Linux QUIC: Bringing a Modern Secure Transport into the Kernel

1st Xin Long
Red Hat
Ottawa, Canada
lucien.xin@gmail.com

Abstract

QUIC has emerged as a foundational transport protocol for modern Internet applications by integrating transport and cryptographic handshakes, providing multiplexed streams, and enabling seamless connection migration. Existing QUIC deployments are almost exclusively based on userspace libraries, which simplifies protocol evolution but limits adoption by kernel subsystems and prevents the operating system from fully exploiting existing networking optimizations and hardware acceleration mechanisms.

This paper presents Linux QUIC, an implementation of QUIC integrated into the Linux kernel networking stack. Linux QUIC introduces a new socket abstraction that preserves the familiar POSIX programming model while exposing QUIC-specific functionality through control messages, socket options, and asynchronous notifications. To avoid duplicating mature TLS infrastructure in the kernel, Linux QUIC adopts a hybrid architecture in which TLS handshake processing remains in userspace while transport functionality—including packet processing, congestion control, retransmission, stream scheduling, and connection migration—is implemented entirely in the kernel.

The implementation is designed to support both userspace applications and kernel consumers such as SMB and NFS while remaining compatible with existing Linux networking infrastructure. Linux QUIC additionally provides mechanisms for stream peel-off, application-layer protocol negotiation, and future hardware offload support. We describe the architecture and programming interfaces of Linux QUIC and evaluate the implementation using interoperability testing, fuzzing, and performance measurements. Experimental results demonstrate broad standards compliance and competitive performance while enabling deployment scenarios that are difficult to support using userspace-only implementations.

Keywords

QUIC, Linux QUIC, in-kernel QUIC, socket API, POSIX API, kernel networking, transport protocols, stream multiplexing, connection migration, congestion control

Introduction

QUIC has become one of the most important transport protocols introduced in recent decades. Standardized by the IETF in RFC 9000 [1] and related specifications, QUIC combines transport and cryptographic handshakes, supports native stream multiplexing, enables connection migration, and incorporates modern congestion-control and loss-recovery mechanisms. The protocol has rapidly gained adoption through applications such as HTTP/3 and is implemented in several mature userspace libraries, including MsQuic, quiche, ngtcp2, and picoquic.

The decision to implement QUIC in userspace was intentional. Unlike TCP, whose evolution is constrained by kernel release cycles and operating-system deployment requirements, QUIC operates over UDP and encrypts most transport metadata, allowing new features and extensions to be deployed independently of kernel upgrades.

However, practical deployment scenarios reveal several limitations of the userspace-only model. An increasing number of kernel subsystems, including SMB and NFS, are adopting QUIC as a transport protocol. Integrating these subsystems with userspace QUIC libraries introduces additional complexity, data copies, and context-switch overhead. At the same time, the Linux kernel already provides mature abstractions for transport protocols, including standardized socket interfaces, packet scheduling, zero-copy mechanisms, and hardware acceleration frameworks.

This paper presents Linux QUIC, an implementation of QUIC integrated into the Linux kernel networking stack. Linux QUIC does not seek to replace existing userspace libraries. Instead, it complements them by enabling deployment scenarios that benefit from kernel integration while preserving compatibility with existing application interfaces.

Designing QUIC for the kernel introduces several challenges. QUIC tightly integrates TLS 1.3 with transport-layer functionality, making it undesirable to duplicate certificate validation and cryptographic policy enforcement inside the kernel. Furthermore, QUIC introduces concepts such as stream multiplexing, connection migration, and application-layer protocol negotiation that do not naturally fit within traditional socket abstractions.

Linux QUIC addresses these challenges through a hybrid architecture that separates TLS processing from transport functionality. TLS handshake message generation and processing remain in userspace, leveraging existing cryptographic libraries and Linux handshake services, while packet processing, congestion control, retransmission, flow control, and stream management execute entirely in the kernel.

To expose QUIC functionality, Linux QUIC extends the socket API using ancillary data, socket options, asynchronous

notifications, and stream peel-off. These mechanisms preserve the familiar POSIX programming model while enabling applications and kernel consumers to access QUIC-specific capabilities.

Linux QUIC is currently under review for upstream inclusion in the Linux kernel. The implementation supports the core QUIC specifications, including transport, TLS integration, and loss recovery, together with extensions such as QUIC Datagrams and QUIC Version 2. We evaluate the implementation through interoperability testing, continuous fuzzing, and performance experiments.

The contributions of this paper are as follows:

- We present Linux QUIC, a kernel-integrated QUIC implementation for Linux.
- We introduce a hybrid architecture that separates TLS processing and transport functionality between userspace and kernel components.
- We extend the Linux socket API with QUIC-specific control messages, socket options, notifications, and stream peel-off.
- We demonstrate compatibility with existing applications and kernel subsystems.
- We evaluate Linux QUIC using interoperability testing, fuzzing, and performance measurements.

