Workgroup: Internet Engineering Task Force
Internet-Draft: draft-lxin-quic-socket-apis-00
Published: 24 April 2024
Intended Status: Standards Track
Expires: 26 October 2024
Author: L. Xin, Ed.
Red Hat

Sockets API Extensions for In-kernel QUIC Implementations

Abstract

This document describes a mapping of these In-kernel QUIC Implementations into a sockets API. The benefits of this mapping include compatibility for TCP applications, access to new QUIC features, and a consolidated error and event notification scheme.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The sockets API has provided a standard mapping of the Internet Protocol suite to many operating systems. Both TCP [RFC9293] and UDP [RFC0768] have benefited from this standard representation and access method across many diverse platforms. SCTP [RFC6458] has also created its own sockets API. Base on [RFC6458], this document defines a method to map the existing sockets API for use with In-kernel QUIC, providing both a base for access to new features and compatibility so that most existing TCP applications can be migrated to QUIC with few (if any) changes.

Some of the QUIC mechanisms cannot be adequately mapped to an existing socket interface. In some cases, it is more desirable to have a new interface instead of using existing socket calls.

1.1. Conventions

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

2. Data Types

Whenever possible, Portable Operating System Interface (POSIX) data types defined in IEEE-1003.1-2008 are used: uintN_t means an unsigned integer of exactly N bits (e.g., uint16_t). This document also assumes the argument data types from POSIX when possible (e.g., the final argument to setsockopt() is a socklen_t value). Whenever buffer sizes are specified, the POSIX size_t data type is used.

3. Interface

A typical QUIC server uses the following socket calls in sequence to prepare an endpoint for servicing requests:

o socket()

o bind()

o listen()

o accept()

o quic_server_handshake()

o recvmsg()

o sendmsg()

o close()

It is similar to TCP server, except quic_server_handshake() call where the TLS message exchange happens to complete the handshake. See Section 3.2.2.

All TLS handshake messages carried in QUIC packets MUST be processed in userspace. The client initial packet received will cause accept() to create a new socket and return, but the TLS handshake message carried in it will be received via this new socket by quic_server_handshake().

A typical QUIC client uses the following calls in sequence to set up an association with a server to request services:

o socket()

o connect()

o quic_client_handshake()

o sendmsg()

o recvmsg()

o close()

It is similar to TCP client, except quic_client_handshake() call where the TLS message exchange happens to complete the handshake. See Section 3.2.2.

On client, connect() SHOULD not send any packet to server, and all TLS handshake messages are created via TLS library and sent out by quic_client_handshake().

In the implementation, one QUIC socket represents one QUIC connection and MAY hold multiple UDP sockets at the same time for connection migration or the future multiple path feature. Meanwhile, one lower UDP socket MAY serve for multiple QUIC sockets.

3.1. Basic Operation

3.1.1. socket()

Applications use socket() to create a socket descriptor to represent an QUIC endpoint.

The function prototype is

```
int socket(int domain,
           int type,
           int protocol);
```

and one uses PF_INET or PF_INET6 as the domain, SOCK_STREAM or SOCK_DGRAM as the type, and IPPROTO_QUIC as the protocol.

Note that QUIC does not possess a protocol number allocated from IANA, and like IPPROTO_MPTCP, IPPROTO_QUIC is merely a value used when opening a QUIC socket, and it can be defined with different value depending on the implementation.

The function returns a socket descriptor, or -1 in case of an error. Using the PF_INET domain indicates the creation of an endpoint that can use only IPv4 addresses, while PF_INET6 creates an endpoint that can use both IPv6 and IPv4 addresses.

3.1.2. bind()

Applications use bind() to specify with which local address and port the QUIC endpoint should associate itself.

The function prototype of bind() is

```
int bind(int sd,
         struct sockaddr *addr,
         socklen_t addrlen);
```

and the arguments are

- sd: The socket descriptor returned by socket().
- addr: The address structure (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3493]).
- addrlen: The size of the address structure.

bind() returns 0 on success and -1 in case of an error.

Applications cannot call bind() multiple times to associate multiple addresses to an endpoint. After the first call to bind(), all subsequent calls will return an error.

Multiple applications can bind() to the same address and the same port, and share the same lower UDP socket.

The IP address part of addr can be specified as a wildcard (INADDR_ANY for an IPv4 address, or as IN6ADDR_ANY_INIT or in6addr_any for an IPv6 address. It will return an error if the IPv4 sin_port or IPv6 sin6_port is set to 0.

If bind() is not called prior to connect() on client, the system picks an ephemeral port to bind in conenct().

The completion of this bind() process does not allow the QUIC endpoint to accept inbound QUIC association requests on server. Until a listen() system call, described below, is performed on the socket.

3.1.3. listen()

An application uses listen() to mark a socket as being able to accept new associations.

The function prototype is

```
int listen(int sd,
           int backlog);
```

and the arguments are

- sd: The socket descriptor of the endpoint.
- backlog: If backlog is non-zero, enable listening, else disable listening.

listen() returns 0 on success and -1 in case of an error.

Multiple applications binding to the same address and the same port must enable SO_REUSEPORT socket option for each socket before calling listen(). These listen sockets will be grouped so that incoming connections will be able to select the socket according to ALPN or with other selectors.

3.1.4. accept()

Applications use the accept() call to remove an established QUIC association from the accept queue of the endpoint. A new socket descriptor will be returned from accept() to represent the newly formed connection.

The function prototype is

```
int accept(int sd,
           struct sockaddr *addr,
           socklen_t *addrlen);
```

and the arguments are

- sd: The socket descriptor of the endpoint.
- addr: The address structure (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3542]).
- addrlen: The size of the address structure.

The function returns the socket descriptor for the newly formed connection on success and -1 in case of an error.

Note that the incoming Client Initial packet wakes the accept() up, and the TLS message carried by the Client Initial packet will queue up in the receive queue of the socket returned by accept(). Then it will be received by userspace via the socket returned by accept() so that the TLS message can be exchanged in userspace.

3.1.5. connect()

Applications use connect() to do routing and then find the proper source address and port to bind if bind() is not called, it also initializes the connection ID and installs initial keys to prepare for handshake.

The function prototype is

```
int connect(int sd,
            const struct sockaddr *addr,
            socklen_t addrlen);
```

and the arguments are

- sd: The socket descriptor of the endpoint.
- addr: The address structure (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3542]).
- addrlen: The size of the address structure.

connect() returns 0 on success and -1 on error.

connect() MUST be called before sending any handshake message.

3.1.6. close()

Applications use close() to gracefully close down an association.

