QUIC in Java

Vergleich zu TCP und Framework-Analyse

Kai Penazzi
AI, EMI
HS-Offenburg

Prof. Dr. rer. nat.
Erwin Mayer
Hands-On Seminar
Computer Netze 2

Contents

1 Einleitung

Im Internet werden Nachrichten hauptsächlich über die Transportprotokolle TCP und UDP übertragen. TCP gewährleistet eine zuverlässige, geordnete und verlustfreie Übertragung, kann aber durch seinen aufwendigen Verbindungsaufbau, eingeschränkte Erweiterungsmöglichkeiten und begrenzte integrierte Sicherheitsmechanismen in modernen, latenz- und sicherheitskritischen Anwendungen an seine Grenzen stoßen. Das QUIC-Protokoll wurde entwickelt, um diese Limitierungen zu überwinden und moderne Anforderungen wie geringe Latenz, zuverlässige Übertragung und integrierte Verschlüsselung zu erfüllen.
Diese Ausarbeitung gibt zunächst eine Einführung in die grundlegenden Konzepte und Eigenschaften des QUIC-Protokolls. Dabei werden zentrale Mechanismen wie Verbindungsaufbau, Multiplexing und integrierte Verschlüsselung erläutert. Darauf aufbauend werden anschließend die drei in Java verwendbaren QUIC-Frameworks Kwik, Quiche4J und Netty miteinander verglichen. Der Vergleich betrachtet unter anderem den Funktionsumfang, den Abstraktionsgrad sowie die Einsatzgebiete, für die sich die jeweiligen Frameworks am besten eignen.

2 Quic Einführung

2.1 Architektur

QUIC ist, ähnlich wie TCP, ein verbindungsorientiertes Client-Server-Protokoll. Viele Anwendungsprotokolle, die auf TCP basieren, nutzen mehrere parallele TCP-Verbindungen, um verschiedene Datenströme gleichzeitig zu übertragen.
Bei QUIC ist der Aufbau mehrerer Verbindungen nicht mehr erforderlich, da innerhalb einer einzelnen QUIC-Verbindung mehrere unabhängige Streams parallel übertragen werden können. Diese Trennung von Verbindung und Datenströmen ist ein zentrales Architekturprinzip von QUIC und bildet die Grundlage für den weiteren Paket- und Frame-Aufbau.

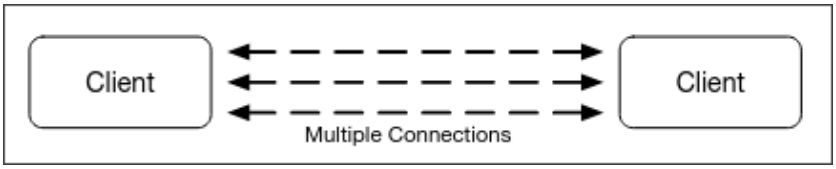

Figure 1: Parallelisierte Datenübertragung über mehrere TCP-Verbindungen [6]

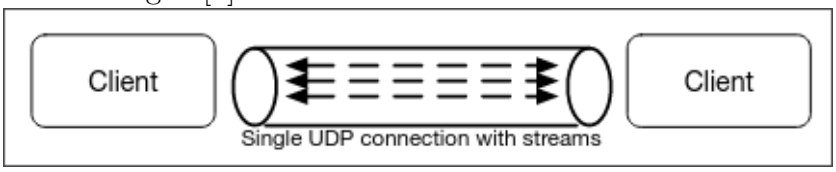

Figure 2: Parallelisierte Datenübertragung über eine einzelne QUIC-Verbindung [6]

2.2 Paket

QUIC-Kommunikation erfolgt über UDP-Datagramme, die ein oder mehrere QUIC-Pakete zwischen Endpunkten transportieren. Jedes QUIC-Paket besteht aus einem Header und einem oder mehreren Frames, die die eigentlichen Nutzdaten und Steuerinformationen enthalten. QUIC definiert dabei verschiedene Header- und Pakettypen, die unterschiedliche Funktionen wie Verbindungshandshake oder schnellen Datentransport übernehmen.