Motivation

The prevailing view in the networking community is that QUIC belongs in userspace. This design choice simplifies protocol evolution and avoids many deployment challenges associated with kernel networking stacks. Nevertheless, several practical considerations motivate the integration of QUIC into the Linux kernel.

First, an increasing number of kernel consumers require secure and multiplexed transport protocols. Storage systems such as SMB and NFS are actively adopting QUIC to benefit from built-in encryption, stream multiplexing, and connection migration. Integrating these subsystems with userspace libraries introduces complexity and incurs additional data movement across the kernel-userspace boundary.

Second, Linux already provides a mature transport abstraction based on sockets. Existing applications rely on interfaces such as `listen()`, `accept()`, `connect()`, `sendmsg()`, and `recvmsg()`. Preserving these abstractions lowers the barrier to adoption and enables existing applications to migrate to QUIC incrementally.

Third, kernel integration allows QUIC to leverage optimizations already present in the networking stack, including Generic Segmentation Offload (GSO), packet scheduling, zero-copy transmission, and hardware cryptographic acceleration.

Finally, QUIC enables new deployment scenarios that benefit from kernel support. Application-Layer Protocol Negotiation (ALPN) allows incoming QUIC connections to be routed dynamically to different applications or services. Future hardware support for QUIC processing can further reduce CPU overhead in high-throughput environments.

At the same time, QUIC presents challenges that distinguish it from traditional transport protocols. QUIC combines transport and TLS handshakes, supports multiple streams within a single connection, encrypts most transport metadata, and identifies connections using connection identifiers rather than network addresses. These properties require extensions to the traditional socket model and motivate the design choices described in the following sections.

Implementation

Linux QUIC adopts a hybrid architecture that separates TLS handshake processing from QUIC transport functionality while preserving compatibility with the Linux networking stack.

Userspace–Kernel Split

One of the primary design challenges concerns the integration of TLS. QUIC tightly couples transport and cryptographic handshakes, making it impractical to reimplement certificate validation, cryptographic policy management, and other TLS functionality inside the kernel.

Linux QUIC therefore separates responsibilities between userspace and kernel components. TLS handshake generation and processing, certificate verification, and cryptographic policy decisions remain in userspace and rely on existing TLS libraries such as GnuTLS. The kernel implements QUIC transport functions, including packet processing, congestion control, loss recovery, stream scheduling, flow control, pacing, and path management.

In userspace, TLS handshake messages are exchanged through standard socket operations using QUIC-specific control messages. Once cryptographic secrets are generated, they are installed into the kernel transport layer using socket options.

In the kernel, Linux QUIC introduces a dedicated socket protocol type `IPPROTO_QUIC`, following the socket extension model used by protocols such as `IPPROTO_MPTCP`. This avoids requiring a new IP protocol number.

Unlike TCP, Linux QUIC is implemented on top of the Linux UDP tunnel infrastructure. This design matches the QUIC connection model, where connections are identified by connection IDs rather than the UDP 4-tuple, and allows a QUIC connection to migrate between different UDP endpoints during its lifetime.

For kernel consumers, Linux QUIC uses the existing net handshake infrastructure to communicate with userspace TLS services such as `tlshd`, which will be discussed later.

Handshake Architecture

Linux QUIC supports two different handshake workflows: userspace applications and kernel consumers as shown in Figure 1.

For userspace applications, the application performs the TLS handshake through a TLS library integration. The TLS library communicates with the QUIC socket using the QUIC handshake API, while handshake records are exchanged through socket control messages.

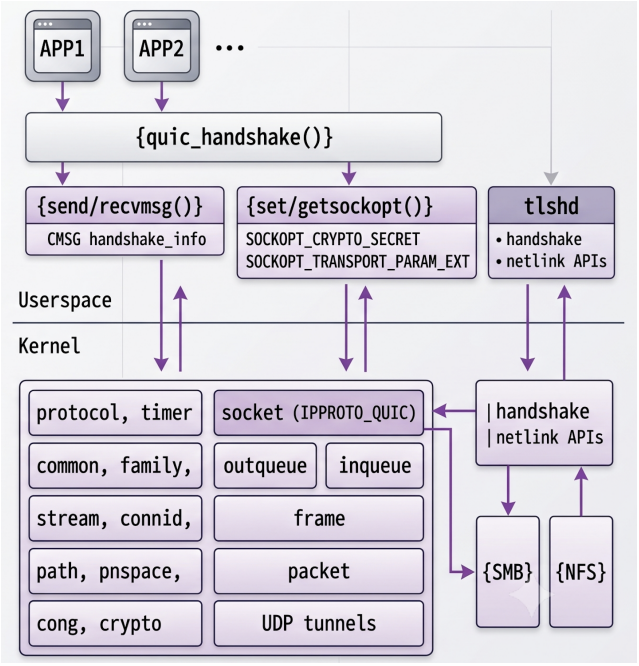

Figure 1: Handshake Architecture.

