Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: 22 March 2026

L. Xin, Ed.
Red Hat
M. Buhl, Ed.
Technical University of Munich
M. Leitner, Ed.
Red Hat
18 September 2025

Sockets API Extensions for In-kernel QUIC Implementations
draft-lxin-quic-socket-apis-02

Abstract

This document describes a mapping of 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. In-kernel QUIC enables usage for both userspace applications and kernel consumers.

Table of Contents

1. Introduction 3
1.1. Conventions 4
2. Data Types 5
3. Interface 5
3.1. Basic Operation 6
3.1.1. socket() 6
3.1.2. bind() 7
3.1.3. listen() 7
3.1.4. accept() 8
3.1.5. connect() 9
3.1.6. close() 9
3.1.7. shutdown() 10
3.1.8. sendmsg() and recvmsg() 10
3.1.9. send(), recv(), read() and write() 11
3.1.10. setsockopt() and getsockopt() 13
3.1.11. getsockname() and getpeername() 13
3.2. Advanced Operation 14
3.2.1. quic_sendmsg() and quic_recvmsg() 14
3.2.2. quic_client/server_handshake() 15
3.2.3. quic_handshake() 16
4. Data Structures 17
4.1. The msghdr and cmsghdr Structures 17
4.2. Ancillary Data Considerations and Semantics 19
4.2.1. Multiple Items and Ordering 19
4.2.2. Accessing and Manipulating Ancillary Data 20
4.2.3. Control Message Buffer Sizing 20
4.3. QUIC msg_control Structures 21
4.3.1. Stream Information 21
4.3.2. Handshake Information 22
5. QUIC Events and Notifications 23
5.1. QUIC Notification Structure 23
5.1.1. QUIC_EVENT_STREAM_UPDATE 23
5.1.2. QUIC_EVENT_STREAM_MAX_DATA 25
5.1.3. QUIC_EVENT_STREAM_MAX_STREAM 25
5.1.4. QUIC_EVENT_CONNECTION_ID 26
5.1.5. QUIC_EVENT_CONNECTION_CLOSE 26
5.1.6. QUIC_EVENT_CONNECTION_MIGRATION 26

5.1.7. QUIC_EVENT_KEY_UPDATE 27
5.1.8. QUIC_EVENT_NEW_TOKEN 27
5.1.9. QUIC_EVENT_NEW_SESSION_TICKET 27
5.2. Notification Interest Options 27
5.2.1. QUIC_SOCKOPT_EVENT Option 28
6. Socket Options 28
6.1. Read/Write Options 28
6.1.1. QUIC_SOCKOPT_EVENT 28
6.1.2. QUIC_SOCKOPT_TRANSPORT_PARAM 28
6.1.3. QUIC_SOCKOPT_CONFIG 29
6.1.4. QUIC_SOCKOPT_CONNECTION_ID 32
6.1.5. QUIC_SOCKOPT_CONNECTION_CLOSE 32
6.1.6. QUIC_SOCKOPT_TOKEN 33
6.1.7. QUIC_SOCKOPT_ALPN 34
6.1.8. QUIC_SOCKOPT_SESSION_TICKET 34
6.1.9. QUIC_SOCKOPT_CRYPTO_SECRET 35
6.1.10. QUIC_SOCKOPT_TRANSPORT_PARAM_EXT 36
6.2. Read-Only Options 36
6.2.1. QUIC_SOCKOPT_STREAM_OPEN 36
6.2.2. QUIC_SOCKOPT_STREAM_PEELOFF 37
6.3. Write-Only Options 38
6.3.1. QUIC_SOCKOPT_STREAM_RESET 38
6.3.2. QUIC_SOCKOPT_STREAM_STOP_SENDING 38
6.3.3. QUIC_SOCKOPT_CONNECTION_MIGRATION 39
6.3.4. QUIC_SOCKOPT_KEY_UPDATE 39
7. Handshake Interface for Kernel Consumers 39
8. IANA Considerations 40
9. Security Considerations 40
10. References 41
10.1. Normative References 41
10.2. Informative References 41
Appendix A. Example For Multi-streaming Usage 41
Appendix B. Example For Stream Peeloff Usage 46
Appendix C. Example For defining custom client/ server_handshake() 50
Appendix D. Example For Session Consumption and 0-RTT transmission 55
Appendix E. Example For Kernel Consumers Architecture Design . . 62
Authors' Addresses 62

1. Introduction

The QUIC protocol, as defined in [RFC9000], offers a UDP-based, secure transport with flow-controlled streams for efficient communication, low-latency connection setup, and network path migration, ensuring confidentiality, integrity, and availability across various deployments.

In-kernel QUIC implementations will be able to offer several key advantages:

* Seamless Integration for Kernel Subsystems: Kernel subsystems such as SMB and NFS can operate over QUIC seamlessly after the handshake, leveraging the netlink APIs.

* Efficient ALPN Routing: It incorporates ALPN routing within the kernel, efficiently directing incoming requests to the appropriate applications across different processes based on ALPN.

* Performance Enhancements: By minimizing data duplication through zero-copy techniques such as sendfile(), and paving the way for crypto offloading in NICs, this implementation enhances performance and prepares for future optimizations.

* Standardized Socket APIs for QUIC: It standardizes the socket APIs for QUIC, covering essential operations like listen(), accept(), connect(), sendmsg(), recvmsg(), close(), get/setsockopt() and getsock/peername().

The socket APIs have 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 socket APIs. Based on [RFC6458], this document defines a method to map the existing socket APIs 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:

- socket()
- bind()
- listen()
- accept()
- quic_server_handshake()
- recvmsg()
- sendmsg()
- close()

It is similar to a TCP server, except for the quic_server_handshake() call, which handles the TLS message exchange to complete the handshake. See Section 3.2.2.

All TLS handshake messages carried in QUIC packets MUST be processed in userspace. When a Client Initial packet is received, it triggers accept() to create a new socket and return. However, the TLS handshake message contained in this packet will be processed by quic_server_handshake() via the newly created socket.

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

- socket()
- connect()
- quic_client_handshake()

- sendmsg()
- recvmsg()
- close()

It is similar to a TCP client, except for the quic_client_handshake() call, which handles the TLS message exchange to complete the handshake. See Section 3.2.2.

On the client side, connect() SHOULD NOT send any packets to the server. Instead, all TLS handshake messages are generated by the TLS library and sent in quic_client_handshake().

In the implementation, one QUIC socket represents a single QUIC connection and MAY manage multiple UDP sockets simultaneously to support connection migration or future multipath features. Conversely, a single lower-layer UDP socket MAY serve multiple QUIC sockets.

3.1. Basic Operation

3.1.1. socket()

Applications use socket() to create a socket descriptor to represent a 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 have a protocol number allocated by IANA. Similar to IPPROTO_MPTCP in Linux, IPPROTO_QUIC is simply a value used when opening a QUIC socket, and its value MAY vary 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 MUST use only IPv4 addresses, while PF_INET6 creates an endpoint that MAY use both IPv6 and IPv4 addresses. See Section 3.7 of [RFC3493].

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 with an endpoint. After the first bind() call, all subsequent calls will return an error. However, multiple applications MAY bind() to the same address and port, sharing the same lower UDP socket in the kernel.

The IP address part of addr MAY be specified as a wildcard (e.g., INADDR_ANY for IPv4 or IN6ADDR_ANY_INIT or in6addr_any for IPv6). If the IPv4 sin_port or IPv6 sin6_port is set to 0, the operating system will choose an ephemeral port for the endpoint.