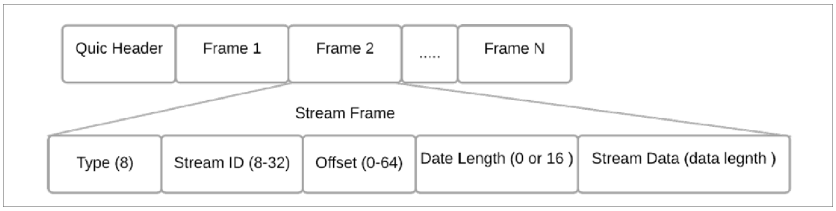

Figure 3: QUIC-Packet [2]

2.2.1 Header

QUIC läuft auf UDP, was bedeutet, dass die Zuordnung zu Anwendung und Endpunkten bereits durch die IP-Adressen und Ports auf UDP-Ebene erfolgt. Da QUIC auf Anwendungsschicht arbeitet, muss es nur noch die Zuordnung auf Verbindungsebene sicherstellen. QUIC verwendet zwei verschiedene Header-Typen. Der Long Header wird für den Verbindungsaufbau verwendet, während der Short Header bei bereits etablierte Verbindungen verwendet wird.

Longheader: [2]

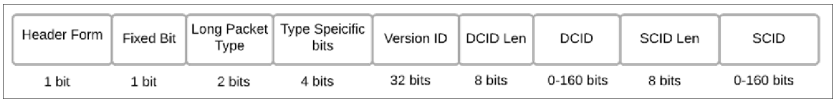

Figure 4: QUIC-Longheader [2]

- **Header Form (HF):** Identifiziert den Header-Typ.
- **Fixed Bit (FB):** Zeigt an, ob das Paket gültig ist oder nicht. Ist das Bit auf 0 gesetzt, ist das Paket ungültig.
- **Long Packet Type (T):** Gibt den Typ des Long-Header-Pakets an, siehe Figure 5.
- **Type-Specific Bits (S):** Bits, die spezifisch für die jeweiligen Long-Header-Pakettypen sind.
- **Version ID (VID):** 32 Bit zur Identifikation der QUIC-Version.

- **Destination Connection ID Length (DCID Len):** Länge der Ziel-Connection-ID.
- **Destination Connection ID (DCID):** Ziel-Connection-ID.
- **Source Connection ID Length (SCID Len):** Länge der Quell-Connection-ID.
- **Source Connection ID (SCID):** Quell-Connection-ID.

Type	Name	Description
0x00	Initial	Transports the first CRYPTO frames transmitted by the client and server during the key exchange and the Acknowledgment frames in both directions.
0x01	0-RTT	A 0-RTT packet sends "early" data from the client to the server before the handshake is completed.
0x02	Handshake	Packet is for sending and receiving encrypted handshake messages and acknowledgments between the server and the client.
0x03	Retry	The packet contains a server-generated address validation token. It's used by a server that wants to retry a connection.

Figure 5: Longheader Packet Typ [2]

Shortheader: [2]

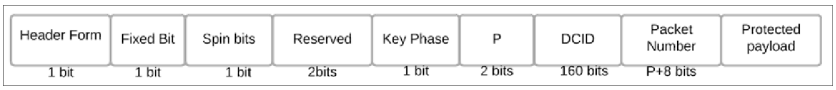

Figure 6: QUIC-Shortheader [2]

- **Header Form (HF):** Identifiziert den Header-Typ.
- **Fixed Bit (FB):** Zeigt an, ob das Paket gültig ist oder nicht.
- **Spin Bit:** Wird zur Latenzmessung verwendet.
- **Reserved Bits:** Für zukünftige Erweiterungen reserviert.
- **Key Phase:** Gibt an, welcher Verschlüsselungsschlüssel für das Paket verwendet wird.
- **Packet Number Length (P):** Länge der Paketnummer.
- **Destination Connection ID (DCID):** Ziel-Connection-ID.
- **Packet Number:** Monoton steigende Paketnummer zur Verlustdetektion und ACKs.

- **Packet Payload:** Enthält die Frames des Pakets (STREAM, ACK, PADDING etc.).

Diese Struktur ermöglicht es QUIC, die Stärken von TCP, wie etwa zuverlässige Datenübertragung, zu übernehmen, während gleichzeitig Verschlüsselung, Integrität und eine flexible Zuordnung von Verbindungen auf Anwendungsebene implementiert werden können. Die Verwendung unterschiedlicher Header-Typen sorgt dafür, dass das Protokoll effizient bleibt.

