1 Einleitung

Dieses Dokument dient dazu, drei Open-Source-Implementierungen des QUIC-Protokolls für Java miteinander zu vergleichen: Kwik, Quiche/quiche4j sowie das QUIC-Modul von Netty (netty-incubator-codec-quic). Ziel ist es, die unterschiedlichen Abstraktionsebenen, Architekturansätze und den Integrationsaufwand der jeweiligen Bibliotheken gegenüberzustellen.

2 Client Implementierungsvergleich

2.1 Kwik

Zunächst muss eine Verbindung aufgebaut werden. Dazu wird ein ‘QuicClient-Connection‘ 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.

2.2 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.

2.2.1 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;
}
```

2.2.2 Send-Loop

```
while(true) {
    final int len = conn.send(buf); // fragt, welche Pakete gesendet werden müssen
    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 ‘streamRecv(streamId, buf)‘ zu empfangen und zu verarbeiten.

2.2.3 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 Server-Implementierung

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

3.3 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();
}
```

4 Fazit

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 gilt: Kwik für einfache Clients, Netty für Netty-basierte Systeme mit Bedarf an Eventloop-Integration, Quiche für maximale Kontrolle über die QUIC-Protokollabläufe.

5 Quellen

Alle Beispiele sind so nahe wie möglich aus den jeweiligen Dokumentationen hergeleitet:

- **Kwik:** https://github.com/ptrd/kwik
- **Quiche/quiche4j:** https://github.com/kachayev/quiche4j
- **Netty QUIC:** https://github.com/netty/netty-incubator-codec-quic/blob/main/codec-native-quic/src/test/java/io/netty/incubator/codec/quic/example/QuicServerExample.java