If bind() is not explicitly called before connect() on the client, the system will automatically determine a valid source address based on the routing table and assign an ephemeral port to bind the socket during connect().

Completing the bind() process does not permit the QUIC endpoint to accept inbound QUIC connection requests on the server. This capability is only enabled after a listen() system call, as 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 connections.

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.

The implementation SHOULD allow the kernel to parse the ALPN from a Client Initial packet and direct the incoming request based on it to different listening sockets (binding to the same address and port). These sockets could belong to different user processes or kernel threads. The ALPNs for sockets are set via the ALPN socket option Section 6.1.7 before calling listen().

If no ALPNs are configured before calling listen(), the listening socket will only be capable of accepting client connections that do not specify any ALPN.

3.1.4. accept()

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

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 request on success and -1 in case of an error.

Note that the incoming Client Initial packet triggers the accept() call, and the TLS message carried by the Client Initial packet will be queued in the receive queue of the socket returned by accept(). This TLS message will then be received and processed by userspace through the newly returned socket, ensuring that the TLS handshake is completed in userspace.

3.1.5. connect()

Applications use connect() to perform routing and determine the appropriate source address and port to bind if bind() has not been called. Additionally, connect() initializes the connection ID and installs the initial keys necessary for encrypting handshake packets.

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 a connection.

The function prototype is

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

and the arguments are

* sd: The socket descriptor of the connection 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 a CLOSE frame to the peer. The close information MAY be set via the CONNECTION_CLOSE socket option Section 6.1.5 before calling close().

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 connection to be closed.

* how: Specifies the type of shutdown

  - SHUT_RD: Disables further receive operations. All incoming data and connection requests SHOULD be discard quietly.

  - SHUT_WR: Disables further send operations. It SHOULD send a CLOSE frame.

  - SHUT_RDWR: Similar to SHUT_WR.

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

Note that users MAY use SHUT_WR to send the CLOSE frame multiple times. The implementation MUST be capable of unblocking sendmsg(), recvmsg(), and accept() operations with SHUT_RDWR.

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

If the application does not provide enough buffer space to completely receive a data message in recvmsg(), MSG_EOR will not be set in msg_flags. Successive reads will consume more of the same message until the entire message has been delivered, and MSG_EOR will be set. This is particularly useful for reading datagram and event messages.

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

During Handshake, users SHOULD use sendmsg() and recvmsg() with Handshake msg_control Section 4.3.2 to send raw TLS messages to and receive from the kernel and to exchange TLS messages in userspace with the help of a third-party TLS library, such as GnuTLS. A pair of high-level APIs MAY be defined to wrap the handshake process in userspace. See Section 3.2.2.

Post Handshake, users SHOULD use sendmsg() and recvmsg() with Stream msg_control Section 4.3.1 to send data msgs to and receive from the kernel with stream_id and stream_flags. One pair of high-level APIs MAY be defined to wrap Stream msg_control. See Section 3.2.1.

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

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

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 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) cannot be used, the flags will operate as described in Section 4.1, but without the context provided by Stream or Handshake msg_control. While sending, the flags function as intended; however, when receiving, users will not be able to obtain any flags from the kernel.

send() can transmit data as datagram messages when MSG_DATAGRAM is set in the flags, and as stream messages on the most recently opened stream if MSG_DATAGRAM is not set. However, it cannot send handshake messages.

recv() can receive datagram, stream, or event messages, but it cannot determine the message type or stream ID without the appropriate flags and ancillary data from the kernel. It SHOULD return -1 with errno set to EINVAL when attempting to receive a handshake message.

Alternatively, applications can use read() and write() to transmit and receive data to and from a peer. These functions are similar to recv() and send() but offer less functionality, as they do not allow the use of a flags parameter.

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. These functions are particularly useful when connection migration occurs, and the corresponding event notifications are 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 buffer 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 quic_sendmsg() and quic_recvmsg() calls to transmit data to and receive data from its peer with stream_id and flags.

The function prototype of quic_sendmsg() is

```
ssize_t quic_sendmsg(int sd,
                     const void *msg,
                     size_t len,
                     int64_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.

* flags: function as stream_flags in Section 4.3.1 and msg_flags/ flags in Section 4.1. Any unknown flags passed into the kernel MUST be rejected with an error returned.

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

The function prototype of quic_recvmsg() is

```
ssize_t quic_recvmsg(int sd,
                     void *msg,
                     size_t len,
                     int64_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 receiving.

* flags: function as stream_flags in Section 4.3.1 and msg_flags/ flags in Section 4.1 used for passing flags to the kernel and then obtaining them from the kernel. Any unknown flags passed into the kernel MUST be rejected with an error returned.

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

These two functions wrap the sendmsg() and recvmsg() with Stream information msg_control and are important for using QUIC multiple streams. See an example in Appendix A

3.2.2. quic_client/server_handshake()

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

The function prototype of quic_server_handshake() is:

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

and the arguments are

* sd: The socket descriptor.

* pkey_file: Private key file for Certificate mode or pre-shared key file for PSK mode.

* cert_file: Certificate file for Certificate mode or null for PSK mode.

* alpns: ALPNs supported and split by ',’.

The function prototype of quic_client_handshake() is:

```
int quic_client_handshake(int sd,
                          const char *pkey_file,
                          const char *hostname,
                          const char *alpns);
```

and the arguments are

* sd: The socket descriptor.

* psk_file: Pre-shared key file for PSK mode.

* hostname: Server name for Certificate mode.

* alpns: ALPNs supported and split by ',’.

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

3.2.3. quic_handshake()

Using quic_handshake() allows an application to have greater control over the configuration of the handshake session.

The function prototype is

```
int quic_handshake(void *session);
```

and the arguments are

* session: A TLS session, which is represented differently across various TLS libraries, such as gnutls_session_t in GnuTLS or SSL * in OpenSSL.

With the session argument, users can configure additional parameters at TLS level and define custom client_handshake() and server_handshake() functions. An example is provided in Appendix C.

In the future, quic_handshake() MAY be considered for integration into these TLS libraries to provide comprehensive support for QUIC stack.

Here are some guidelines for handling TLS and QUIC communications between kernel and userspace when implement quic_handshake():

* Handling Raw TLS Messages:

  The implementation SHOULD utilize sendmsg() and recvmsg() with Handshake msg_control Section 4.3.2 to send and receive raw TLS messages between the kernel and userspace. These messages should be processed in userspace using a TLS library, such as GnuTLS.

* Processing the TLS QUIC Transport Parameters Extension:

  The implementation SHOULD retrieve the local TLS QUIC transport parameters extension from the kernel using the TRANSPORT_PARAM_EXT socket option Section 6.1.10 for building TLS messages. Additionally, remote handshake parameters should be set in the kernel using the same socket option for constructing QUIC packets.

* Setting Secrets for Different Crypto Levels:

  The implementation SHOULD set secrets for various crypto levels using the CRYPTO_SECRET socket option Section 6.1.9.

4. Data Structures

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

4.1. The msghdr and cmsghdr Structures

The msghdr structure used in sendmsg() and recvmsg() calls, along with the ancillary data it carries, is crucial for applications to set and retrieve 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:

* The flags parameter in sendmsg() can be set to:

  - MSG_MORE: Indicates that data will be held until the next data is sent without this flag.