2.2.2 Frame

Ein Frame enthält die Daten eines Streams, der über QUIC übertragen wird.

- **Type:** Gibt die Art des Frames an, z. B. Stream-Daten, ACK oder Control-Informationen.
- **Stream ID:** Identifiziert den Stream, zu dem die Daten gehören. Ungerade IDs stehen für vom Client initiierte Streams, gerade IDs für vom Server initiierte Streams.
- **Offset:** Gibt die Position der Daten innerhalb des Streams an, ähnlich dem TCP-Offset.
- **Data Length:** Die Länge der übertragenen Daten in Bytes.
- **Stream Data:** Die eigentlichen Nutzdaten des Streams.

2.3 Verbindungsaufbau

Beim Aufbau einer Verbindung über TCP erfolgt zunächst ein dreistufiger Handshake (SYN, SYN-ACK, ACK), um eine Verbindung aufzubauen. Dieser Handshake verursacht mindestens eine Round-Trip-Time (RTT), bevor Daten übertragen werden können. Wird zusätzlich eine Verschlüsselung wie TLS verwendet, muss über der bereits etablierten TCP-Verbindung ein weiterer Handshake stattfinden, der in der Regel ein bis zwei zusätzliche RTTs benötigt. In Szenarien mit vielen kurzen Verbindungen summieren sich diese Verzögerungen und kann die Performance erheblich beeinträchtigen. Das QUIC-Protokoll verwendet UDP als Transportbasis und baut darauf eine eigene Verbindungsschicht auf. Bereits beim ersten Verbindungsaufbau werden die Verschlüsselungsschlüssel ausgehandelt, sodass dafür in der

Regel eine RTT erforderlich ist. Eine Wiederverbindung kann sogar ohne zusätzlichen Verbindungsaufbau erfolgen (0-RTT). Durch diese Integration von Transport- und Sicherheitsmechanismen reduziert QUIC die Latenz beim Verbindungsaufbau erheblich, wodurch es auch für Anwendungen mit häufigen oder kurzlebigen Verbindungen interessant wird.

2.3.1 Flusskontrolle

TCP und QUIC verwenden einen ähnlichen Congestion-Control-Mechanismus, um eine Überlastung des Netzwerks zu vermeiden. TCP nutzt dafür Algorithmen wie NewReno oder CUBIC, die die Menge an unbestätigten Paketen (in-flight) überwachen und das Senden neuer Pakete bremsen, sobald Paketverluste auftreten.
QUIC übernimmt diese Grundprinzipien und wendet die gleichen Algorithmen wie TCP an, verfolgt die Übertragung jedoch auf Ebene der Bytes-in-Flight, was eine feinere Kontrolle der Daten ermöglicht. Wie bei TCP dürfen auch bei QUIC Pakete nur gesendet werden, solange die Congestion-Window-Grenze nicht überschritten wird.

2.4 Paketverlust und Fehlerbehandlung

QUIC übernimmt viele grundlegende Konzepte der Verlustbehandlung aus TCP, darunter schnelle Wiederübertragungen sowie timeoutbasierte Mechanismen zur Verlustdetektion. Durch die Trennung von Transport- und Anwendungsschicht sowie den Betrieb im Userspace erweitert QUIC diese Mechanismen jedoch um zusätzliche Informationen.

2.4.1 Packet- und Datenzuordnung

Ein zentraler Unterschied zwischen TCP und QUIC liegt in der Handhabung der Sequenznummern. Während TCP eine einzige Sequenznummer für die gesamte Übertragung verwendet, verwendet QUIC ein abgewandelteres Konzept. Jedes Paket erhält eine monoton steigende Paketnummer, die für ACKs, RTT-Messungen und Loss Detection verwendet wird. Die eigentlichen Daten werden zusätzlich durch Stream-ID und Stream-Offset identifiziert, wodurch die Reihenfolge innerhalb eines Streams eindeutig bestimmt ist. Durch diese Trennung kann QUIC jederzeit genau nachvollziehen, welche Pakete bestätigt wurden und welche verloren gingen, was

die RTT-Berechnung und die Verlustbehandlung deutlich vereinfacht.