The function prototype is

```
int close(int sd);
```

and the arguments are

- sd: The socket descriptor of the association to be closed.

close() returns 0 on success and -1 in case of an error.

After an application calls close() on a socket descriptor, no further socket operations will succeed on that descriptor.

close() will send CLOSE frame to peer, the close information can be set via QUIC_SOCKOPT_CONNECTION_CLOSE socket option before calling close(), see Section 6.1.2.

3.1.7. shutdown()

QUIC differs from TCP in that it does not have half close semantics.

The function prototypes are

```
int shutdown(int sd,
             int how);
```

and the arguments are

- sd: The socket descriptor of the association to be closed.
- how: Specifies the type of shutdown. 1. SHUT_RD: Disables further receive operations, the socket state is set to closed. 2. SHUT_WR: Disables further send operations, after sending CLOSE frame, the socket state is set to closed. 3. SHUT_RDWR: similar to SHUT_WR.

shutdown() returns 0 on success and -1 in case of an error.

Note that users can use SHUT_WR to send close frame multiple times.

3.1.8. sendmsg() and recvmsg()

An application uses the sendmsg() and recvmsg() calls to transmit data to and receive data from its peer.

The function prototypes are

```
ssize_t sendmsg(int sd,
                const struct msghdr *message,
                int flags);
ssize_t recvmsg(int sd,
                struct msghdr *message,
                int flags);
```

and the arguments are

- sd: The socket descriptor of the endpoint.

- message: Pointer to the msghdr structure that contains a single user message and possibly some ancillary data. See Section 4 for a complete description of the data structures.
- flags: No new flags are defined for QUIC at this level. See Section 4 for QUIC-specific flags used in the msghdr structure.

sendmsg() returns the number of bytes accepted by the kernel or -1 in case of an error. recvmsg() returns the number of bytes received or -1 in case of an error.

As described in Section 4, different types of ancillary data can be sent and received along with user data.

During Handshake, users can use sendmsg() and recvmsg() with Handshake msg_control Section 4.3.2 to send raw TLS messages to and receive from kernel and exchange TLS messages in userspace with the help of third-party TLS library like gnutls.

Two pairs of high level APIs are defined to wrap the handshake process in userspace, see Section 3.2.2 and Section 3.2.3.

Post Handshake, users can use sendmsg() and recvmsg() with Stream msg_control Section 4.3.1 to send data msgs to and receive from kernel with stream_id and stream_flags.

One pair of high level APIs are defined to wrap the stream msg_control, see Section 3.2.1.

3.1.9. send(), recv(), read() and write()

Applications can use send() and recv() to transmit data to the peer and receive data from the peer with basic access.

The function prototypes are

```
ssize_t send(int sd,
             const void *msg,
             size_t len,
             int flags);
ssize_t recv(int sd,
             void *buf,
             size_t len,
             int flags);
```

and the arguments are

- sd: The socket descriptor of the endpoint.
- msg: The message to be sent.
- len: The size of the message or the size of the buffer.
- flags: (described below).

send() returns the number of bytes accepted by the kernel or -1 in case of an error. recv() returns the number of bytes received or -1 in case of an error.

Since ancillary data (msg_control field) can NOT be used, the flags will work as stream_flags, and the latest opened stream will always be used as stream_id. see Section 4.1

send() and recv() can not be used to transmit and receive TLS messages as without ancillary data Handshake Information can be carried.

Applications can use read() and write() to receive and send data from and to a peer. They have the same semantics as recv() and send() but less access, as the flags parameter cannot be used.

3.1.10. setsockopt() and getsockopt()

Applications use setsockopt() and getsockopt() to set or retrieve socket options. Socket options are used to change the default behavior of socket calls. They are described in Section 6 .

The function prototypes are

```
int getsockopt(int sd,
               int level,
               int optname,
               void *optval,
               socklen_t *optlen);
int setsockopt(int sd,
               int level,
               int optname,
               const void *optval,
               socklen_t optlen);
```

and the arguments are

- sd: The socket descriptor.
- level: Set to SOL_QUIC for all QUIC options.
- optname: The option name.
- optval: The buffer to store the value of the option.
- optlen: The size of the buffer (or the length of the option returned).

These functions return 0 on success and -1 in case of an error.

3.1.11. getsockname() and getpeername()

Applications use getsockname() to retrieve the locally bound socket address of the specified socket and use getpeername() to retrieve the peer socket address. They are especially useful when connection migration occurs while the corresponding event is not enabled.

The function prototypes are

```
int getsockname(int sd,
                struct sockaddr *address,
                socklen_t *len);
int getpeername(int sd,
                struct sockaddr *address,
                socklen_t *len);
```

and the arguments are

- sd: The socket descriptor to be queried.
- address: On return, one locally bound or peer address (chosen by the QUIC stack) is stored in this buffer. If the socket is an IPv4 socket, the address will be IPv4. If the socket is an IPv6 socket, the address will be either an IPv6 or IPv4 address..
- len: The caller should set the length of the address here. On return, this is set to the length of the returned address.

These functions return 0 on success and -1 in case of an error.

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address will be truncated.

3.2. Advanced Operation

3.2.1. quic_sendmsg() and quic_recvmsg()

An application uses the quic_sendmsg() and quic_recvmsg() calls to transmit data to and receive data from its peer with stream_id and stream_flags.

The function prototypes are

```
ssize_t quic_sendmsg(int sd,
                     const void *msg,
                     size_t len,
                     uint64_t sid,
                     uint32_t flags);
ssize_t quic_recvmsg(int sd,
                     void *msg,
                     size_t len,
                     uint64_t *sid,
                     uint32_t *flags);
```

and the arguments are

- sd: The socket descriptor.
- msg: The message buffer to be filled.
- len: The length of the message buffer.
- sid: stream_id to point for sending or to get for receiving.
- flags: stream_flags to point for sending or to get for receiving.

quic_sendmsg() returns the number of bytes accepted by the kernel or -1 in case of an error. quic_recvmsg() returns the number of bytes received or -1 in case of an error.

These functions wrap the sendmsg() and recvmsg() with Stream information msg_control.

3.2.2. quic_client/server_handshake()

An application uses quic_client_handshake() or quic_server_handshake() to start a QUIC handshake with Certificate or PSK mode from client or server side.

The function prototypes are

```
int quic_server_handshake(int sd,
                          char *pkey_file,
                          char *cert_file);
int quic_client_handshake(int sd,
                          char *pkey_file,
                          char *cert_file);
```

and the arguments are

- sd: The socket descriptor.
- pkey_file: private key file or pre-shared key file.
- cert_file: certificate file or null.