  - MSG_DONTWAIT: Prevents blocking if there is no send buffer.

  - MSG_DATAGRAM: Sends data as an unreliable datagram.

  Additionally, the flags can also be set to the values in stream_flags on the send side Section 4.3.1 if Stream msg_control is not being used. In this case, the most recently opened stream will be used for sending data.

* msg_flags of msghdr passed to the kernel is ignored.

On receive side:

* The flags parameter in recvmsg() might be set to:

  - MSG_DONTWAIT: Prevents blocking if there is no data in recv buffer.

* msg_flags of msghdr returned from the kernel might be set to:

  - MSG_EOR: Indicates that the received data is read completely.

  - MSG_DATAGRAM: Indicates that the received data is a datagram.

  - MSG_NOTIFICATION: Indicates that the received data is a notification message.

  - These flags might also be set to the values in stream_flags on the receive side Section 4.3.1 if Stream msg_control is not being used. In this case, the stream id for received data is invisible to user space.

  Additionally, the flags might also be set to the values in stream_flags on the receive side Section 4.3.1 if Stream msg_control is not being used. In this case, the stream ID for received data is not visible to users.

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 QUIC-specific items, non-QUIC items (such as IP-level items), or both. The ordering of ancillary data items, whether QUIC-related or from another protocol, is implementation-dependent and not significant. Therefore, applications MUST NOT rely on any specific ordering.

QUIC_STREAM_INFO and QUIC_HANDSHAKE_INFO type ancillary data always correspond to the data in the msghdr’s msg_iov member. Only one such type of ancillary data is allowed per 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 control message (cmsg) specifies QUIC stream options for sendmsg() and describes QUIC stream information about a received message via recvmsg(). It uses struct quic_stream_info

```
struct quic_stream_info {
  int64_t stream_id;
  uint32_t stream_flags;
};
```

On send side:

* stream_id:

  - -1:

    o If MSG_STREAM_NEW is set: Open the next bidirectional stream and uses it for sending data.

    o If both MSG_STREAM_NEW and MSG_STREAM_UNI are set: Opens the next unidirectional stream and uses it for sending data.

    o Otherwise: Use the latest opened stream for sending data.

- !-1: The specified stream ID is used with the first 2 bits:

  - QUIC_STREAM_TYPE_SERVER_MASK(0x1): Indicates if it is a server-side stream.

  - QUIC_STREAM_TYPE_UNI_MASK(0x2): Indicates if it is a unidirectional stream.

* stream_flags:

  - MSG_STREAM_NEW: Open a stream and send the first data.

  - MSG_STREAM_FIN: Send the last data and close the stream.

  - MSG_STREAM_UNI: Open the next unidirectional stream.

  - MSG_STREAM_DONTWAIT: Open the stream without blocking.

  - MSG_STREAM_SNDBLOCK: Send streams blocked when no capacity.

On receive side:

* stream_id: Identifies the stream to which the received data belongs.

* stream_flags:

  - MSG_STREAM_FIN: Indicates that the data received is the last one for this stream.

This cmsg is specifically used for sending user stream data, including early or 0-RTT data. When sending user unreliable datagrams, this cmsg SHOULD NOT be set.

4.3.2. Handshake Information

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

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

crypto_level: Specifies the level of data:

* QUIC_CRYPTO_INITIAL: Initial level data.

* QUIC_CRYPTO_HANDSHAKE: Handshake level data.

This cmsg is used only during the handshake process.

5. QUIC Events and Notifications

A QUIC application MAY need to understand and process events and errors that occur within the QUIC stack, such as stream updates, max_stream changes, connection close, connection migration, key updates and new tokens. These events are categorized under the quic_event_type enum:

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

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 in Section 4.3.1.

The first byte of the received data indicates the type of the event, corresponding to one of the values in the quic_event_type enum. The subsequent bytes contain the content of the event, meaning the length of the content is the total data length minus one byte. To manage and enable these events, refer to the EVENT socket option Section 5.2.1.

5.1. QUIC Notification Structure

5.1.1. QUIC_EVENT_STREAM_UPDATE

Only notifications with one of the following states are delivered to userspace:

* QUIC_STREAM_SEND_STATE_RECVD

  An update is delivered when all data on the stream has been acknowledged. This indicates that the peer has confirmed receipt of all sent data for this stream.

* QUIC_STREAM_SEND_STATE_RESET_SENT

  An update is delivered only if a STOP_SENDING frame is received from the peer and a STREAM_RESET frame is triggered to send out. The STOP_SENDING frame MAY be sent by the peer via the STREAM_STOP_SENDING socket option Section 6.3.2.

* QUIC_STREAM_SEND_STATE_RESET_RECVD

  An update is delivered when a STREAM_RESET frame has been received and acknowledged by the peer. The STREAM_RESET frame MAY be sent via the socket option STREAM_RESET Section 6.3.1.

* QUIC_STREAM_RECV_STATE_RECV

  An update is delivered only when the last fragment of data has not yet arrived. This event is sent to inform the application that there is pending data for the stream.

* QUIC_STREAM_RECV_STATE_SIZE_KNOWN

  An update is delivered only if data arrives out of order. This indicates that the size of the data is known, even though the fragments are not in sequential order.

* QUIC_STREAM_RECV_STATE_RECVD

  An update is delivered when all data on the stream has been fully received. This signifies that the application has received the complete data for the stream.

* QUIC_STREAM_RECV_STATE_RESET_RECVD

  An update is delivered when a STREAM_RESET frame is received. This indicates that the peer has reset the stream, and further data SHOULD NOT be expected.

Data format in the event

```
struct quic_stream_update {
  int64_t id;
  uint32_t state;
  uint32_t errcode;
  uint64_t finalsz;
};
```

id: The stream ID.

state: The new stream state. All valid states are listed above.

errcode: Error code for the application protocol. It is used for the RESET_SENT or RESET_RECVD state update on send side, and for the RESET_RECVD update on receive side.

finalsz: The final size of the stream. It is used for the SIZE_KNOWN, RESET_RECVD, or RECVD state updates on receive side.

5.1.2. QUIC_EVENT_STREAM_MAX_DATA

This notification is delivered when a MAX_STREAM_DATA frame is received. If a stream is blocked, a non-blocking sendmsg() call will return ENOSPC. This event notifies the application when additional send space becomes available for a stream, allowing applications to adjust stream scheduling accordingly.

Data format in the event

```
struct quic_stream_max_data {
  int64_t id;
  uint64_t max_data;
};
```

id: The stream ID.

max_data: The updated maximum amount of data that can be sent on the stream.

5.1.3. QUIC_EVENT_STREAM_MAX_STREAM

This notification is delivered when a MAX_STREAMS frame is received. It is particularly useful when using MSG_STREAM_DONTWAIT stream_flags to open a stream via the STREAM_OPEN socket option Section 6.2.1 whose ID exceeds the current maximum stream count. After receiving this notification, the application SHOULD attempt to open the stream again.

Data format in the event

```
uint64_t max_stream;
```

It indicates the maximum stream limit for a specific stream byte. The stream type is encoded in the first 2 bits, and the maximum stream limit is calculated by shifting max_stream right by 2 bits.

5.1.4. QUIC_EVENT_CONNECTION_ID

This notification is delivered when any source or destination connection IDs are retired. This usually occurs during connection migration or when managing connection IDs via the CONNECTION_ID socket option Section 6.1.4.