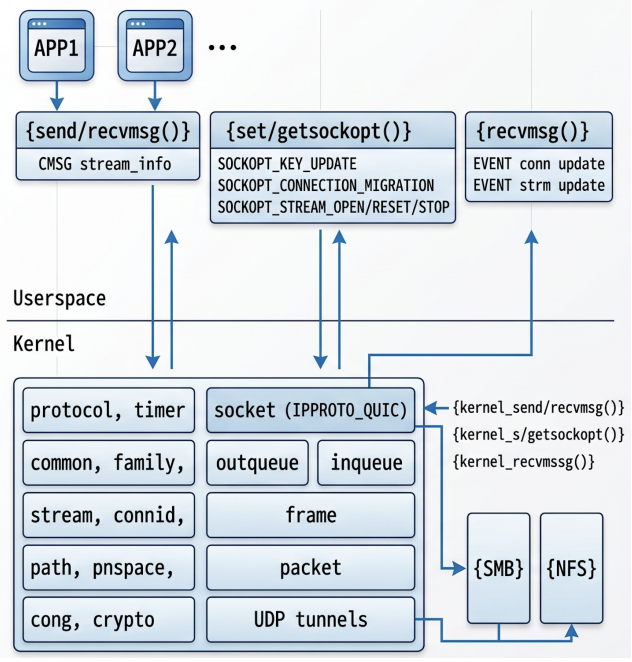

Figure 2: User Data Architecture.

For kernel consumers, Linux QUIC uses the existing Linux handshake infrastructure. Kernel subsystems such as SMB and NFS do not directly implement TLS. Instead, they delegate TLS processing to a userspace daemon, such as `tlshd`. The daemon performs certificate verification and secret derivation, then transfers the resulting cryptographic state into the kernel QUIC socket.

This model keeps security-sensitive TLS policy decisions in userspace while allowing kernel subsystems to consume QUIC through a native kernel socket interface.

User Data Architecture

After handshake completion, QUIC data processing is fully handled in the kernel shown in Figure 2.

Applications use standard socket operations to exchange stream data, while QUIC-specific information is provided through ancillary control messages and socket options.

Unlike TCP, a QUIC connection contains multiple independent streams. Linux QUIC exposes stream management through socket APIs, including stream creation, stream state notification, and stream-specific operations.

Core Data Structures

The central data structure in Linux QUIC is `quic_sock`, which represents a QUIC connection and extends the Linux socket abstraction with QUIC-specific transport state.

A `quic_sock` is integrated into the Linux socket lookup infrastructure and is inserted into the appropriate listening or established socket hash table. In addition, Linux QUIC maintains a dedicated connection ID hash table indexed by source connection IDs to support QUIC packet demultiplexing, since QUIC connections are identified independently of the network 4-tuple.

Beyond basic socket state, `quic_sock` maintains the major components required by the QUIC protocol, including stream management, packet number spaces, cryptographic contexts, congestion control state, loss recovery timers, and network path information. This organization allows a single QUIC connection to efficiently manage multiple streams and adapt to network changes such as connection migration.

```
struct quic_hashinfo {
  struct quic_shash_table shash;    // 1 : * quic_source_conn_id
  struct quic_shash_table lhash;    // 1 : * sock (listen)
  struct quic_shash_table chash;    // 1 : * sock (regular)
  struct quic_uhash_table uhash;    // 1 : * quic_udp_sock
};

struct quic_sock {
  struct inet_sock          inet;   // 1 : 1 sock
  struct list_head          reqs;   // 1 : * quic_request_sock
                                    //       (listen)

  struct quic_data          ticket;
  struct quic_data          token;
  struct quic_data          alpn;

  struct quic_stream_table  streams; // 1 : * quic_stream
  struct quic_conn_id_set   source;  // 1 : * quic_source_conn_id
  struct quic_conn_id_set   dest;    // 1 : * quic_dest_conn_id
  struct quic_path_group    paths;   // 1 : * quic_path \
                                     // 1 : 1 quic_udp_sock
  struct quic_cong          cong;
  struct quic_pnspace       space[];
  struct quic_crypto        crypto[];

  struct quic_outqueue      outq;    // 1 : * quic_frame
  struct quic_inqueue       inq;     // 1 : * quic_frame
  struct quic_packet        packet;
  struct quic_timer         timers[];
};
```

Interfaces

Linux QUIC extends the traditional socket API while preserving compatibility with existing applications and kernel subsystems.

Programming Model

Userspace Applications Applications create QUIC sockets and interact with them using familiar system calls, as shown in Figure 3.