These functions return 0 for success and errcode in case of an error.

These functions use the sendmsg() and recvmsg() with Handshake information msg_control to send and receive raw TLS messages from or to kernel and exchange them in userspace via TLS library like gnutls. Meanwhile, they use some socket options to get necessary information like Transport Parameters from kernel to build TLS messages, and set secrets derived for different levels to kernel for QUIC packets encryption and decryption.

3.2.3. quic_client/server_handshake_parms()

An application uses quic_client_handshake_parms() or quic_server_handshake_parms() to start a QUIC handshake from client or server side with more detailed TLS Handshake Parameters.

The function prototypes are

```
struct quic_handshake_parms {
  uint32_t          timeout;
  gnutls_privkey_t  privkey;
  gnutls_pcert_st   *cert;
  char              *peername;
  char              *names[10];
  gnutls_datum_t    keys[10];
  uint32_t          num_keys;
};
int quic_client_handshake_parms(int sd,
                                struct quic_handshake_parms *parms);
int quic_server_handshake_parms(int sd,
                                struct quic_handshake_parms *parms);
```

and the arguments are

- sd: The socket descriptor.
- parms: more TLS Handshake Parameters. 1. timeout: handshake timeout in milliseconds. 2. privkey: private key for x509 handshake. 3. cert: certificate for x509 handshake. 4. peername: server name for client side x509 handshake or psk identity name chosen during PSK handshake. 5. names[]: psk identifies in PSK handshake. 6. keys[]: psk keys in PSK handshake, or certificates received in x509 handshake. 7. num_keys: keys total numbers.

These functions return 0 for success and errcode in case of an error.

These functions are useful when adapting to the other userland QUIC handshake tools like ktls-utils.

4. Data Structures

This section discusses important data structures that are specific to QUIC and are used with sendmsg() and recvmsg() calls to control QUIC endpoint operations and to access ancillary information and notifications.

4.1. The msghdr and cmsghdr Structures

The msghdr structure used in the sendmsg() and recvmsg() calls, as well as the ancillary data carried in the structure, is the key for the application to set and get various control information from the QUIC endpoint.

The msghdr and the related cmsghdr structures are defined and discussed in detail in [RFC3542]. They are defined as

```
struct msghdr {
  void *msg_name;           /* ptr to socket address structure */
  socklen_t msg_namelen;    /* size of socket address structure */
  struct iovec *msg_iov;    /* scatter/gather array */
  int msg_iovlen;           /* # elements in msg_iov */
  void *msg_control;        /* ancillary data */
  socklen_t msg_controllen; /* ancillary data buffer length */
  int msg_flags;            /* flags on message */
};

struct cmsghdr {
  socklen_t cmsg_len; /* # bytes, including this header */
  int cmsg_level;     /* originating protocol */
  int cmsg_type;      /* protocol-specific type */
                      /* followed by unsigned char cmsg_data[]; */
};
```

The msg_name is not used when sending a message with sendmsg().

The scatter/gather buffers, or I/O vectors (pointed to by the msg_iov field) are treated by QUIC as a single user message for both sendmsg() and recvmsg().

The QUIC stack uses the ancillary data (msg_control field) to communicate the attributes, such as QUIC_STREAM_INFO, of the message stored in msg_iov to the socket endpoint. The different ancillary data types are described in Section 4.3.

On send side, msg_flags is used if QUIC_STREAM_INFO msg_control is not used, msg_flags works as the stream_flags in QUIC_STREAM_INFO msg_control.

- MSG_SYN: equals QUIC_STREAM_FLAG_NEW in stream_flags
- MSG_FIN: equals QUIC_STREAM_FLAG_FIN
- MSG_DONTWAIT: equals QUIC_STREAM_FLAG_ASYNC
- MSG_STREAM_UNI: equals QUIC_STREAM_FLAG_UNI
- MSG_DATAGRAM: equals QUIC_STREAM_FLAG_NOTIFICATION

On receive side, msg_flags is always set

- MSG_EOR: equals QUIC_STREAM_FLAG_FIN in stream_flags
- MSG_PEEK
- MSG_DONTWAIT
- MSG_NOTIFICATION: equals QUIC_STREAM_FLAG_NOTIFICATION
- MSG_DATAGRAM: equals QUIC_STREAM_FLAG_NOTIFICATION

This means users can send/receive stream data without QUIC_STREAM_INFO msg_control. However, as stream_id will be pointed/passed, the latest opened stream will always be used, and MSG_SYN flag will be used to open the next available stream. Therefore, if a user wants to operate multiple streams at the same time, QUIC_STREAM_INFO msg_control must be used.

4.2. Ancillary Data Considerations and Semantics

Programming with ancillary socket data (msg_control) contains some subtleties and pitfalls, which are discussed below.

4.2.1. Multiple Items and Ordering

Multiple ancillary data items may be included in any call to sendmsg() or recvmsg(); these may include multiple QUIC items, non-QUIC items (such as IP-level items), or both.

The ordering of ancillary data items (either by QUIC or another protocol) is not significant and is implementation dependent, so applications must not depend on any ordering.

QUIC_STREAM_INFO and QUIC_HANDSHAKE_INFO type ancillary data always corresponds to the data in the msghdr's msg_iov member. There can be only one such type of ancillary data for each sendmsg() or recvmsg() call.

4.2.2. Accessing and Manipulating Ancillary Data

Applications can infer the presence of data or ancillary data by examining the msg_iovlen and msg_controllen msghdr members, respectively

Implementations may have different padding requirements for ancillary data, so portable applications should make use of the macros CMSG_FIRSTHDR, CMSG_NXTHDR, CMSG_DATA, CMSG_SPACE, and CMSG_LEN. See [RFC3542] for more information. The following is an example, from [RFC3542], demonstrating the use of these macros to access ancillary data

```
struct msghdr msg;
struct cmsghdr *cmsgptr;

/* fill in msg */

/* call recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
  if (cmsgptr->cmsg_len == 0) {
     /* Error handling */
     break;
  }
  if (cmsgptr->cmsg_level == ... && cmsgptr->cmsg_type == ... ) {
    u_char  *ptr;

    ptr = CMSG_DATA(cmsgptr);
    /* process data pointed to by ptr */
  }
}
```

4.2.3. Control Message Buffer Sizing

The information conveyed via QUIC_STREAM_INFO and QUIC_HANDSHAKE_INFO ancillary data will often be fundamental to the correct and sane operation of the sockets application. For example, if an application needs to send and receive data on different QUIC streams, QUIC_STREAM_INFO ancillary data is indispensable.