Data format in the event

```
struct quic_connection_id_info {
  uint8_t  dest;
  uint32_t active;
  uint32_t prior_to;
};
```

* dest: Indicates whether to operate on destination connection IDs.

* active: The number of the connection ID in use.

* prior_to: The lowest connection ID number.

5.1.5. QUIC_EVENT_CONNECTION_CLOSE

This notification is delivered when a CLOSE frame is received from the peer. The peer MAY set the close information via the CONNECTION_CLOSE socket option Section 6.1.5 before calling close().

Data format in the event

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

errcode: Error code for the application protocol.

phrase: Optional string for additional details.

frame: Frame type that caused the closure.

5.1.6. QUIC_EVENT_CONNECTION_MIGRATION

This notification is delivered when either side successfully changes its source address using the CONNECTION_MIGRATION socket option Section 6.3.3, or when the destination address is changed by the peer’s connection migration.

Data format in the event

uint8_t local_migration;

It indicates whether the migration was local or initiated by the peer. After receiving this notification, the new address can be retrieved using getsockname() for the local address or getpeername() for the peer's address.

5.1.7. QUIC_EVENT_KEY_UPDATE

This notification is delivered when both sides have successfully updated to the new key phase after a key update via the KEY_UPDATE socket option Section 6.3.4 on either side.

Data format in the event

uint8_t key_update_phase;

It indicates which key phase is currently in use.

5.1.8. QUIC_EVENT_NEW_TOKEN

This notification is delivered whenever a NEW_TOKEN frame is received from the peer. Since the handshake occurs in userspace, you can send these tokens using the TOKEN socket option Section 6.1.6.

Data format in the event

uint8_t token[];

It carries the token data.

5.1.9. QUIC_EVENT_NEW_SESSION_TICKET

This notification is delivered whenever a NEW_SESSION_TICKET message carried in crypto frame is received from the peer.

Data format in the event

uint8_t ticket[];

It carries the data of the TLS session ticket message.

5.2. Notification Interest Options

5.2.1. QUIC_SOCKOPT_EVENT Option

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

The optval type is

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

type: Specifies the event type, as defined in Section 5.1.

on: Indicates whether the event is enabled or disabled:

* 0: disable.

* !0: enable.

By default, all events are disabled.

6. Socket Options

The following subsection outlines various SOL_QUIC level socket options. A getsockopt() call MUST be used to retrieve any readable options, while a setsockopt() call MUST be used to configure any writable options. For further details on specific options and their corresponding structures, see the rest of this section.

6.1. Read/Write Options

6.1.1. QUIC_SOCKOPT_EVENT

This socket option is used to set a specific notification option. For a detailed description of this option and its usage, please refer to Section 5.2.1.

6.1.2. QUIC_SOCKOPT_TRANSPORT_PARAM

This option is used to configure QUIC transport parameters.

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 */
  uint8_t  active_connection_id_limit;       /* 7 */
  uint8_t  ack_delay_exponent;               /* 3 */
  uint16_t max_datagram_frame_size;          /* 0 */
  uint16_t max_udp_payload_size;             /* 65527 */
  uint32_t max_idle_timeout;                 /* 30000000 us */
  uint32_t max_ack_delay;                    /* 25000 */
  uint16_t max_streams_bidi;                 /* 100 */
  uint16_t max_streams_uni;                  /* 100 */
  uint64_t max_data;                         /* 65536 * 32 */
  uint64_t max_stream_data_bidi_local;       /* 65536 * 4 */
  uint64_t max_stream_data_bidi_remote;      /* 65536 * 4 */
  uint64_t max_stream_data_uni;              /* 65536 * 4 */
  uint64_t reserved;
};
```

These parameters and described in [RFC9000] and their default values are specified in the struct code.

The remote member allows users to set remote transport parameters. When used in conjunction with session resumption ticket, it enables the configuration of remote transport parameters from the previous connection. This configuration is crucial for sending 0-RTT data efficiently.

6.1.3. QUIC_SOCKOPT_CONFIG

This option is used to configure various settings for QUIC connection, and also includes some handshake-specific options for kernel consumers.

The optval type is

```
struct quic_config {
  uint32_t version;
  uint32_t plpmtud_probe_interval;
  uint32_t initial_smoothed_rtt;
  uint32_t payload_cipher_type;
  uint8_t  congestion_control_algo;
  uint8_t  validate_peer_address;
  uint8_t  stream_data_nodelay;
  uint8_t  receive_session_ticket;
  uint8_t  certificate_request;
  uint8_t  reserved[3];
};
```

* version:

  QUIC version, options include:

  - QUIC_VERSION_V1 (by default)

  - QUIC_VERSION_V2

* plpmtud_probe_interval (in usec):

  The probe interval of Packetization Layer Path MTU Discovery.
  options include:

  - 0: disabled (by default)

  - !0: at least QUIC_MIN_PROBE_TIMEOUT (5000000)

* initial_smoothed_rtt (in usec):

  The initial smoothed rtt, options include:

  - 333000 (by default)

  - at least QUIC_RTO_MIN (100000)

  - less than QUIC_RTO_MAX (6000000)

* congestion_control_algo:

  Congestion control algorithm, options MAY include:

  - NEW_RENO (by default)

  - CUBIC

- BBR

* payload_cipher_type:

  This option is for kernel consumers only, allowing them to inform userspace handshake of the preferred cipher type, options include:

  - 0: any type (by default)

  - AES_GCM_128

  - AES_GCM_256

  - AES_CCM_128

  - CHACHA20_POLY1305

* validate_peer_address:

  This option is Server-side only, and if it is enabled, server will send a retry packet to client upon receiving the first handshake request. This serves to validate client's IP address, ensuring it is legitimate and reachable before proceeding with the rest of the handshake.

  - 0: disabled (by default)

  - !0: enabled

* receive_session_ticket (in sec):

  This option is for Client-side kernel consumers only, enabling userspace handshake to receive session ticket, either via event NEW_SESSION_TICKET Section 5.1.9 or the socket option SESSION_TICKET Section 6.1.8 and then set back to kernel via the socket option SESSION_TICKET.

  - 0: disabled (by default)

  - !0: maximum time (in sec) to wait

* certificate_request:

  This option is Server-side kernel consumers only, instructing userspace handshake whether to request a certificate from client, options include:

  - 0: IGNORE (by default)

- 1: REQUEST

- 2: REQUIRE

* stream_data_nodelay:

  This option is to disable the Nagle algorithm, options include:

  - 0: Enable the Nagle algorithm (by default)

  - !0: disable the Nagle algorithm

6.1.4. QUIC_SOCKOPT_CONNECTION_ID

This option is used to get or set the source and destination connection IDs, including dest, active and prior_to. Along with the active_connection_id_limit in the transport parameters, it helps determine the range of available connection IDs.

The optval type is