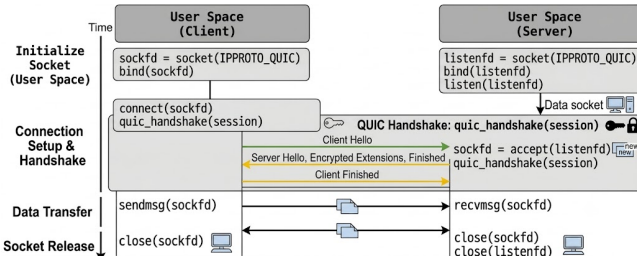

Figure 3: Userspace application workflow.

The main difference from traditional sockets is that `connect()` does not establish the QUIC connection; it only performs local socket initialization. The actual handshake is performed through the `quic_handshake()` function. The client initiates the handshake by sending a handshake message, the server processes the request and returns a new socket through `accept()`, and the handshake continues on the new socket until completion.

The `quic_handshake()` function is intended to be integrated into TLS libraries, providing applications with a familiar TLS-style interface while the QUIC transport remains implemented in the kernel. The TLS integration is discussed in more detail later.

Kernel Consumers Kernel subcomponents use similar socket interfaces, as shown in Figure 4.

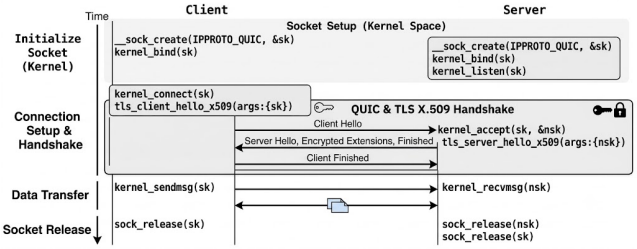

Figure 4: Kernel consumer workflow.

The primary difference is that the handshake is performed through the kernel handshake infrastructure, using helpers such as `tls_client_hello_x509()` and `tls_server_hello_x509()`. The userspace `tlshd` daemon must be running to provide TLS handshake processing services.

Socket API

Control Messages Linux QUIC introduces two classes of ancillary data.

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);

struct msghdr {
  ...
  void *msg_control;    /* ancillary data */
  socklen_t msg_controllen; /* ancillary data buffer length */
};

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[]; */
};

cmsg_level:

    #define SOL_QUIC        288

cmsg_type:

    enum quic_cmsg_type {
        QUIC_STREAM_INFO,
        QUIC_HANDSHAKE_INFO,
    };

cmsg_data:

    struct quic_stream_info {
          __s64    stream_id;
          __u32    stream_flags;
    };

    struct quic_handshake_info {
          __u8     crypto_level;
    };
```

Handshake control messages identify the encryption level associated with TLS records, including Initial, Handshake, and Application Data levels.

Stream control messages carry stream metadata, including stream identifiers and stream flags used to open, close, or reset streams.

Socket Options Socket options expose QUIC-specific functionality throughout the connection lifecycle.

Before connection establishment, applications may configure transport parameters and ALPN values.

```
#define QUIC_SOCKOPT_TRANSPORT_PARAM                8
#define QUIC_SOCKOPT_CONFIG                         9
#define QUIC_SOCKOPT_TOKEN                          10
#define QUIC_SOCKOPT_ALPN                           11
#define QUIC_SOCKOPT_SESSION_TICKET                 12
```

During the handshake, socket options install cryptographic secrets and transport extensions.

```
#define QUIC_SOCKOPT_CRYPTO_SECRET                  13
#define QUIC_SOCKOPT_TRANSPORT_PARAM_EXT            14
```

After handshake completion, socket options support stream management, connection migration, key updates, and transport-state queries.

```
#define QUIC_SOCKOPT_EVENT                          0
#define QUIC_SOCKOPT_STREAM_OPEN                    1
#define QUIC_SOCKOPT_STREAM_RESET                   2
#define QUIC_SOCKOPT_STREAM_STOP_SENDING            3
#define QUIC_SOCKOPT_CONNECTION_ID                  4
#define QUIC_SOCKOPT_CONNECTION_CLOSE               5
#define QUIC_SOCKOPT_CONNECTION_MIGRATION           6
#define QUIC_SOCKOPT_KEY_UPDATE                     7
```

Linux QUIC also introduces stream peel-off, which detaches an individual stream into a dedicated socket descriptor.

```
/* in draft */
#define QUIC_SOCKOPT_STREAM_PEELOFF                 xx
```

After peel-off, applications can interact with the stream using conventional `send()` and `recv()` operations without explicitly specifying stream identifiers. This mechanism simplifies the integration of QUIC into applications originally designed around TCP sockets.