Given that some ancillary data is critical, and that multiple ancillary data items may appear in any order, applications should be carefully written to always provide a large enough buffer to contain all possible ancillary data that can be presented by recvmsg(). If the buffer is too small, and crucial data is truncated, it may pose a fatal error condition.

Thus, it is essential that applications be able to deterministically calculate the maximum required buffer size to pass to recvmsg(). One constraint imposed on this specification that makes this possible is that all ancillary data definitions are of a fixed length. One way to calculate the maximum required buffer size might be to take the sum of the sizes of all enabled ancillary data item structures, as calculated by CMSG_SPACE. For example, if we enabled QUIC_STREAM_INFO and IPV6_RECVPKTINFO [RFC3542], we would calculate and allocate the buffer size as follows

```
size_t total;
void *buf;

total = CMSG_SPACE(sizeof(struct quic_stream_info)) +
        CMSG_SPACE(sizeof(struct in6_pktinfo));

buf = malloc(total);
```

We could then use this buffer (buf) for msg_control on each call to recvmsg() and be assured that we would not lose any ancillary data to truncation.

4.3. QUIC msg_control Structures

4.3.1. Stream Information

This cmsg specifies QUIC options for sendmsg() and describes QUIC header information about a received message through recvmsg() with struct quic_stream_info.

```
struct quic_stream_info {
  uint64_t stream_id;
  uint32_t stream_flag;
};
```

For stream_id, the first 2 bits are for the stream type for sending or receiving

- QUIC_STREAM_TYPE_SERVER_MASK: 0x1, server-side stream
- QUIC_STREAM_TYPE_UNI_MASK: 0x2, unidirectional stream

For stream_flag on send side

- QUIC_STREAM_FLAG_NEW: open a stream and send the first data
- QUIC_STREAM_FLAG_FIN: send the last data and close a stream
- QUIC_STREAM_FLAG_DATAGRAM: send data as datagram

For stream_flag on receive side

- QUIC_STREAM_FLAG_NOTIFICATION: data received is an event
- QUIC_STREAM_FLAG_FIN: data received is the last one for this stream
- QUIC_STREAM_FLAG_DATAGRAM: data received is datagram

4.3.2. Handshake Information

This cmsg provides information for sending and receiving handshake/TLS messages via sendmsg() or recvmsg() with struct quic_handshake_info.

```
struct quic_handshake_info {
  uint8_t crypto_level;
};
```

crypto_level includes these levels

```
enum quic_crypto_level {
  QUIC_CRYPTO_APP,
  QUIC_CRYPTO_INITIAL,
  QUIC_CRYPTO_HANDSHAKE,
  QUIC_CRYPTO_EARLY,
};
```

So this cmsg is only used inside handshake APIs.

5. QUIC Events and Notifications

An QUIC application may need to understand and process events and errors that happen on the QUIC stack. These events include stream updates and max_streams, connection close and migration, key updates, new token.

```
enum quic_event_type {
  QUIC_EVENT_NONE,
  QUIC_EVENT_STREAM_UPDATE,
  QUIC_EVENT_STREAM_MAX_STREAM,
  QUIC_EVENT_CONNECTION_CLOSE,
  QUIC_EVENT_CONNECTION_MIGRATION,
  QUIC_EVENT_KEY_UPDATE,
  QUIC_EVENT_NEW_TOKEN,
};
```

When a notification arrives, recvmsg() returns the notification in the application-supplied data buffer via msg_iov, and sets MSG_NOTIFICATION in msg_flags of msghdr and QUIC_STREAM_FLAG_NOTIFICATION in stream_flags of cmsg quic_stream_info in Section 4.3.1

See socket option Section 5.2.1 for the event enabling.

5.1. QUIC Notification Structure

5.1.1. QUIC_EVENT_STREAM_UPDATE

Only the notification with one of these states is sent to userspace

- QUIC_STREAM_SEND_STATE_RECVD
- QUIC_STREAM_SEND_STATE_RESET_SENT: update is sent only if STOP_SENDING is received
- QUIC_STREAM_SEND_STATE_RESET_RECVD
- QUIC_STREAM_RECV_STATE_RECV: update is sent only when the last frag hasn't arrived.
- QUIC_STREAM_RECV_STATE_SIZE_KNOWN: update is sent only if data comes out of order
- QUIC_STREAM_RECV_STATE_RECVD
- QUIC_STREAM_RECV_STATE_RESET_RECVD

Data format in the event

```
struct quic_stream_update {
  uint64_t id;
  uint32_t state;
  uint32_t errcode;
  uint64_t finalsz;
};
```

5.1.2. QUIC_EVENT_STREAM_MAX_STREAM

This notification is sent when max_streams frame is received, and this is useful when using QUIC_STREAM_FLAG_ASYNC to open a stream whose id exceeds the max stream count. After receiving this notification, try to open this stream again.

Data format in the event

```
uint64_t max_stream;
```

5.1.3. QUIC_EVENT_CONNECTION_CLOSE

This notification is sent when receiving a close frame from peer where it can set the close info with Section 6.1.2 socket option.

Data format in the event

```
struct quic_connection_close {
  uint32_t errcode;
  uint8_t frame;
  uint8_t phrase[];
};
```

5.1.4. QUIC_EVENT_CONNECTION_MIGRATION

This notification is sent when either side successfully changes its source address by Section 6.3.3 socket option or dest address by peer's CONNECTION_MIGRATION. The parameter tells you if it is a local or peer CONNECTION_MIGRATION, and then you can get the new address with getsockname() or getpeername().

Data format in the event

```
uint8_t local_migration;
```

5.1.5. QUIC_EVENT_KEY_UPDATE

This notification is sent when both sides have used the new key after key update, and the parameter tells you which the new key phase is

Data format in the event

```
uint8_t key_update_phase;
```

5.1.6. QUIC_EVENT_NEW_TOKEN

Since the handshake is in userspace, this notification is sent whenever the frame of NEW_TOKEN is received from the peer where it can send these frame via Section 6.1.4 socket option.

Data format in the event

```
uint8_t *token;
```

5.2. Notification Interest Options

5.2.1. QUIC_SOCKOPT_EVENT Option

This option is used to enable or disable one type of event or notification.

the optval type is

```
struct quic_event_option {
  uint8_t type;
  uint8_t on;
};
```

type is defined on Section 5.1.

on can be set to

- 0: disable.
- !0: enable.

all events are disabled by default.