```
struct quic_connection_id_info {
  uint8_t  dest;
  uint32_t active;
  uint32_t prior_to;
};
```

* dest: Indicates whether to operate on destination connection IDs.

* active: The number of the connection ID in use.

* prior_to: The lowest connection ID number.

The active field is used to switch the connection ID in use. The prior_to field, for source connection IDs, specifies prior to which ID will be retired by sending NEW_CONNECTION_ID frames; for destination connection IDs, it indicates prior to which ID issued by the peer will no longer be used and should be retired by sending RETIRE_CONNECTION_ID frames.

6.1.5. QUIC_SOCKOPT_CONNECTION_CLOSE

This option is used to get or set the close context, which includes errcode, phrase, and frame. On the closing side, set it before calling close() to tell the peer the closing information. On the side being closed, get it to receive the peer’s closing information.

* On the closing side: Set this option before calling close() to communicate the closing information to the peer.

* On the receiving side: Get this option to retrieve the closing information from the peer.

The optval type is

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

* errcode: Error code for the application protocol. Defaults to 0.

* phrase: Optional string for additional details. Defaults to null.

* frame: Frame type that caused the closure. Defaults to 0.

6.1.6. QUIC_SOCKOPT_TOKEN

This option is used to manage tokens for address verification in QUIC connections. It behaves differently depending on whether it’s being used on the client side or the server side.

* Client-Side Usage:

  The client uses this option to set a token provided by the peer server. This token is typically issued by the server during the last connection and is used for address verification in subsequent connections.

  The token can be obtained from the server during the previous connection, either via getsockopt() with this option or from event NEW_TOKEN Section 5.1.8.

  The optval type is

  ```
  uint8_t *opt;
  ```

* Server-Side Usage:

  The server uses this option to issue a new token to the client. This token will be used by the client for address verification in the next connection.

  The optval type is null.

The default value in the socket is null.

6.1.7. QUIC_SOCKOPT_ALPN

This option is used on listening sockets for kernel ALPN routing. On regular sockets, it allows kernel consumers to communicate ALPN identifiers with userspace handshake.

* On regular sockets:

  It sets the desired ALPNs before the kernel consumers send handshake requests to userspace. Multiple ALPNs can be specified separated by commas (e.g., "smbd,h3,ksmbd"). The userspace handshake SHOULD return the selected ALPN to the kernel via this socket option.

* On listening socket:

  If kernel supports ALPN routing, this feature directs incoming requests to the appropriate application based on ALPNs. The ALPNs MUST be set on the socket before calling listen() to enable this functionality.

The optval type is

```
char *alpn;
```

The default value in the socket is null.

6.1.8. QUIC_SOCKOPT_SESSION_TICKET

This option is used on listening sockets to retrieve the key for enabling the session tickets on server. On regular sockets, it can be used to receive the session ticket message on client. It is also used by client-side kernel consumers to communicate session data with userspace handshake.

* For the userspace handshake:

  On the server side, a userspace handshake requires a key to enable the session ticket option, this key SHOULD be retrieved via this socket option.

  On the client side, a userspace handshake can receive NEW_SESSION_TICKET messages via this socket option from the kernel to generate session data while processing this message.

  See an example in Appendix D.

* For kernel consumers:

  After userspace handshake handles NEW_SESSION_TICKET messages, it MUST return session data to kernel via this socket option, and kernel consumers can get it via this socket option for session resumption in future connections.

  During session resumption, kernel consumers will use this socket option to inform userspace handshake about session data.

The optval type is

```
uint8_t *opt;
```

The default value in the socket is null.

6.1.9. QUIC_SOCKOPT_CRYPTO_SECRET

This socket option is used to set cryptographic secrets derived from userspace to the socket in the kernel during the QUIC handshake process.

The optval type is

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

* level: Specifies the QUIC cryptographic level for which the secret is intended.
  - QUIC_CRYPTO_APP: Application level
  - QUIC_CRYPTO_HANDSHAKE: Handshake level
  - QUIC_CRYPTO_EARLY: Early or 0-RTT level
* send: Indicates the direction of the secret.
  - 0: Set secret for receiving
  - !0: Set secret for sending
* type: Specifies the encryption algorithm used.

- AES_GCM_128
- AES_GCM_256
- AES_CCM_128
- CHACHA20_POLY1305

* secret: The cryptographic key material to be used. The length of this array depends on the type and SHOULD be filled accordingly in kernel.

This option is only relevant during the handshake phase.

6.1.10. QUIC_SOCKOPT_TRANSPORT_PARAM_EXT

This socket option is used to retrieve or set the QUIC Transport Parameters Extension, which is essential for building TLS messages and handling extended QUIC transport parameters during the handshake process.

* Get Operation:

  When retrieving parameters via getsockopt, this option generates the QUIC Transport Parameters Extension based on the local transport parameters configured in the kernel. The retrieved extension helps userspace applications build the appropriate TLS messages.

* Set Operation:

  When setting parameters via setsockopt, this option updates the kernel with the QUIC Transport Parameters Extension received from the peer's TLS message. This ensures that the remote transport parameters are correctly configured in the kernel.

uint8_t *opt;

This option is used exclusively during the handshake phase.

6.2. Read-Only Options

6.2.1. QUIC_SOCKOPT_STREAM_OPEN

This socket option is used to open a new QUIC stream. It allows applications to initiate streams for data transmission within a QUIC connection.

The optval type is

```
struct quic_stream_info {
  int64_t stream_id;
  uint32_t stream_flags;
};
```

* stream_id: Specifies the stream ID for the new stream. It can be set to:

  - >= 0: Open a stream with the specific stream_id provided.

  - -1: Open the next available stream. The assigned stream ID will be returned to the user via the stream_id field after the operation.

* stream_flags: Specifies flags for stream creation. It can be set to:

  - QUIC_STREAM_UNI: Open the next unidirectional stream.

  - QUIC_STREAM_DONTWAIT: Open the stream without blocking; this allows the request to be processed asynchronously.

6.2.2. QUIC_SOCKOPT_STREAM_PEELOFF

This socket option is used to peel off a QUIC stream. It allows an application to detach an individual QUIC stream from its parent connection and obtain a dedicated socket for that stream.

The optval type is

```
struct quic_stream_peeloff {
  int64_t stream_id;
  uint32_t flags;
  int sd;
};
```

* stream_id: Specifies the ID of the stream to be peeled off.

* flags: Specifies flags for file descriptor creation (e.g., O_CLOEXEC).

* sd: The new file descriptor returned to the application.

With this peeled-off socket, applications can operate on the stream using the standard socket APIs, such as sendmsg(), recvmsg() (without requiring msg_control for stream IDs as in Section 4.3.1), or send(), recv(), read(), and write() in a way similar to TCP.

The implementation SHOULD also support the following interfaces:

* poll(): Return the mask based on both the stream state and the parent socket state.

* close(): If permitted by the stream state and the parent socket state, send a FIN on a sending stream and a STOP_SENDING frame on a receiving stream.

* shutdown(): If permitted by the stream state and the parent socket state, send a FIN on a sending stream with SHUT_RD, a STOP_SENDING frame on a receiving stream with SHUT_WR, or both with SHUT_RDWR.

An example is provided in Appendix B.

6.3. Write-Only Options

6.3.1. QUIC_SOCKOPT_STREAM_RESET

This socket option is used to reset a specific QUIC stream. Resetting a stream indicates that the endpoint will no longer guarantee the delivery of data associated with that stream.

The optval type is

```
struct quic_errinfo {
  int64_t stream_id;
  uint32_t errcode;
};
```

* stream_id: Specifies the ID of the stream to be reset.

* errcode: An application protocol error code indicating the reason for the stream reset.

6.3.2. QUIC_SOCKOPT_STREAM_STOP_SENDING

This socket option is used to request that the peer stop sending data on a specified QUIC stream. This is typically used to signal that the local endpoint is no longer interested in receiving further data on that stream.

The optval type is