```c
/* Open an 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)
    return -1;

/* Peel off the stream from the socket */
optlen = sizeof(pinfo);
pinfo.stream_id = sinfo.stream_id;
strmfd = getsockopt(sockfd, SOL_QUIC, QUIC_SOCKOPT_STREAM_PEELOFF,
                    &pinfo, &optlen);
if (strmfd < 0)
    return -1;

/* Send messages without stream_info cmsg */
ret = send(strmfd, msg, strlen(msg), 0);
```

Notifications QUIC introduces events that do not exist in traditional transport protocols. Streams may open or close, paths may migrate, and cryptographic keys may be updated.

Linux QUIC delivers these events asynchronously through the receive path. Applications subscribe to notification classes and receive structured event messages alongside regular transport data.

```c
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);

struct msghdr {
  ...
  int msg_flags;            /* flags on message */
};

msg_flags:
  enum quic_msg_flags {
    ...
    MSG_QUIC_NOTIFICATION           = MSG_MORE,
  };

msg_data:
  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,
    QUIC_EVENT_MAX,
  };
```

Notifications additionally provide stream-level flow-control updates, allowing applications to implement custom scheduling policies.

TLS API

We also proposed a new API in GnuTLS to integrate TLS sessions with QUIC sockets. This API allows applications to reuse the familiar GnuTLS programming model while performing a TLS 1.3 handshake over a QUIC transport.

```c
/**
 * gnutls_quic_handshake:
 * @session: a #gnutls_session_t initialized for QUIC transport.
 *
 * Perform a TLS 1.3 handshake over a QUIC transport.
 *
 * Returns: %GNUTLS_E_SUCCESS on success, or a negative error code.
 **/
int gnutls_quic_handshake(gnutls_session_t session);
```

The application initializes the GnuTLS session as usual, associates it with a QUIC socket file descriptor, and invokes `gnutls_quic_handshake()` to complete the QUIC handshake. This provides a consistent TLS API experience while allowing the transport processing to remain in the kernel.

```c
gnutls_certificate_credentials_t x509_cred;
gnutls_session_t session;

gnutls_certificate_allocate_credentials(&x509_cred);

/* GNUTLS_SERVER for server side */
gnutls_init(&session, GNUTLS_CLIENT);
gnutls_handshake_set_timeout(session, 0);

gnutls_priority_set_direct(session, prio, NULL);

gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, x509_cred);

/* fd is QUIC socket after connect() for client or accept()
 * for server.
 */
gnutls_transport_set_int(session, fd);

ret = gnutls_quic_handshake(session);
```

Use Cases

Linux QUIC is designed to support both userspace applications and kernel subsystems.

Samba Integration

Linux QUIC has been integrated into Samba on both the client and server sides by Stefan Metzmacher [2]. Samba provides a realistic workload that exercises QUIC connection management and high-throughput SMB data transfer. The integration required only minimal changes to Samba's transport layer: the existing SMB protocol processing remains unchanged, while the underlying transport can be switched from TCP to QUIC using the Linux QUIC socket API.

The Samba integration validates the Linux QUIC socket API design and demonstrates that existing applications can adopt Linux QUIC with minimal architectural changes.

SMB and NFS

One of the primary motivations for Linux QUIC is to enable in-kernel consumers.

The Linux handshake daemon, `tlshd`, provides QUIC handshake processing support as part of `ktls-utils` [12].

SMB over QUIC support on Linux is currently under development and is expected to become available once Linux QUIC is merged upstream. The initial SMB over Linux QUIC patchset has already been submitted for review [3].

NFS over QUIC is also being explored within the IETF community through the RPC-over-QUIC effort [4]. The proposed protocol enables existing RPC-based applications, including NFSv4, to use QUIC as the underlying transport without requiring changes to RPC semantics.

Linux QUIC provides the transport infrastructure required by these kernel subsystems while avoiding the need to embed TLS implementations directly in the kernel.

HTTP/3 and Curl

Linux QUIC has been integrated experimentally into curl to provide HTTP/3 support by Moritz Buhl [5]. Prototype HTTP/3 servers have also been implemented using the Linux QUIC socket API.

These experiments demonstrate that Linux QUIC is suitable not only for kernel consumers but also for traditional userspace applications.

Additional Applications

Performance tools such as NetPerfMeter [6] provide QUIC support for evaluating Linux QUIC performance and comparing it with other transport protocols and QUIC implementations.

Beyond storage systems and web services, Linux QUIC can benefit a broad range of applications that require secure, reliable, and multiplexed transport. Potential use cases include NVMe, RPC frameworks, distributed storage systems, VPN technologies, and high-performance network services.

By providing a standard socket-based interface and kernel-assisted transport processing, Linux QUIC enables existing applications to adopt QUIC with minimal architectural changes, without requiring each project to maintain an independent QUIC implementation.

Testing

Linux QUIC has been evaluated using interoperability testing, fuzzing, and performance experiments.

Interoperability Testing

Interoperability testing is performed using [7].

Linux QUIC is continuously tested against major implementations, including MsQuic, quiche, picoquic, ngtcp2, and neqo. The test suite covers handshake, versionnegotiation, transfer, chacha20, keyupdate, retry, resumption, zerortt, http3, multiconnect, v2, rebind-port, rebind-addr and connectionmigration.

The results in Figure 5 demonstrate broad standards compliance and successful interoperability across implementations.

Figure 5: Functions Test Result with QUIC Interop Runner.

The Interop Runner additionally provides throughput measurements under constrained network conditions. It measures QUIC connection goodput over a 10 Mbps link. The first result in each box represents a QUIC-only flow, while the second represents the cross-traffic scenario, where a TCP flow competes for the same link capacity.

The results in Figure 6 show that Linux QUIC achieves competitive performance under both scenarios.

Figure 6: Measurement Test Result with QUIC Interop Runner.

Syzkaller Fuzzing

Linux QUIC is continuously fuzzed using [8] with the TEST_CASE [9].

Install test case into syzkaller:

```
# git clone https://github.com/google/syzkaller
# cd syzkaller/
# cp TEST_CASE/socket_inet_quic.txt* sys/linux/
# make all && cp bin/* /usr/local/bin/
```

Run with enable_syscalls in syzkaller-test.cfg:

```
"enable_syscalls" : [
  "socket$inet_quic",
  "socket$inet6_quic",
  "getsockopt$inet_quic6_*",
  "getsockopt$inet_quic_*",
  "setsockopt$inet_quic6_*",
  "setsockopt$inet_quic_*",
  "ioctl$sock_inet6_quic_*",
  "ioctl$sock_inet_quic_*",
  "getsockname", "getpeername",
  "bind", "listen", "connect", "accept", "accept4", "close",
  "shutdown", "sendmsg", "sendmmsg", "sendto", "recvmsg",
  "recvmmsg", "recvfrom", "epoll_create", "epoll_create1",
  "epoll_ctl", "epoll_wait", "epoll_pwait", "epoll_pwait2",
  "mmap", "poll", "ppoll", "pread64", "preadv", "select",
  "pselect6", "sendfile"
],
```

Support for QUIC-specific system calls and socket operations has been added to the fuzzing framework, enabling automatic generation of workloads that exercise connection establishment, stream operations, transport parameters, and protocol state transitions.

Fuzz testing has been employed since the beginning of the project and has uncovered numerous bugs and corner cases.

Performance Evaluation

Performance experiments compare Linux QUIC with both kernel TLS and existing userspace QUIC implementations.

With kTLS Experiments against kTLS were performed using modified benchmarking tools [10] for QUIC and [11] for kTLS on systems connected through 100-Gbit/s network interfaces under different MTU configurations.

The results in Figure 7 show that Linux QUIC currently underperforms kTLS in some scenarios. However, the performance gap decreases substantially when TCP Generic Segmentation Offload is disabled or when larger MTUs are used. The remaining overhead primarily originates from additional memory copies during STREAM frame construction and the cryptographic processing required for QUIC header protection.

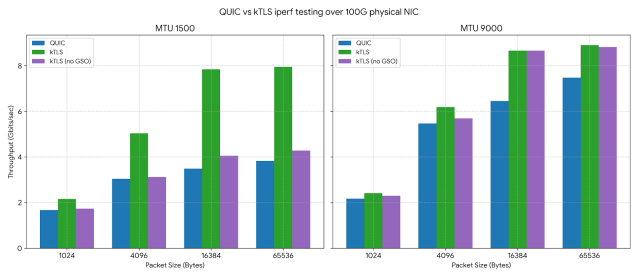

Figure 7: Performance Test Result vs kTLS.

With other QUIC implementations via Samba Additional experiments compare Linux QUIC with userspace QUIC implementations using Samba workloads provided by Sebastian Rust. The experiments evaluate different client and server OS combinations over 10 Gigabit NICs with various payload sizes, as shown in Figure 8.

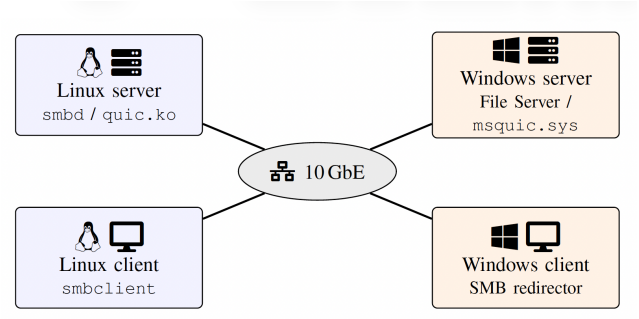

Figure 8: Performance Test Bed with other QUIC implementations.

The results in Figure 9 and Figure 10 show that Linux QUIC achieves better overall performance than TCP2 and approaches the performance of MsQuic when Linux QUIC is used as the server. However, Linux QUIC performs worse than MsQuic when the server side uses Windows MsQuic.