6. Socket Options

6.1. Read/Write Options

6.1.1. QUIC_SOCKOPT_EVENT

This socket option is used to set a specific notification option. Please see Section 5.2.1 for a full description of this option and its usage.

6.1.2. QUIC_SOCKOPT_CONNECTION_CLOSE

This option is used to get or get the close context, which includes errcode and phrase and frame. On close side, set it before calling close() to tell peer the closing info, while on being closed side get it to show the peer closing info.

the optval type is

```
struct quic_connection_close {
  uint32_t errcode;
  uint8_t frame;
  uint8_t phrase[];
};
```

errcode is Application Protocol Error Code left to application protocols.

phrase is a string to describe more details.

frame is the frame type that caused the closing.

All three are 0 or null by default.

6.1.3. QUIC_SOCKOPT_TRANSPORT_PARAM

This option is used to configure the transport parameters, including not only the quic original transport param, but also some handshake options.

the optval type is

```
struct quic_transport_param {
  uint8_t         remote;
  uint8_t         disable_active_migration; (0 by default)
  uint8_t         grease_quic_bit; (0)
  uint8_t         stateless_reset; (0)
  uint8_t         disable_1rtt_encryption; (0)
  uint8_t         disable_compatible_version; (0)
  uint64_t        max_udp_payload_size; (65527)
  uint64_t        ack_delay_exponent; (3)
  uint64_t        max_ack_delay; (25000)
  uint64_t        active_connection_id_limit; (7)
  uint64_t        max_idle_timeout; (30000000 us)
  uint64_t        max_datagram_frame_size; (0)
  uint64_t        max_data; (sk_rcvbuf / 2)
  uint64_t        max_stream_data_bidi_local; (sk_rcvbuf / 4)
  uint64_t        max_stream_data_bidi_remote; (sk_rcvbuf / 4)
  uint64_t        max_stream_data_uni; (sk_rcvbuf / 4)
  uint64_t        max_streams_bidi; (100)
  uint64_t        max_streams_uni; (100)
  uint64_t        initial_smoothed_rtt; (333000)

  uint32_t        plpmtud_probe_timeout; (0)
  uint8_t         validate_peer_address; (0)
  uint8_t         receive_session_ticket; (0)
  uint8_t         certificate_request; (0)
  uint8_t         congestion_control_alg; (QUIC_CONG_ALG_RENO)
  uint32_t        payload_cipher_type; (0)
  uint32_t        version; (QUIC_VERSION_V1)
};
```

These members in the 1st group are from [RFC9000], and the members in the 2nd group are

plpmtud_probe_timeout is in usec, 0: disabled.

validate_peer_address is for server only, send retry packet and verify token.

receive_session_ticket is for client only, handshake is done until ticket is received

certificate_request is for server only, and can be set to

- 0: IGNORE
- 1: REQUEST
- 2: REQUIRE

congestion_control_alg is congestion control algorithm

payload_cipher_type can be set to

- AES_GCM_128
- AES_GCM_256

- AES_CCM_128
- CHACHA20_POLY1305

version can be set to

- QUIC_VERSION_V1
- QUIC_VERSION_V2

The default values are inline the struct code.

Note 'remote' member allows users to set remote transport parameter. Together with the session resumption ticket, it is used to set the remote transport parameter from last connection before sending 0-RTT DATA.

6.1.4. QUIC_SOCKOPT_TOKEN

On Client this option is used to set regular token, which is used for the peer server's address verification. The token is usually issued by peer from the last connection and got via setsockopt with this option or Section 5.1.6 event.

On Server this option is used to issue the token to Client for the next connection's address verification.

On Client the optval type is

```
uint8_t *opt;
```

On Server the optval type is NULL.

The default value in socket is NULL.

6.1.5. QUIC_SOCKOPT_ALPN

This option is used to set or get the Application-Layer Protocol Negotiation before handshake, multiple ALPNs are separated by ',' e.g. "smbd, h3, ksmbd".

On server side, during handshake it gets ALPN via this socket option and matches the ALPN from the client side, and then sets the matched ALPN to the socket, so that users can get the selected ALPN via this socket option after handshake.

The optval type is

```
char *alpn;
```

The default value in socket is NULL.

6.1.6. QUIC_SOCKOPT_SESSION_TICKET

This option is used to set session resumption ticket on Client, which is used for session resumption. The ticket is usually issued by peer from the last connection and got via setsockopt with this option.

On client the optval type is

```
uint8_t *opt;
```

On Server the optval type is NULL.

The default value in socket is NULL.

6.1.7. QUIC_SOCKOPT_CRYPTO_SECRET

This option is used to set the secret (not keys) derived from the userspace to kernel socket during the handshake.

On Client the optval type is

```
struct quic_crypto_secret {
  uint8_t level;
  uint16_t send;
  uint32_t type;
  uint8_t secret[48];
};
```

level can be set to

- QUIC_CRYPTO_APP: set secret for application level
- QUIC_CRYPTO_HANDSHAKE: set secret for handshake level
- QUIC_CRYPTO_EARLY: set secret for early/0rtt level

send can be set to

- 0: set secret for receive.
- !0: set secret for send.

type can be set to

- AES_GCM_128
- AES_GCM_256
- AES_CCM_128
- CHACHA20_POLY1305

secret is the key meterial to set and the length depends on type

This option is only used for doing handshake.

6.1.8. QUIC_SOCKOPT_TRANSPORT_PARAM_EXT

This option is used to get the QUIC Transport Extension from kernel to build the TLS message and set the QUIC Transport Extension from the TLS message received from the peer.

```
uint8_t *opt;
```

This option is only used for doing handshake.

6.2. Read-Only Options

6.2.1. QUIC_SOCKOPT_STREAM_OPEN

This option is used to open a stream.

the optval type is

```
struct quic_stream_info {
  uint64_t stream_id;
  uint32_t stream_flag;
};
```

stream_id can be set to

- >= 0: open a stream with a specific stream id.
- -1: open next available stream and return the stream id to users via stream_id.

stream_flag can be set to

- QUIC_STREAM_FLAG_UNI: open the next unidirectional stream.
- QUIC_STREAM_FLAG_ASYNC: open the stream without block

6.3. Write-Only Options

6.3.1. QUIC_SOCKOPT_STREAM_RESET

This option is used to reset a stream and it means that the endpoint will not guarantee delivery of stream data.

the optval type is

```
struct quic_errinfo {
  uint64_t stream_id;
  uint32_t errcode;
};
```

errcode is Application Protocol Error Code left to application protocols.