```
struct quic_errinfo {
  int64_t stream_id;
  uint32_t errcode;
};
```

* stream_id: Specifies the ID of the stream to be reset.

* errcode: An application protocol error code indicating the reason for the stream stop_sending.

6.3.3. QUIC_SOCKOPT_CONNECTION_MIGRATION

This socket option is used to initiate a connection migration, which allows the QUIC connection to switch to a new address. It can also be used on the server side to set the preferred address transport parameter before the handshake.

The optval type is

struct sockaddr_in(6);

6.3.4. QUIC_SOCKOPT_KEY_UPDATE

This socket option is used to initiate a key update or rekeying process for the QUIC connection.

The optval type is null.

7. Handshake Interface for Kernel Consumers

In-kernel QUIC is not only beneficial for userspace applications such as HTTP/3, but it also naturally supports kernel consumers like SMB and NFS. For these kernel consumers, a userspace service is needed to handle handshake requests from the kernel. The communication between userspace and kernel can vary across different operating systems. Generally, there are two methods to handle this interaction:

* Attaching and Sending Socket Descriptors:

  - Kernel consumers attach a socket file descriptor (sockfd) to a connecting or accepting kernel socket.

  - This sockfd is then sent to userspace, where the userspace service performs the handshake as if it were handling a userspace socket.

- Once the handshake is completed, the service sends the sockfd back to the kernel. The Kernel then detaches the sockfd from the kernel socket, which is now in an established state.

* Sending Raw TLS Messages with a session ID:

- Kernel consumers send raw TLS messages to userspace, identified by a session ID.

- The userspace service creates a TLS session associated with this ID and exchanges TLS messages with the session.

- The service then sends the response TLS messages back to the kernel. This method requires maintaining the TLS session identified by the ID in userspace until the handshake is complete after multiple exchanges.

On Linux, the tlshd service utilizes the first method via Netlink.

For further details on the infrastructure design in Linux, refer to Appendix E

8. IANA Considerations

No actions from IANA are required.

9. Security Considerations

The socket receive buffer SHOULD be adjusted according to the max_data parameter from the struct quic_transport_param. The implementation SHOULD update the socket receive buffer whenever the local transport parameter max_data changes. Using a socket receive buffer smaller than the local transport parameter max_data MAY impair performance.

Similarly, the socket send buffer SHOULD be adjusted based on the peer’s max_data transport parameter to optimize performance rather than setting it manually.

Additionally, the size of the optval for the following socket options SHOULD be limited to avoid excessive memory allocation:

* QUIC_SOCKOPT_ALPN

* QUIC_SOCKOPT_TOKEN

* QUIC_SOCKOPT_SESSION_TICKET

* QUIC_SOCKOPT_CONNECTION_CLOSE

10. References

10.1. 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>.

10.2. 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 demonstrates how to use quic_sendmsg() and quic_recvmsg() to send and receive messages across multiple QUIC streams simultaneously.

```
#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> [ALPN]\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, argv[4]))
        return -1;
```

```
       /* Open stream 0 and send first data on stream 0 with
        * MSG_STREAM_NEW, or call getsockopt(QUIC_SOCKOPT_STREAM_OPEN)
        * to open a stream and then send data with out MSG_STREAM_NEW
        * needed
        */
       strcpy(msg, "hello ");
       sid = 0;
       flags = MSG_STREAM_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 = MSG_STREAM_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 and then close stream
        * 2 with MSG_STREAM_FIN
        */
       strcpy(msg, "server stream 2!");
       sid = 2;
       flags = MSG_STREAM_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 = MSG_STREAM_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));
        flags = 0;
        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> [ALPN]\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], argv[6]))
        return -1;

    while (!(stream[0].flags & MSG_STREAM_FIN) ||
           !(stream[1].flags & MSG_STREAM_FIN)) {
        flags = 0;
        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 = MSG_STREAM_NEW | MSG_STREAM_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 Stream Peeloff Usage

This example shows how to use the QUIC_SOCKOPT_STREAM_PEELOFF socket option to create a new socket bound to a specific QUIC stream, which can then be managed using the standard socket APIs in the same way as TCP.

```
#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 int do_client(int argc, char *argv[])
{
    struct quic_stream_peeloff pinfo = {};
    struct quic_stream_info sinfo = {};
```

```
struct sockaddr_in ra = {};
int ret, sockfd, strmfd;
unsigned int optlen;
const char *rc;
char msg[50];

if (argc < 3) {
    printf("%s client <PEER ADDR> <PEER PORT> [ALPN]\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, argv[4]))
    return -1;

/* open a unidirectional stream */
optlen = sizeof(sinfo);
sinfo.stream_id = -1;
sinfo.stream_flags = MSG_STREAM_UNI;
ret = getsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_STREAM_OPEN,
                 &sinfo, &optlen);
if (ret) {
    printf("getsockopt stream_open failed\n");
    return -1;
}

/* peel off the stream */
optlen = sizeof(pinfo);
pinfo.stream_id = sinfo.stream_id;
strmfd = getsockopt(sockfd, SOL_QUIC,
                    QUIC_SOCKOPT_STREAM_PEELOFF,
                    &pinfo, &optlen);
if (strmfd < 0) {
```

```
            printf("getsockopt stream_peeloff failed %d\n", strmfd);
            return -1;
        }

        /* send data on the peeled off stream */
        strcpy(msg, "hello quic server!");
        ret = send(strmfd, msg, strlen(msg), 0);
        if (ret == -1) {
            printf("send error %d %d\n", ret, errno);
            return -1;
        }
        printf("send '%s' on stream %d\n", msg, (int)pinfo.stream_id);
        close(strmfd);

        recv(sockfd, NULL, 0, 0);
        close(sockfd);
        return 0;
    }

    static int do_server(int argc, char *argv[])
    {
        struct quic_stream_peeloff pinfo = {};
        unsigned int addrlen, flags, optlen;
        int listenfd, sockfd, strmfd, ret;
        struct quic_event_option event;
        struct sockaddr_in sa = {};
        union quic_event *ev;
        const char *rc;
        char msg[50];

        if (argc < 5) {
            printf("%s server <LOCAL ADDR> <LOCAL PORT>"
                   " <PRIVATE_KEY_FILE> <CERTIFICATE_FILE> [ALPN]\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], argv[6]))
        return -1;

    /* enable stream update event */
    event.type = QUIC_EVENT_STREAM_UPDATE;
    event.on = 1;
    ret = setsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_EVENT, &event,
                     sizeof(event));
    if (ret == -1) {
        printf("socket setsockopt event error %d\n", errno);
        return -1;
    }

    while (1) {
        /* wait for stream update event for new recv stream */
        flags = 0;
        memset(msg, 0, sizeof(msg));
        ret = quic_recvmsg(sockfd, msg, sizeof(msg) - 1, NULL,
                           &flags);
        if (ret == -1) {
            printf("recv error %d %d\n", ret, errno);
            return 1;
        }
        if (!(flags & MSG_NOTIFICATION) ||
            msg[0] != QUIC_EVENT_STREAM_UPDATE)
            continue;
        ev = (union quic_event *)&msg[1];
        if (ev->update.state != QUIC_STREAM_RECV_STATE_RECV)
            continue;

        /* peel off the stream */
        optlen = sizeof(pinfo);
        pinfo.stream_id = ev->update.id;
        strmfd = getsockopt(sockfd, SOL_QUIC,
                            QUIC_SOCKOPT_STREAM_PEELOFF,
                            &pinfo, &optlen);
```

```
            if (strmfd < 0) {
                printf("getsockopt stream_peeloff failed %d\n", ret);
                return -1;
            }

            /* read data from the peeled off stream */
            memset(msg, 0, sizeof(msg));
            ret = recv(strmfd, msg, sizeof(msg) - 1, 0);
            if (ret == -1) {
                printf("recv error %d %d\n", ret, errno);
                return 1;
            }
            printf("recv '%s' on stream %d\n", msg,
                   (int)pinfo.stream_id);
            close(strmfd);
            break;
        }

        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 defining custom client/server_handshake()

This example demonstrates how to use quic_handshake() to define client_handshake() and server_handshake() for session resumption using GnuTLS.