One observation from this test is that Linux QUIC did not perform well with small packet sizes. This was mainly due to the use of a large ACK delay configuration, which introduced additional latency and reduced throughput. This issue has been addressed by adding piggyback ACK support, which improves performance for small packets.

With other QUIC implementations via HTTP/3 Additional experiments compare Linux QUIC with userspace QUIC implementations using HTTP/3 workloads provided by Moritz Buhl. The experiments measure the receiver execution time when downloading a 1 GB file via HTTP/3. Each QUIC client-server combination is evaluated over 30 repetitions using directly connected 10 Gigabit NICs.

The results in Figure 11 show that HTTP/3 performance varies significantly across different client-server combinations. The main observations are:

- No single client/server combination consistently achieves the best performance; application architecture and system integration often have a larger impact than the QUIC implementation alone.
- Curl-based clients generally outperform standalone interoperability clients.
- Production servers such as Nginx, Apache, and Tengine typically achieve lower goodput than lightweight interoperability servers due to additional application-layer overhead.

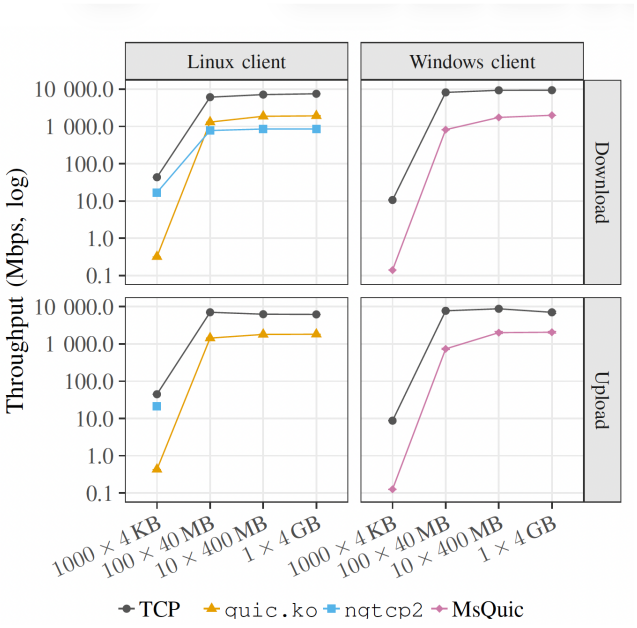

Figure 9: Performance Test Results with other QUIC implementations and Linux Server via Samba.

Overall, HTTP/3 performance depends not only on the QUIC implementation, but also on the design of the application and the surrounding system stack.

While no clear winner emerges from this evaluation, Linux QUIC achieves competitive performance compared with existing userspace QUIC implementations.

Future Work

Linux QUIC remains under active development, with ongoing work hosted through the Linux QUIC GitHub repository [13] and discussed on the Linux QUIC mailing list [14]. Several important directions remain for future work.

Upstreaming

The implementation is currently under review for inclusion in the upstream Linux kernel.

The code base is organized into multiple patch series covering infrastructure, transport functionality, socket APIs, and test suites. Parallel efforts are underway to integrate QUIC-aware APIs into TLS libraries such as GnuTLS.

The long-term goal is to provide complete QUIC support in standard Linux distributions without requiring additional libraries.

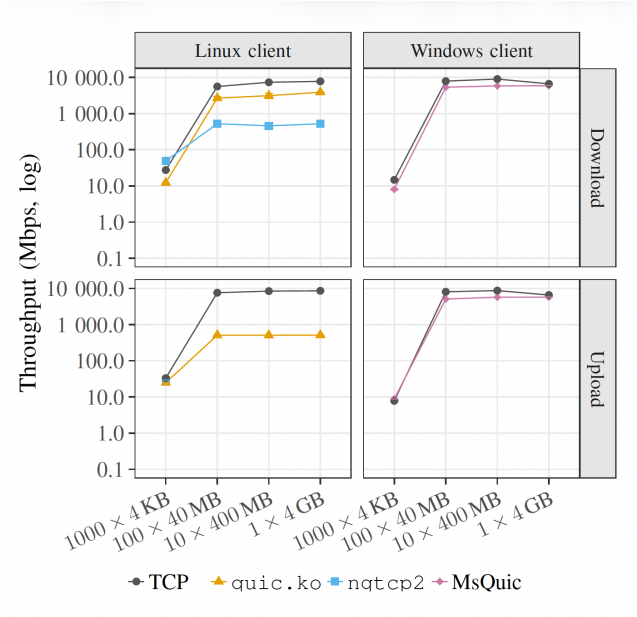

Figure 10: Performance Test Results with other QUIC implementations and Windows Server via Samba.

Hardware Offload

Hardware acceleration represents one of the most promising future directions.

Linux QUIC already includes preliminary infrastructure modeled after existing TLS and IPsec offload frameworks. Ongoing work explores implementations on FPGA-based NICs and commercial hardware platforms.