2.4.2 Ack-Zurücknehmen

QUIC erlaubt kein Zurücknehmen bereits bestätigter Pakete. Dadurch wird die Verlustbehandlung auf beiden seiten vereinfacht. [4]

2.4.3 Ack-Ranges

Im Gegensatz zu TCP, das ACKs höchstens drei SACK-Ranges unterstützt, erlaubt QUIC mehrere ACK-Ranges in einem einzelnen ACK-Paket. Diese Flexibilität beschleunigt die Paketverlust-Erkennung und reduziert unnötige Retransmissions insbesondere in Netzen mit hohem Paketverlust. [4]

2.4.4 Korrektur für verzögerte ACKs

QUIC-ACKs kodieren explizit die Verzögerung, die beim Empfänger zwischen dem Eintreffen eines Pakets und dem Versenden der zugehörigen Bestätigung entsteht. Dadurch kann der Empfänger des ACKs die empfangsseitige Verzögerungen in der Schätzung der RTT berücksichtigen. [4]

2.4.5 Fehlererkennung

TCP verwendet auf Paketebene eine Prüfsumme, um sicherzustellen, dass Header und Daten auf dem Transportweg nicht verändert wurden. QUIC gewährleistet die Integrität und Manipulationssicherheit seiner Pakete hingegen über die in das Protokoll integrierte TLS-1.3-Verschlüsselung. Jedes QUIC-Paket wird dabei verschlüsselt und enthält einen Authentifizierungs-Tag, sodass Änderungen am Header oder an den enthaltenen Frames vom Empfänger zuverlässig erkannt werden. Auf diese Weise übernimmt QUIC die Funktion der klassischen TCP-Prüfsumme, erweitert diese jedoch um kryptographisch gesicherte Integrität.
Wie auch bei TCP enthält QUIC standardmäßig keine Forward Error Correction Funktionalität, aufgrund der Implementierung im Userspace ist es jedoch deutlich einfacher, solche Erweiterungen nachträglich in QUIC zu integrieren.

2.5 QUIC Features

Durch die Architektur und dem Aufbau auf UDP kann QUIC zusätzliche Features implementieren, die über die Möglichkeiten klassischer TCP-Verbindungen hinausgehen.

2.5.1 Head-Of-Line Blocking

Head-of-Line Blocking passiert, wenn zum Beispiel HTTP/1.1 oder HTTP/2 mehrere parallele TCP-Verbindungen aufbauen, um mehrere Ressourcen gleichzeitig zu übertragen. Kommt es in einer dieser Verbindungen zu Verzögerungen oder Paketverlusten, blockieren die darauf aufbauenden Daten anderer Verbindungen, obwohl diese eigentlich schon übertragen werden könnten. Dadurch wird die gesamte Übertragung unnötig verlangsamt.
QUIC löst dieses Problem, indem es mehrere unabhängige Streams über eine einzige Verbindung laufen lässt. Geht auf Verbindungsebene ein Paket verloren, kümmert sich QUIC um das Pakethandling und die Retransmission nur für den betroffenen Stream, während die anderen Streams weiterhin ungehindert übertragen werden. Dadurch wird Head-of-Line Blocking effektiv vermieden und der Datenfluss insgesamt schneller und effizienter.

2.5.2 Connection Migration

Ein weiteres Feature von QUIC ist die Connection Migration. Dabei bleibt die Verbindung bestehen, selbst wenn sich die IP-Adresse oder das Netzwerk des Clients ändert, etwa beim Wechsel zwischen WLAN und Mobilfunk. TCP-Verbindungen würden in einem solchen Fall abbrechen und müssten neu aufgebaut werden. QUIC ermöglicht dies, da die Nachrichten über UDP übertragen werden und die Connection ID im Header die Verbindung eindeutig identifiziert.

2.5.3 0-RTT

QUIC unterstützt 0-RTT, wodurch beim Wiederaufbau einer Verbindung der Client sofort Daten senden kann, ohne den vollständigen Handshake erneut durchlaufen zu müssen. Dafür wird ein spezieller 0-RTT Long Header verwendet, der die Verbindung eindeutig identifiziert und die Daten korrekt dem jeweiligen Stream zuordnet.

2.5.4 Verschlüsselung

TLS 1.3 ist in QUIC implementiert, wodurch die Daten verschlüsselt sind und gegen Manipulation geschützt werden. Ein zusätzlicher Verschlüsselungslayer ist dadurch nicht erforderlich.