```
#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 int quic_session_get_data(gnutls_session_t session,
                                 void *data, size_t *size)
{
    int ret, sockfd = gnutls_transport_get_int(session);
    unsigned int len = *size;

    if (getsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_SESSION_TICKET,
                   data, &len))
        return -errno;
    if (!len) {
        *size = 0;
        return 0;
    }

    ret = quic_handshake_process(session, QUIC_CRYPTO_APP,
                                 data, len);
    if (ret)
        return ret;
    return gnutls_session_get_data(session, data, size);
}

static int quic_session_set_data(gnutls_session_t session,
                                 const void *data, size_t size)
{
    return gnutls_session_set_data(session, data, size);
}

static int quic_session_get_alpn(gnutls_session_t session,
                                 void *alpn, size_t *size)
{
    gnutls_datum_t alpn_data;
    int ret;

    ret = gnutls_alpn_get_selected_protocol(session, &alpn_data);
    if (ret)
        return ret;

    if (*size < alpn_data.size)
```

```
            return -EINVAL;

        memcpy(alpn, alpn_data.data, alpn_data.size);
        *size = alpn_data.size;
        return 0;
    }

    static int quic_session_set_alpn(gnutls_session_t session,
                                     const void *alpns, size_t size)
    {
        gnutls_datum_t alpn_data[5];
        char *s, data[64] = {};
        int count = 0;

        if (size >= 64)
            return -EINVAL;

        memcpy(data, alpns, size);
        s = strtok(data, ",");
        while (s) {
            while (*s == ' ')
                s++;
            alpn_data[count].data = (unsigned char *)s;
            alpn_data[count].size = strlen(s);
            count++;
            s = strtok(NULL, ",");
        }

        return gnutls_alpn_set_protocols(session, alpn_data, count,
                                         GNUTLS_ALPN_MANDATORY);
    }

    static int client_handshake(int sockfd, const char *alpns,
                                const uint8_t *ticket_in,
                                size_t ticket_in_len,
                                uint8_t *ticket_out,
                                size_t *ticket_out_len)
    {
        gnutls_certificate_credentials_t cred;
        gnutls_session_t session;
        size_t alpn_len;
        char alpn[64];
        int ret;

        ret = gnutls_certificate_allocate_credentials(&cred);
        if (ret)
            goto err;
        ret = gnutls_certificate_set_x509_system_trust(cred);
```

```
    if (ret < 0)
        goto err_cred;

    ret = gnutls_init(&session, GNUTLS_CLIENT |
                                    GNUTLS_ENABLE_EARLY_DATA |
                                    GNUTLS_NO_END_OF_EARLY_DATA);
    if (ret)
        goto err_cred;
    ret = gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                                 cred);
    if (ret)
        goto err_session;

    ret = gnutls_priority_set_direct(session, QUIC_PRIORITY, NULL);
    if (ret)
        goto err_session;

    if (alpns) {
        ret = quic_session_set_alpn(session, alpns, strlen(alpns));
        if (ret)
            goto err_session;
    }

    if (host) {
        ret = gnutls_server_name_set(session, GNUTLS_NAME_DNS, host,
                                     strlen(host));
        if (ret)
            goto err_session;
    }

    gnutls_transport_set_int(session, sockfd);

    if (ticket_in) {
        ret = quic_session_set_data(session, ticket_in,
                                    ticket_in_len);
        if (ret)
            goto err_session;
    }

    ret = quic_handshake(session);
    if (ret)
        goto err_session;

    if (alpns) {
        alpn_len = sizeof(alpn);
        ret = quic_session_get_alpn(session, alpn, &alpn_len);
        if (ret)
            goto err_session;
```

```
        }

        if (ticket_out) {
            sleep(1);
            ret = quic_session_get_data(session, ticket_out,
                                        ticket_out_len);
            if (ret)
                goto err_session;
        }

    err_session:
        gnutls_deinit(session);
    err_cred:
        gnutls_certificate_free_credentials(cred);
    err:
        return ret;
    }

    static int server_handshake(int sockfd, const char *pkey,
                                const char *cert, const char *alpns,
                                uint8_t *key, unsigned int keylen)
    {
        gnutls_certificate_credentials_t cred;
        gnutls_datum_t skey = {key, keylen};
        gnutls_session_t session;
        size_t alpn_len;
        char alpn[64];
        int ret;

        ret = gnutls_certificate_allocate_credentials(&cred);
        if (ret)
            goto err;
        ret = gnutls_certificate_set_x509_system_trust(cred);
        if (ret < 0)
            goto err_cred;
        ret = gnutls_certificate_set_x509_key_file(cred, cert, pkey,
                                                   GNUTLS_X509_FMT_PEM);
        if (ret)
            goto err_cred;
        ret = gnutls_init(&session, GNUTLS_SERVER |
                                    GNUTLS_NO_AUTO_SEND_TICKET |
                                    GNUTLS_ENABLE_EARLY_DATA |
                                    GNUTLS_NO_END_OF_EARLY_DATA);
        if (ret)
            goto err_cred;
        ret = gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                                     cred);
        if (ret)
```

```
            goto err_session;

        ret = gnutls_session_ticket_enable_server(session, &skey);
        if (ret)
            goto err_session;

        ret = gnutls_priority_set_direct(session, QUIC_PRIORITY, NULL);
        if (ret)
            goto err_session;

        if (alpns) {
            ret = quic_session_set_alpn(session, alpns, strlen(alpns));
            if (ret)
                goto err_session;
        }

        gnutls_transport_set_int(session, sockfd);

        ret = quic_handshake(session);
        if (ret)
            goto err_session;

        if (alpns) {
            alpn_len = sizeof(alpn);
            ret = quic_session_get_alpn(session, alpn, &alpn_len);
        }

    err_session:
        gnutls_deinit(session);
    err_cred:
        gnutls_certificate_free_credentials(cred);
    err:
        return ret;
    }
```

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

This example illustrates how to use the TOKEN, SESSION_TICKET, and TRANSPORT_PARAM socket options to achieve session resumption and 0-RTT data transmission with QUIC (client_handshake() and server are from Appendix C).