6.3.2. QUIC_SOCKOPT_STREAM_STOP_SENDING

This option is used to request that a peer cease transmission on a stream.

the optval type is

```
struct quic_errinfo {
  uint64_t stream_id;
  uint32_t errcode;
};
```

errcode is Application Protocol Error Code left to application protocols.

6.3.3. QUIC_SOCKOPT_CONNECTION_MIGRATION

This option is used to initiate a connection migration. It can also be used to set preferred_address transport param before handshake on server side.

the optval type is

```
struct sockaddr_in(6);
```

6.3.4. QUIC_SOCKOPT_KEY_UPDATE

This option is used to initiate a key update or rekeying with the optval == NULL

7. IANA Considerations

No actions from IANA required.

8. Security Considerations

The socket receive buffer SHOULD be adjusted by the local max_data from struct quic_transport_param, so the implementation should change the socket receive buffer whenever the local transport param max_data changes, it may impair performance with socket receive buffer smaller than the local transport param max_data.

The socket send buffer SHOULD also be adjusted by the peer max_data of transport param to get the best performance, instead of setting it manually.

The optval size of these sockopt options, QUIC_SOCKOPT_ALPN, QUIC_SOCKOPT_TOKEN, QUIC_SOCKOPT_SESSION_TICKET, QUIC_SOCKOPT_CONNECTION_CLOSE, must be limited to avoid too much memory allocation.

9. References

9.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[RFC9293] Eddy, W., Ed., "Transmission Control Protocol (TCP)", STD 7, RFC 9293, DOI 10.17487/RFC9293, August 2022, <https://www.rfc-editor.org/info/rfc9293>.

9.2. Informative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3542] Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei, "Advanced Sockets Application Program Interface (API) for IPv6", RFC 3542, DOI 10.17487/RFC3542, May 2003, <https://www.rfc-editor.org/info/rfc3542>.

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, DOI 10.17487/RFC3493, February 2003, <https://www.rfc-editor.org/info/rfc3493>.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <https://www.rfc-editor.org/info/rfc6458>.

Appendix A. Example For Multi-streaming Usage

This example shows how to use quic_sendmsg() and quic_recvmsg() to send and receive messages on multiple streams at the same time.

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#include <netinet/quic.h>

struct stream {
```

```
    char msg[50];
    uint32_t len;
    uint32_t flags;
};

static int do_client(int argc, char *argv[])
{
    struct stream stream[2] = {};
    struct sockaddr_in ra = {};
    int ret, sockfd;
    uint32_t flags;
    uint64_t sid;
    char msg[50];

    if (argc < 3) {
        printf("%s client <PEER ADDR> <PEER PORT>\n", argv[0]);
        return 0;
    }

    sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_QUIC);
    if (sockfd < 0) {
        printf("socket create failed\n");
        return -1;
    }

    ra.sin_family = AF_INET;
    ra.sin_port = htons(atoi(argv[3]));
    inet_pton(AF_INET, argv[2], &ra.sin_addr.s_addr);

    if (connect(sockfd, (struct sockaddr *)&ra, sizeof(ra))) {
        printf("socket connect failed\n");
        return -1;
    }

    if (quic_client_handshake(sockfd, NULL, NULL))
        return -1;

    /* Open stream 0 and send first data on stream 0 */
    strcpy(msg, "hello ");
    sid = 0;
    flags = QUIC_STREAM_FLAG_NEW;
    ret = quic_sendmsg(sockfd, msg, strlen(msg), sid, flags);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    stream[sid >> 1].len += ret;

    /* Open stream 2 and send first data on stream 2 */
    strcpy(msg, "hello quic ");
    sid = 2;
    flags = QUIC_STREAM_FLAG_NEW;
    ret = quic_sendmsg(sockfd, msg, strlen(msg), sid, flags);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    stream[sid >> 1].len += ret;
```

```
    /* Send second data on stream 0 */
    strcpy(msg, "quic ");
    sid = 0;
    flags = 0;
    ret = quic_sendmsg(sockfd, msg, strlen(msg), sid, flags);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    stream[sid >> 1].len += ret;

    /* Send second (last) data on stream 2 */
    strcpy(msg, "server stream 2!");
    sid = 2;
    flags = QUIC_STREAM_FLAG_FIN;
    ret = quic_sendmsg(sockfd, msg, strlen(msg), sid, flags);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    stream[sid >> 1].len += ret;

    /* Send third (last) data on stream 0 */
    strcpy(msg, "server stream 0!");
    sid = 0;
    flags = QUIC_STREAM_FLAG_FIN;
    ret = quic_sendmsg(sockfd, msg, strlen(msg), sid, flags);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    stream[sid >> 1].len += ret;
    sid = 0;
    printf("send %d, len: %u, sid: %lu\n", ret,
           stream[sid >> 1].len, sid);
    sid = 2;
    printf("send %d, len: %u, sid: %lu\n", ret,
           stream[sid >> 1].len, sid);

    memset(msg, 0, sizeof(msg));
    ret = quic_recvmsg(sockfd, msg, sizeof(msg), &sid, &flags);
    if (ret == -1) {
        printf("recv error %d %d\n", ret, errno);
        return 1;
    }
    printf("recv: \"%s\", len: %d, sid: %lu\n", msg, ret, sid);

    close(sockfd);
    return 0;
}

static int do_server(int argc, char *argv[])
{
    struct stream stream[2] = {};
    struct sockaddr_in sa = {};
    int listenfd, sockfd, ret;
    unsigned int addrlen;
```

```
    uint32_t flags;
    uint64_t sid;
    char msg[50];

    if (argc < 5) {
        printf("%s server <LOCAL ADDR> <LOCAL PORT>"
               "<PRIVATE_KEY_FILE> <CERTIFICATE_FILE>\n", argv[0]);
        return 0;
    }

    sa.sin_family = AF_INET;
    sa.sin_port = htons(atoi(argv[3]));
    inet_pton(AF_INET, argv[2], &sa.sin_addr.s_addr);
    listenfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_QUIC);
    if (listenfd < 0) {
        printf("socket create failed\n");
        return -1;
    }
    if (bind(listenfd, (struct sockaddr *)&sa, sizeof(sa))) {
        printf("socket bind failed\n");
        return -1;
    }
    if (listen(listenfd, 1)) {
        printf("socket listen failed\n");
        return -1;
    }
    addrlen = sizeof(sa);
    sockfd = accept(listenfd, (struct sockaddr *)&sa, &addrlen);
    if (sockfd < 0) {
        printf("socket accept failed %d %d\n", errno, sockfd);
        return -1;
    }

    if (quic_server_handshake(sockfd, argv[4], argv[5]))
        return -1;