2.5.5 User-Space Implementierung

Da QUIC im Userspace arbeitet, können Erweiterungen, Updates oder experimentelle Features deutlich einfacher implementiert werden als bei klassischen Kernel-basierten TCP-Stacks.

3 Analyse von Java-QUIC-Frameworks

QUIC ist inzwischen kein rein theoretisches Protokoll mehr, sondern wird aktiv in Anwendungen eingesetzt. Im Folgenden werden drei Open-Source-Implementierungen des QUIC-Protokolls für Java betrachtet Kwik, Quiche4J sowie das QUIC-Modul von Netty (netty-incubator-codec-quic). Ziel der Analyse ist es, die jeweiligen APIs hinsichtlich Abstraktionsgrad und Integrationsaufwand zu vergleichen und einzuordnen, für welche Einsatzszenarien sich die einzelnen Frameworks besonders eignen.

3.1 Client Implementierungsvergleich

3.1.1 Kwik

Zunächst muss eine Verbindung aufgebaut werden. Dazu wird ein ‘QuicClientConnection‘ erstellt, konfiguriert und anschließend mit wird mit ‘connect()‘ die Verbindung aufgebaut.

```
String applicationProtocolId = "....";
QuicClientConnection connection = QuicClientConnection.newBuilder()
        .uri(URI.create("http://127.0.0.1:7000"))
        .applicationProtocol(applicationProtocolId)
        .build();

connection.connect();
```

Nach dem Verbindungsaufbau können Streams erstellt werden, über die Daten gelesen und geschrieben werden.

```
QuicStream quicStream = connection.createStream(true);
OutputStream output = quicStream.getOutputStream();
output.write(...)
output.close();
InputStream input = quicStream.getInputStream();
input.read(...)
```

Kwik bietet ein High-Level-API für QUIC in Java.

- Die Verbindung wird über ‘connect()‘ aufgebaut, inklusive Handshake und Socket-Management.
- Streams werden über ‘QuicStream‘-Objekte bereitgestellt, die ‘InputStream‘ und ‘OutputStream‘ unterstützen.
- Lesen und Schreiben von Daten erfolgt über Standard-Java-Streams, sehr einfach zu verwenden.
- Entwickler müssen sich kaum um Low-Level-Netzwerkdetails kümmern, Kwik übernimmt die meisten Aufgaben automatisch.

3.1.2 Quiche/Quiche4j

Zuerst muss ein Connection-Objekt erzeugt werden:

```
final Config config = new ConfigBuilder(Quiche.PROTOCOL_VERSION).build();
final byte[] connId = Quiche.newConnectionId();
final Connection conn = Quiche.connect("127.0.0.1:7000", connId, config);
```

Damit Quiche die Verbindung aufbauen kann, müssen regelmäßig die Send- und Recv-Loops ausgeführt werden. Der Entwickler muss dabei selbst entscheiden, wie und wie oft diese Loops ausgeführt werden und welcher UDP-Socket verwendet wird.

Recv-Loop

```
final byte[] buf = new byte[1350];
while(true) {
    DatagramPacket packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);
    final byte[] buffer = Arrays.copyOfRange(
        packet.getData(), packet.getOffset(), packet.getLength());
    final int read = conn.recv(buffer); // verarbeitet Pakete inkl. Handshake
    if(read <= 0) break;
}
```

Send-Loop

```
while(true) {
    final int len = conn.send(buf); // fragt, welche Pakete gesendet werden müsser
    if(len <= 0) break;
    DatagramPacket packet = new DatagramPacket(buf, len, address, port);
    socket.send(packet);
}
```

Sobald die Verbindung *established* ist, kann der Entwickler Nachrichten auf einem Stream senden:

```
if(conn.isEstablished()) {
    // Stream 0 verwenden, FIN = true
    conn.streamSend(0, "hello".getBytes(), true);
}
```

Zum Empfangen von Nachrichten können alle lesbaren Streams durchlaufen werden:

```
if(conn.isEstablished()) {
    final byte[] buf = new byte[1350];
    for(long streamId : conn.readable()) {
        while(true) {
            final int len = conn.streamRecv(streamId, buf);
            if(len <= 0) break;
            // hier können die gelesenen Daten verarbeitet werden
        }
    }
}
```

Quiche4j bietet ein Low-Level-API für QUIC in Java.

