*
- *Tiny Receive History*

Hashes

- lsquic_hash
- hash_data_in

Min-heaps

- *Advisory Tick Time Queue*
- lsquic_min_heap

Bloom Filters

- CID Purgatory

3.5.28 Appendix B: Obsolete and Defunct Code

Mem Used

Engine History

QLOG

3.6 Frequently Asked Questions

3.6.1 API/Design

Why have a separate engine for server and client? Surely traffic could be differentiated as much as it needs to be internally in one engine?

The traffic *cannot* be differentiated for gQUIC versions Q046 and Q050. This is because in these versions, the server never includes a connection ID into the packets it sends to the client. To have more than one connection, then, the client must open a socket per connection: otherwise, the engine would not be able to dispatch incoming packets to correct connections.

To aid development, there is a *`LSQUIC_FORCED_TCID0_VERSIONS`* that specifies the list of versions with 0-sized connections. (If you, for example, want to turn them off.)

Once gQUIC becomes deprecated in the future, there will remain no technical reason why a single engine instance could not be used both for client and server connections. It will be just work. For example, the single engine settings *`lsquic_engine_settings`* will have to be separated into client and server settings, as the two usually do need to have separate settings.

3.6.2 Example Programs

http_client does not work with www.google.com, www.facebook.com, etc.

Check the version. By defaut, `http_client` will use the latest supported version (at the time of this writing, “h3-31”), while the server may be using an older version, such as “h3-29”. Adding `-o version=h3-29` to the command line may well solve your issue.

There is an outstanding bug where lsquic client does not perform version negotiation correctly for HTTP/3. We do not expect this to be fixed, because a) this version negotiation mechanism is likely to become defunct when QUIC v1 is released and b) version negotiation is not necessary for an HTTP/3 client, because the other side’s version is communicated to it via the `Alt-Svc` HTTP header.

CHAPTER 4

Indices and tables

- genindex
- search

Index