    while (!(stream[0].flags & QUIC_STREAM_FLAG_FIN) ||
           !(stream[1].flags & QUIC_STREAM_FLAG_FIN)) {
        ret = quic_recvmsg(sockfd, msg, sizeof(msg), &sid, &flags);
        if (ret == -1) {
            printf("recv error %d %d\n", ret, errno);
            return 1;
        }
        sid >>= 1;
        memcpy(stream[sid].msg + stream[sid].len, msg, ret);
        stream[sid].len += ret;
        stream[sid].flags = flags;
    }
    sid = 0;
    printf("recv: \"%s\", len: %d, sid: %lu\n",
           stream[sid >> 1].msg, stream[sid >> 1].len, sid);
    sid = 2;
    printf("recv: \"%s\", len: %d, sid: %lu\n",
           stream[sid >> 1].msg, stream[sid >> 1].len, sid);

    strcpy(msg, "hello quic client stream 1!");
    sid = 1;
    flags = QUIC_STREAM_FLAG_NEW | QUIC_STREAM_FLAG_FIN;
```

```
    ret = quic_sendmsg(sockfd, msg, strlen(msg), sid, flags);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    printf("send %d, sid: %lu\n", ret, sid);

    close(sockfd);
    close(listenfd);
    return 0;
}

int main(int argc, char *argv[])
{
    if (argc < 2 || (strcmp(argv[1], "server") &&
        strcmp(argv[1], "client"))) {
        printf("%s server|client ...\n", argv[0]);
        return 0;
    }

    if (!strcmp(argv[1], "client"))
        return do_client(argc, argv);

    return do_server(argc, argv);
}
```

Appendix B. Example For Session Consumption and 0-RTT transmission

This example shows how to combine socket option QUIC_SOCKOPT_TOKEN, QUIC_SOCKOPT_SESSION_TICKET and QUIC_SOCKOPT_TRANSPORT_PARAM to achieve Session Consumption and 0-RTT transmission.

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#include <netinet/quic.h>

static uint8_t ticket[4096];
static uint8_t token[256];

static int do_client(int argc, char *argv[])
{
    unsigned int ticket_len, param_len, token_len, addr_len;
    struct quic_transport_param param = {};
    struct sockaddr_in ra = {}, la = {};
    int ret, sockfd;
    char msg[50];
```

```
    if (argc < 3) {
        printf("%s client <PEER ADDR> <PEER PORT>\n", argv[0]);
        return 0;
    }

    sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_QUIC);
    if (sockfd < 0) {
        printf("socket create failed\n");
        return -1;
    }

    ra.sin_family = AF_INET;
    ra.sin_port = htons(atoi(argv[3]));
    inet_pton(AF_INET, argv[2], &ra.sin_addr.s_addr);

    if (connect(sockfd, (struct sockaddr *)&ra, sizeof(ra))) {
        printf("socket connect failed\n");
        return -1;
    }

    param.receive_session_ticket = 1;
    param_len = sizeof(param);
    ret = setsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_TRANSPORT_PARAM,
                     &param, param_len);
    if (ret == -1)
        return -1;

    if (quic_client_handshake(sockfd, NULL, NULL))
        return -1;

    /* get ticket and param after handshake (you can save
     * it somewhere).
     */
    ticket_len = sizeof(ticket);
    ret = getsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_SESSION_TICKET,
                     ticket, &ticket_len);
    if (ret == -1 || !ticket_len) {
        printf("socket getsockopt session ticket\n");
        return -1;
    }

    param_len = sizeof(param);
    param.remote = 1;
    ret = getsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_TRANSPORT_PARAM,
                     &param, &param_len);
    if (ret == -1) {
        printf("socket getsockopt remote transport param\n");
        return -1;
    }

    /* get token and local address (needed when peer
     * validate_address is set).
     */
    token_len = sizeof(token);
    ret = getsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_TOKEN, &token,
                     &token_len);
    if (ret == -1) {
        printf("socket getsockopt regular token\n");
```

```
            return -1;
        }

        addr_len = sizeof(la);
        ret = getsockname(sockfd, (struct sockaddr *)&la, &addr_len);
        if (ret == -1) {
            printf("getsockname local address and port used\n");
            return -1;
        }

        printf("get the session ticket %d and transport param %d and"
               "token %d, save it\n", ticket_len, param_len, token_len);

        strcpy(msg, "hello quic server!");
        ret = send(sockfd, msg, strlen(msg), MSG_SYN | MSG_FIN);
        if (ret == -1) {
            printf("send error %d %d\n", ret, errno);
            return -1;
        }
        printf("send %d\n", ret);

        memset(msg, 0, sizeof(msg));
        ret = recv(sockfd, msg, sizeof(msg), 0);
        if (ret == -1) {
            printf("recv error %d %d\n", ret, errno);
            return 1;
        }
        printf("recv: \"%s\", len: %d\n", msg, ret);

        close(sockfd);

        printf("start new connection with the session ticket used...\n");
        sleep(2);

        sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_QUIC);
        if (sockfd < 0) {
            printf("socket create failed\n");
            return -1;
        }

        /* bind previous address and port and set token for
         * address validation.
         */
        if (bind(sockfd, (struct sockaddr *)&la, addr_len)) {
            printf("socket bind failed\n");
            return -1;
        }
        ret = setsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_TOKEN, token,
                         token_len);
        if (ret == -1) {
            printf("socket setsockopt token\n");
            return -1;
        }

        ra.sin_family = AF_INET;
        ra.sin_port = htons(atoi(argv[3]));
        inet_pton(AF_INET, argv[2], &ra.sin_addr.s_addr);
```

```
    if (connect(sockfd, (struct sockaddr *)&ra, sizeof(ra))) {
        printf("socket connect failed\n");
        return -1;
    }

    /* set the ticket and remote param and early data into
     * the socket for handshake.
     */
    ret = setsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_SESSION_TICKET,
                     ticket, ticket_len);
    if (ret == -1) {
        printf("socket setsockopt session ticket\n");
        return -1;
    }
    ret = setsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_TRANSPORT_PARAM,
                     &param, param_len);
    if (ret == -1) {
        printf("socket setsockopt remote transport param\n");
        return -1;
    }
    strcpy(msg, "hello quic server, I'm back!");
    ret = send(sockfd, msg, strlen(msg), MSG_SYN | MSG_FIN);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    printf("send %d\n", ret);

    if (quic_client_handshake(sockfd, NULL, NULL))
        return -1;

    memset(msg, 0, sizeof(msg));
    ret = recv(sockfd, msg, sizeof(msg), 0);
    if (ret == -1) {
        printf("recv error %d %d\n", ret, errno);
        return 1;
    }
    printf("recv: \"%s\", len: %d\n", msg, ret);

    close(sockfd);
    return 0;
}

static int do_server(int argc, char *argv[])
{
    struct quic_transport_param param = {};
    struct sockaddr_in sa = {};
    int listenfd, sockfd, ret;
    unsigned int addrlen;
    char msg[50];

    if (argc < 5) {
        printf("%s server <LOCAL ADDR> <LOCAL PORT>"
               "<PRIVATE_KEY_FILE> <CERTIFICATE_FILE>\n", argv[0]);
        return 0;
    }

    sa.sin_family = AF_INET;
```

```
    sa.sin_port = htons(atoi(argv[3]));
    inet_pton(AF_INET, argv[2], &sa.sin_addr.s_addr);
    listenfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_QUIC);
    if (listenfd < 0) {
        printf("socket create failed\n");
        return -1;
    }
    if (bind(listenfd, (struct sockaddr *)&sa, sizeof(sa))) {
        printf("socket bind failed\n");
        return -1;
    }
    if (listen(listenfd, 1)) {
        printf("socket listen failed\n");
        return -1;
    }
    param.validate_peer_address = 1;
    if (setsockopt(listenfd, SOL_QUIC, QUIC_SOCKOPT_TRANSPORT_PARAM,
                   &param, sizeof(param)))
        return -1;
    addrlen = sizeof(sa);
    sockfd = accept(listenfd, (struct sockaddr *)&sa, &addrlen);
    if (sockfd < 0) {
        printf("socket accept failed %d %d\n", errno, sockfd);
        return -1;
    }

    if (quic_server_handshake(sockfd, argv[4], argv[5]))
        return -1;

    memset(msg, 0, sizeof(msg));
    ret = recv(sockfd, msg, sizeof(msg), 0);
    if (ret == -1) {
        printf("recv error %d %d\n", ret, errno);
        return 1;
    }
    printf("recv: \"%s\", len: %d\n", msg, ret);

    strcpy(msg, "hello quic client!");
    ret = send(sockfd, msg, strlen(msg), MSG_SYN | MSG_FIN);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    printf("send %d\n", ret);

    close(sockfd);

    printf("wait for the client next connection...\n");

    addrlen = sizeof(sa);
    sockfd = accept(listenfd, (struct sockaddr *)&sa, &addrlen);
    if (sockfd < 0) {
        printf("socket accept failed %d %d\n", errno, sockfd);
        return -1;
    }

    if (quic_server_handshake(sockfd, argv[4], argv[5]))
        return -1;
```

```
    memset(msg, 0, sizeof(msg));
    ret = recv(sockfd, msg, sizeof(msg), 0);
    if (ret == -1) {
        printf("recv error %d %d\n", ret, errno);
        return 1;
    }
    printf("recv: \"%s\", len: %d\n", msg, ret);

    strcpy(msg, "hello quic client! welcome back!");
    ret = send(sockfd, msg, strlen(msg), MSG_SYN | MSG_FIN);
    if (ret == -1) {
        printf("send error %d %d\n", ret, errno);
        return -1;
    }
    printf("send %d\n", ret);

    close(sockfd);
    close(listenfd);
    return 0;
}

int main(int argc, char *argv[])
{
    if (argc < 2 || (strcmp(argv[1], "server") &&
        strcmp(argv[1], "client"))) {
        printf("%s server|client ...\n", argv[0]);
        return 0;
    }

    if (!strcmp(argv[1], "client"))
        return do_client(argc, argv);

    return do_server(argc, argv);
}
```