- Quiche stellt nur die QUIC-State-Machine bereit. UDP-Sockets, Event-Loops und Paketversand müssen vom Entwickler selbst bereitgestellt und verwaltet werden, z.B. über regelmäßige ‘recv()‘- und ‘send()‘-Loops.
- Sobald die Verbindung etabliert ist (‘conn.isEstablished()‘), können über ‘streamSend(streamId, ...)‘ Nachrichten auf bestimmten Streams geschrieben werden.
- Mit ‘conn.isEstablished()‘ und ‘conn.readable()‘ können alle lesbaren Streams durchlaufen werden, um ausstehende Nachrichten über ‘stream-Recv(streamId, buf)‘ zu empfangen und zu verarbeiten.

3.1.3 Netty QUIC

Zuerst wird der SSL-Kontext für den Client erstellt und das Netty-Eventloop-Group initialisiert:

```
QuicSslContext context =
    QuicSslContextBuilder.forClient()
        .trustManager(InsecureTrustManagerFactory.INSTANCE)
        .build();

NioEventLoopGroup group = new NioEventLoopGroup(1);
```

Anschließend wird der QUIC-Codec konfiguriert und ein Channel für die Verbindung erstellt:

```
ChannelHandler codec = new QuicClientCodecBuilder()
    .sslContext(context)
    .build();

Bootstrap bs = new Bootstrap();
Channel channel = bs.group(group)
    .channel(NioDatagramChannel.class)
    .handler(codec)
    .bind(0).sync().channel();
```

Die QUIC-Verbindung wird aufgebaut:

```
QuicChannel quicChannel = QuicChannel.newBootstrap(channel)
    .remoteAddress(new InetSocketAddress("127.0.0.1", 7000))
    .connect()
    .get();
```

Sobald die Verbindung steht, können Streams erstellt und Daten gesendet bzw. empfangen werden:

```
QuicStreamChannel streamChannel =
    quicChannel.createStream(
        QuicStreamType.BIDIRECTIONAL,
        new ChannelInboundHandlerAdapter() {
            @Override
            public void channelRead(ChannelHandlerContext ctx, Object msg) {
                ByteBuf buf = (ByteBuf) msg;
                System.out.println(buf.toString(CharsetUtil.US_ASCII));
                buf.release();
            }
        }
    ).sync().getNow();

streamChannel.writeAndFlush(
    Unpooled.copiedBuffer("Hello QUIC\r\n", CharsetUtil.US_ASCII)
);
```

Netty QUIC integriert QUIC in das Netty-Eventloop-Modell.

- Die Verbindung wird automatisch über ‘connect()‘ aufgebaut, inklusive Handshake.
- Streams werden über Netty-typische ‘QuicStreamChannel‘s bereitgestellt.
- Das Lesen und Schreiben von Nachrichten erfolgt über die Channel-Pipeline.
- UDP-Socket und Eventloop werden von Netty verwaltet, sodass der Entwickler sich primär auf Stream-Logik konzentrieren kann.

3.2 Server-Implementierung

3.2.1 Kwik

Für einen Kwik-basierten QUIC-Server müssen zunächst zwei Kernklassen implementiert werden: eine Factory, die neue Verbindungen erzeugt, und die konkrete Connection-Klasse, die Streams akzeptiert und verarbeitet. Zuerst die Factory-Klasse, die bei neuen eingehenden Verbindungen eine ‘MyQuicConnection‘ erstellt:

```
public class MyQuicFactory implements ApplicationProtocolConnectionFactory {
    @Override
    public ApplicationProtocolConnection createConnection(
            String protocol,
            QuicConnection quicConnection) {
        return new MyQuicConnection(quicConnection);
    }
}
```

Dann die konkrete Connection-Klasse, die vom Framework aufgerufen wird, wenn ein Peer einen Stream startet. Hier kann die eigentliche Verarbeitung der Daten stattfinden.

```
public class MyQuicConnection implements ApplicationProtocolConnection {
    private final QuicConnection quicConnection;

    public MyQuicConnection(QuicConnection quicConnection) {
        this.quicConnection = quicConnection;
    }

    @Override
    public void acceptPeerInitiatedStream(QuicStream stream) {
        // do stuff with stream
    }
}
```