```
#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 param_len, token_len, addr_len, flags;
    struct quic_transport_param param = {};
    struct sockaddr_in ra = {}, la = {};
    char msg[50], *alpn;
    size_t ticket_len;
    int ret, sockfd;
    int64_t sid;

    if (argc < 3) {
        printf("%s client <PEER ADDR> <PEER PORT> [ALPN]\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;
    }

    /* get ticket and param after handshake
     * (you can save it somewhere).
     */
    alpn = argv[4];
    ticket_len = sizeof(ticket);
    if (client_handshake(sockfd, alpn, NULL, NULL, ticket,
                         &ticket_len))
        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;
        }

        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!");
        sid = 2;
        flags = MSG_STREAM_NEW | MSG_STREAM_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 '%s' on stream %ld\n", msg, sid);

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

        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:port and set token for address
        * validation
        */
       if (bind(sockfd, (struct sockaddr *)&la, addr_len)) {
           printf("socket bind 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;
       }

       /* set the ticket and remote param and early data into
        * the socket for handshake.
        */
       ret = setsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_TOKEN, token,
                        token_len);
       if (ret == -1) {
           printf("socket setsockopt token\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;
       }

       /* send early data before handshake */
       strcpy(msg, "hello quic server, I’m back!");
       sid = 2;
       flags = MSG_STREAM_NEW | MSG_STREAM_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 '%s' on stream %ld\n", msg, sid);

        if (client_handshake(sockfd, alpn, ticket, ticket_len, NULL,
                             NULL))
            return -1;

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

        close(sockfd);
        return 0;
    }

    static int do_server(int argc, char *argv[])
    {
        unsigned int addrlen, keylen, flags;
        struct quic_config config = {};
        struct sockaddr_in sa = {};
        int listenfd, sockfd, ret;
        char msg[50], *alpn;
        uint8_t key[64];
        int64_t sid;

        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;
        }
        alpn = argv[6];
        if (alpn && setsockopt(listenfd, SOL_QUIC, QUIC_SOCKOPT_ALPN,
                               alpn, strlen(alpn))) {
            printf("socket setsockopt alpn failed\n");
            return -1;
        }

        if (listen(listenfd, 1)) {
            printf("socket listen failed\n");
            return -1;
        }
        config.validate_peer_address = 1; /* trigger retry packet */
        if (setsockopt(listenfd, SOL_QUIC, QUIC_SOCKOPT_CONFIG, &config,
                       sizeof(config)))
            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;
        }

        keylen = sizeof(key);
        if (getsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_SESSION_TICKET,
                       key, &keylen)) {
            printf("socket getsockopt session ticket error %d", errno);
            return -1;
        }

        if (server_handshake(sockfd, argv[4], argv[5], alpn, key,
                             keylen))
            return -1;

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

        strcpy(msg, "hello quic client!");
        sid = 1;
```

```
        flags = MSG_STREAM_NEW | MSG_STREAM_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 '%s' on stream %ld\n", msg, sid);

        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 (server_handshake(sockfd, argv[4], argv[5], alpn, key,
                             keylen))
            return -1;

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

        strcpy(msg, "hello quic client! welcome back!");
        sid = 1;
        flags = MSG_STREAM_NEW | MSG_STREAM_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 '%s' on stream %ld\n", msg, sid);

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

    int main(int argc, char *argv[])
```

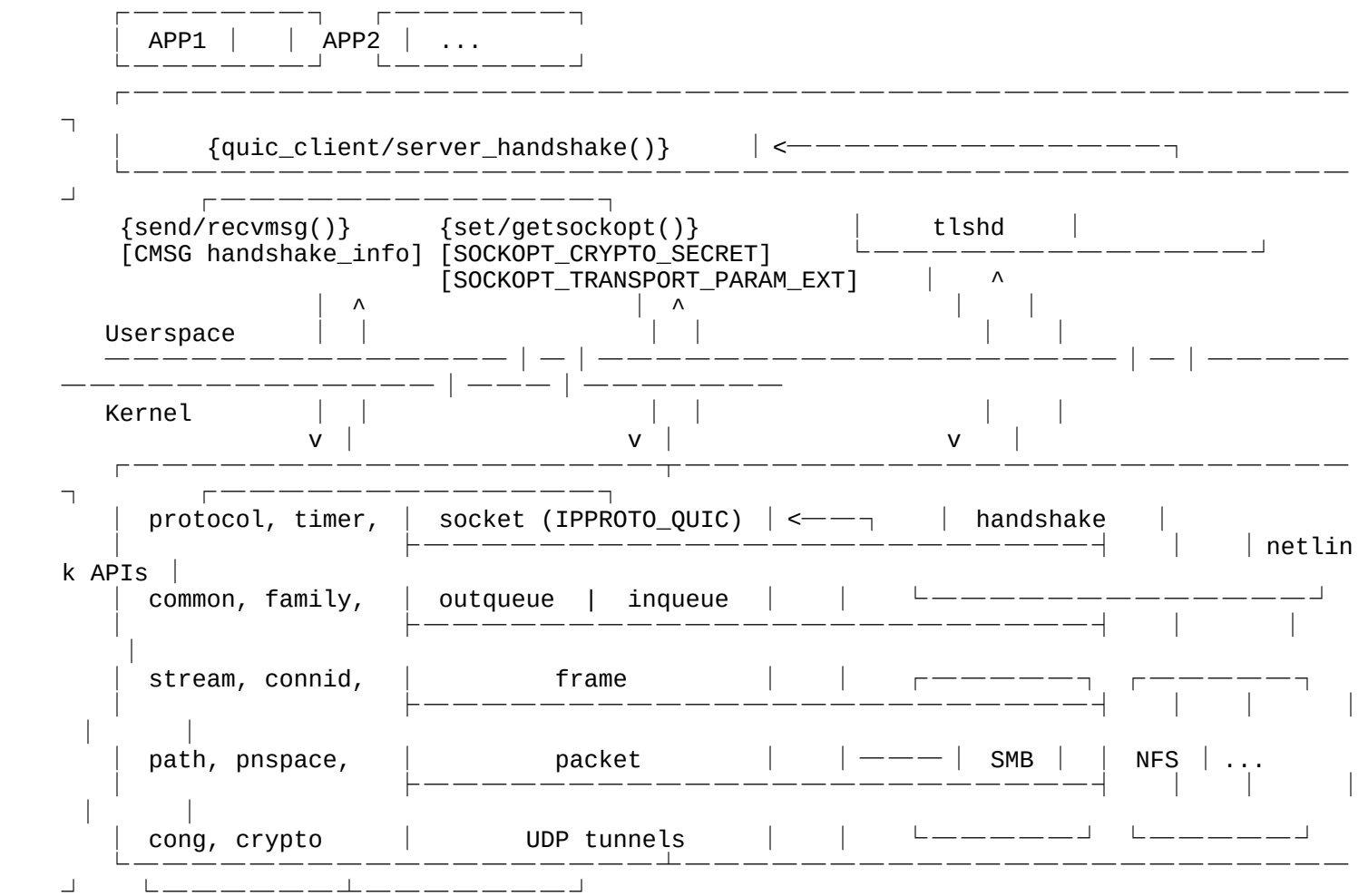
```
   {
       if (argc < 2 ||
           (strcmp(argv[1], "server") && strcmp(argv[1], "client"))) {
           printf("%s server|client ...\n", argv[0]);
           return 0;
       }

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

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

Appendix E. Example For Kernel Consumers Architecture Design

In-kernel QUIC facilitates integration with kernel consumers. Below is the design architecture for handling QUIC handshakes in Linux kernel:



Authors' Addresses

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

Moritz Buhl (editor)
Technical University of Munich
Boltzmannstrasse 3
85748 Garching
Germany
Email: ietf@moritzbuhl.de

Marcelo Ricardo Leitner (editor)
Red Hat
Av. Brg. Faria Lima, 3732
Sao Paolo-SP
Brazil
Email: mleitner@redhat.com