```
#define MAX_CONN_ID_SIZE        20
#define MAX_KEY_SIZE            32
#define IV_SIZE                 12

enum quic_crypto_dir {
  QUIC_CRYPTO_DIR_RX,
  QUIC_CRYPTO_DIR_TX,
};

struct quic_crypto_info {
  struct sockaddr_storage daddr;
  struct sockaddr_storage saddr;
  u8 conn_id[MAX_CONN_ID_SIZE];
  u8 conn_id_len;

  u8 data_key[MAX_KEY_SIZE];
  u8 hdr_key[MAX_KEY_SIZE];
  u8 iv[IV_SIZE];
  u16 cipher;
};

struct quicdev_ops {
  int (*quic_dev_add)(struct net_device *dev,
                      enum quic_crypto_dir dir,
                      struct quic_crypto_info *info);

  void (*quic_dev_del)(struct net_device *dev,
                       enum quic_crypto_dir dir,
                       struct quic_crypto_info *info);
};
```

Efficient hardware support has the potential to reduce CPU utilization and improve throughput in high-bandwidth environments.

API Evolution

The socket interface continues to evolve as new deployment scenarios emerge.

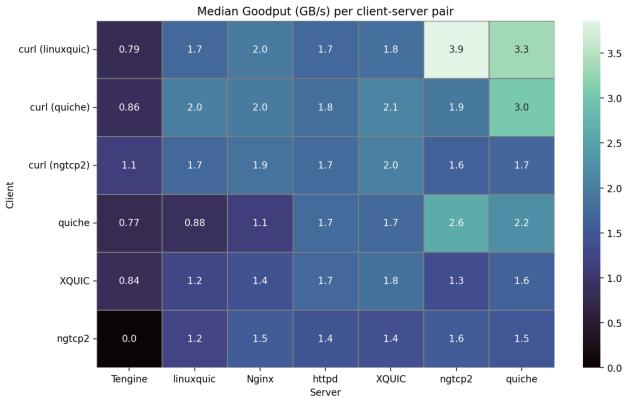

Figure 11: Performance Test Results with other QUIC implementations via HTTP/3.

To encourage portability across operating systems, QUIC socket API extensions have been proposed within the IETF [15]. Feedback from application developers, storage systems, and networking communities will guide future API design.

As additional applications adopt QUIC, Linux QUIC will continue to evolve to meet their requirements while preserving compatibility with existing Linux networking infrastructure.

Acknowledgments

TBD

References

[1] Iyengar, J. and Thomson, M. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. https://www.rfc-editor.org/rfc/rfc9000.

[2] Metzmacher, S. 2025. New Transports in Samba: QUIC and SMB-Direct Support. SNIA Storage Developer Conference (SDC) 2025 presentation. https://www.samba.org/~metze/presentations/2025/SDC/StefanMetzmacher_SDC2025-transports-rev1-compact.pdf.

[3] Carvalho, H. 2026. smb: implement SMB over QUIC linux-cifs mailing list. https://lore.kernel.org/linux-cifs/20260428154604.222551-1-henrique.carvalho@suse.com/.

[4] Coddington, B., Mayhew, S. and Lever, C. 2026. Remote Procedure Call over QUIC Version 1. https://datatracker.ietf.org/doc/draft-cel-nfsv4-rpc-over-quicv1/.

[5] Moritz, B. 2026. Quantitative Comparison of QUIC with Moritz Buhl. https://www.youtube.com/watch?v=TX3ZLRXXopI.

[6] Thomas D. 2025. A TCP/MPTCP/UDP/SCTP/DCCP/QUIC Network Performance Meter Tool. https://www.nntb.no/~dreibh/netperfmeter/.

[7] Marten S. 2019. QUIC / WebTransport Interop Runner. https://github.com/quic-interop/quic-interop-runner.

[8] Google. 2015. syzkaller: unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[9] Long X. 2024. QUIC Syzkaller Test Cases. https://github.com/lxin/quic/tree/main/tests/syzkaller.

[10] Long X. 2024. Network performance measurement tool for QUIC. https://github.com/lxin/iperf.

[11] Mellanox. 2015. Network performance measurement tool for kTLS. https://github.com/Mellanox/iperf_ssl.

[12] Chuck L. 2024. TLS handshake utilities for in-kernel TLS consumers. https://github.com/oracle/ktls-utils.

[13] Long, X. 2024. In-kernel QUIC implementation with Userspace handshake. https://github.com/lxin/quic.

[14] Long, X. 2025. Linux QUIC protocol development <quic@lists.linux.dev>. https://lore.kernel.org/quic/.

[15] Long, X. 2024. Sockets API Extensions for In-kernel QUIC Implementations. https://datatracker.ietf.org/doc/draft-lxin-quic-socket-apis/.