Abschließend muss der Server konfiguriert und gestartet werden. Dazu werden Keystore, Logger, unterstützte QUIC-Versionen und weitere Parameter gesetzt. Anschließend wird die Factory für das gewünschte Application Protocol registriert und der Server gestartet:

```
KeyStore keystore = KeyStore.getInstance("PKCS12");
try (FileInputStream keystoreStream = new FileInputStream(new File(KEYSTORE_PATH)
        .getAbsolutePath())) {
    keystore.load(keystoreStream, KEYSTORE_PASSWORD.toCharArray());
}

Logger log = new SysOutLogger();
List<QuicConnection.QuicVersion> supportedVersions = new ArrayList<>();
supportedVersions.add(QuicConnection.QuicVersion.V2);

ServerConnectionConfig config = ServerConnectionConfig.builder()
        .build();

ServerConnector connector = ServerConnector.builder()
        .withPort(7000)
        .withSupportedVersions(supportedVersions)
        .withKeyStore(keystore, KEY_ALIAS, KEYSTORE_PASSWORD.toCharArray())
        .withConfiguration(config)
        .withLogger(log)
        .build();

connector.registerApplicationProtocol("quic", new MyQuicFactory(log));
connector.start();
```

3.2.2 Quiche/Quiche4j

Bei Quiche/Quiche4j ist der Serveraufbau sehr ähnlich zum Client. Der einzige Unterschied besteht darin, dass anstelle von ‘connect()‘ die Methode ‘accept()‘ verwendet wird, um eingehende Verbindungen zu akzeptieren.

```
final Config config = new ConfigBuilder(Quiche.PROTOCOL_VERSION).build();
final byte[] connId = Quiche.newConnectionId();

final Connection conn = Quiche.accept(sourceConnId, originalDestinationId, config)
```

Wie beim Client müssen auch hier die Loops zum Empfangen und Senden von Paketen vom Entwickler bereitgestellt und ausgeführt werden. Der En-

twickler ist dafür verantwortlich, den UDP-Socket zu verwalten und die QUIC-State-Machine regelmäßig zu aktualisieren.

3.2.3 Netty QUIC Server

Für einen Netty-basierten QUIC-Server müssen zunächst TLS-Zertifikate und der QUIC-SSL-Kontext erstellt werden. In diesem Beispiel wird ein selbstsigniertes Zertifikat verwendet:

```
SelfSignedCertificate cert = new SelfSignedCertificate();
QuicSslContext context = QuicSslContextBuilder.forServer(
        cert.privateKey(), null, cert.certificate())
        .build();
```

Anschließend wird das QUIC-Codec für den Server erstellt und ein Stream-Handler registriert.

```
NioEventLoopGroup group = new NioEventLoopGroup(1);

ChannelHandler codec = new QuicServerCodecBuilder()
    .sslContext(context)
    .streamHandler(new ChannelInitializer<QuicStreamChannel>() {
        @Override
        protected void initChannel(QuicStreamChannel ch) {
            ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                @Override
                public void channelRead(ChannelHandlerContext ctx, Object msg) {
                    // do stuff with stream
                    ByteBuf buf = (ByteBuf) msg;
                    buf.release();
                }
            });
        }
    })
    .build();
```

Zum Schluss wird der Server gestartet, der UDP-Socket gebunden und auf eingehende Verbindungen gewartet:

```
try {
    Bootstrap bs = new Bootstrap();
    Channel channel = bs.group(group)
        .channel(NioDatagramChannel.class)
        .handler(codec)
        .bind(9999).sync().channel();

    channel.closeFuture().sync();
} finally {
    group.shutdownGracefully();
}
```

3.3 State- und Event-Transparenz

Obwohl QUIC vollständig im Userspace implementiert ist, gewähren die betrachteten Java-Frameworks nur begrenzte Einblicke in den aktuellen internen Zustand einer Verbindung. Ein direkter Zugriff oder eine Manipulation der QUIC-State-Machine ist in der Regel nicht vorgesehen. Stattdessen ähneln sich die Frameworks in den Abstraktionen, die sie dem Entwickler zur Verfügung stellen.
Alle drei Implementierungen stellen Objekte zur Repräsentation von Verbindungen und Streams bereit und ermöglichen es, über Callback-Mechanismen oder Event-Handler über den erfolgreichen Aufbau oder das Beenden einer Verbindung. Darüber hinaus bieten die Frameworks Zugriff auf Statistik-Objekte, die grundlegende Metriken zur Überwachung der Verbindung enthalten.
Diese Statistiken umfassen typischerweise Informationen über gesendete und empfangene Daten beispielsweise:

- empfangene Pakete
- gesendete Pakete
- verlorene Pakete
- geschätzte Round-Trip-Time
- aktuelles Congestion Window
- geschätzte Übertragungsrate

Damit ermöglichen die Frameworks eine Beobachtung der Übertragungsqualität, ohne jedoch eine detaillierte Kontrolle über die internen Protokollzustände offenzulegen. Diese Designentscheidung vereinfacht die Nutzung der APIs, schränkt jedoch die Möglichkeiten für tiefgreifende Protokollmanipulationen ein.

4 Fazit

4.1 Java-Frameworks

Die drei verglichenen Java-QUIC-Implementierungen zeigen unterschiedliche Abstraktionsgrade und Integrationsaufwände:

- **Kwik:** Bietet eine High-Level-API, die Handshake, Socket-Management und Stream-Verwaltung automatisch übernimmt. Ideal für schnelle Implementierungen ohne tiefe Netzwerkkenntnisse.
- **Quiche/Quiche4j:** Stellt lediglich die QUIC-State-Machine bereit. Der Entwickler muss UDP-Socket, Eventloop und Handshake selbst implementieren. Maximale Flexibilität, aber hoher Aufwand.
- **Netty QUIC:** Integriert QUIC in das Netty-Eventloop-Modell. Handshake und UDP-Transport werden größtenteils verwaltet, Streams werden über Netty-Channels bereitgestellt. Gut geeignet für Anwendungen, die bereits auf Netty basieren.

Zusammenfassend lässt sich festhalten, dass Kwik sich für übersichtliche Client- und Serveranwendungen eignet, bei denen eine klare QUIC-Abstraktion und eine einfache Stream-Verarbeitung im Vordergrund stehen. Netty QUIC eignet sich besonders für Anwendungen, die bereits im Netty-Ökosystem umgesetzt sind oder zukünftig darauf aufbauen sollen. Quiche4J richtet sich hingegen an Szenarien, in denen Kontrolle über den QUIC-Datenfluss erforderlich ist oder bereits eigene UDP-Datagramm- und Eventloop-Implementierungen vorhanden sind.

4.2 QUIC

QUIC ist ein modernes Transportprotokoll, das die grundlegenden Eigenschaften von TCP übernimmt und diese um zusätzliche Mechanismen erweitert. Durch die Realisierung auf Basis von UDP kann QUIC ohne Anpassungen an bestehender Netzwerkinfrastruktur eingesetzt werden, da weder Router noch Endgeräte auf Kernel- oder Hardwareebene verändert werden müssen.
Ein zentrales Merkmal von QUIC ist die Implementierung im User Space. Dadurch lassen sich neue Protokollfunktionen schneller entwickeln und ausrollen als bei klassischen Kernel-basierten Transportprotokollen. Beispiele

hierfür sind das native Stream-Multiplexing ohne Head-of-Line-Blocking sowie die enge Kopplung von Transport- und Sicherheitsmechanismen, wodurch Verschlüsselung nicht als zusätzlicher Layer, sondern als integraler Bestandteil des Protokolls umgesetzt wird.
Insgesamt zeigt sich, dass QUIC viele Einschränkungen von TCP adressiert, ohne dessen Konzepte aufzugeben.

References

[1] netty/netty-incubator-codec-quic. original-date: 2020-11-06T15:51:08Z.

[2] Saleh Alawaji. IETF QUIC v1 design.

[3] Peter Doornbosch. ptrd/kwik. original-date: 2019-12-18T20:46:54Z.

[4] Jana Iyengar and Ian Swett. QUIC loss detection and congestion control. Num Pages: 30.

[5] Oleksii Kachaiev. kachayev/quiche4j. original-date: 2020-09-05T18:49:34Z.

[6] Puneet Kumar. QUIC (quick UDP internet connections) – a quick study.