Appendix C. Example For Kernel Consumers Archtechiture Design

In-kernel QUIC enables the usage for kernel consumers, here is the design in Linux Kernel

Handshake Archtechiture

```
+------+  +------+
| APP1 |  | APP2 | ...
+------+  +------+
+---------------------------------------------+
|            libquic (ktls-utils)           |<----------+
|  {quic_handshake_server/client/param()}    |           |
+---------------------------------------------+  +------------------+
 {send/recvmsg()}       {set/getsockopt()}      |tlshd (ktls-utils)|
 [CMSG handshake_info]  [SOCKOPT_CRYPTO_SECRET] +------------------+
                        [SOCKOPT_TRANSPORT_PARAM_EXT]
            | ^                    | ^                     | ^
Userspace   | |                    | |                     | |
------------|-|--------------------|-|---------------------|-|------
Kernel      | |                    | |                     | |
            v |                    v |                     v |
+----------------------------------------------+    +-------------+
| socket (IPPRTOTO_QUIC) |      protocol      |<-+ | handshake   |
+----------------------------------------------+  | | netlink APIs|
| stream | connection_id | cong | path |timer |  | +-------------+
+----------------------------------------------+  |    |       |
|   packet  |   frame  |  crypto   |   pnmap  |  | +-----+ +-----+
+----------------------------------------------+  | |     | |     |
|         input        |       output         |  |-| SMB | | NFS |..
+----------------------------------------------+  | |     | |     |
|                UDP tunnels                  |  | +-----+ +--+--+
+----------------------------------------------+  +------------|
```

User Data Archtechiture

```
+------+  +------+
| APP1 |  | APP2 | ...
+------+  +------+
  {send/recvmsg()}         {set/getsockopt()}
  [CMSG stream_info]       [SOCKOPT_KEY_UPDATE]
                           [SOCKOPT_CONNECTION_MIGRATION]
                           [SOCKOPT_STREAM_OPEN/RESET/STOP_SENDING]
                           [...]
          | ^                    | ^
Userspace | |                    | |
----------|-|--------------------|-|-----------------
Kernel    | |                    | |
          v |                    v |
+----------------------------------------------+
| socket (IPPRTOTO_QUIC) |      protocol      |<-+ {kernel_}
+----------------------------------------------+  | {send/recvmsg()}
| stream | connection_id | cong | path |timer |  | {set/getsockopt()}
+----------------------------------------------+  |
|   packet  |   frame  |  crypto   |   pnmap  |  | +-----+ +-----+
+----------------------------------------------+  | |     | |     |
|         input        |       output         |  |-| SMB | | NFS |..
+----------------------------------------------+  | |     | |     |
|                UDP tunnels                  |  | +-----+ +--+--+
+----------------------------------------------+  +------------|
```

Author's Address

Xin Long (EDITOR)
Red Hat
20 Deerfiled Drive
Ottawa ON
Canada
Email: lucien.xin@gmail.com