Implementierung eines QUIC-Demonstrators mit dem Java-Framework Kwik

Praktische Umsetzung eines QUIC-basierten Client-Server-Systems

Kai Penazzi
AI, EMI
HS-Offenburg

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

Contents

1 Einleitung

Dieses Dokument beschreibt die Implementierung eines Demonstrators für das QUIC-Transportprotokoll unter Verwendung des Java-Frameworks Kwik. Ziel des Demonstrators ist es, die Funktionsweise der Kwik-API zu verstehen und zentrale Konzepte von QUIC anhand einer praktischen Anwendung nachzuvollziehen.
Der Demonstrator ist als klassisches Client-Server-System aufgebaut, bei dem der Server die QUIC-Verbindungen verwaltet und der Client über Streams mit dem Server kommuniziert. So können sowohl lokale als auch globale Zustände verwaltet, Nachrichten ausgetauscht und verschiedene Features von QUIC, wie Multi-Streaming und bidirektionale Kommunikation, demonstriert werden

2 Server

2.1 Schlüsselgenerierung

Obwohl der Client im Demonstrator keine Serverauthentifizierung durchführt, benötigt QUIC aufgrund der integrierten TLS 1.3-Verschlüsselung dennoch entsprechendes Schlüsselmaterial. Das folgende Bash-Skript wurde von ChatGPT erstellt und erzeugt das Schlüsselmaterial.

```
#!/bin/bash

# Generiert von ChatGPT
# Erstelle ein Self-Signed Zertifikat mit SAN
openssl req -x509 -nodes -days 365 \
  -newkey rsa:2048 \
  -keyout server.key \
  -out server.crt \
  -subj "/CN=localhost" \
  -addext "subjectAltName=DNS:localhost,IP:127.0.0.1"

# Exportiere in PKCS12-Keystore mit Alias 'kwikserver'
openssl pkcs12 -export \
  -in server.crt \
  -inkey server.key \
  -name kwikserver \
  -out keystore.p12 \
  -passout pass:keystorepass

# Optional: Prüfe Inhalt des Keystores
echo "Folgende Einträge befinden sich im Keystore:"
keytool -list -keystore keystore.p12 -storetype PKCS12 -storepass
    keystorepass
```

2.2 GlobalState

Für den Demonstrator wird ein globaler Zustand implementiert, der von allen aktiven QUIC-Verbindungen gemeinsam genutzt wird. Dies simuliert in einer realen Anwendung eine einfache Persistenz, auf die mehrere Verbindungen zugreifen können. In diesem Fall ist der globale Zustand nur ein einfacher String, der gelesen und geschrieben werden kann.

```
public class MyGlobelState {
    private String state = "";

    public String getState() {
        return this.state;
    }

    public void setState(String newState) {
        this.state = newState;
    }
}
```

2.3 MyConnectionListener

`MyConnectionListener` implementiert die Schnittstelle `ConnectionListener` und überwacht wichtige Ereignisse der QUIC-Verbindung. Die Überwachung beschränkt sich dabei auf den Aufbau und die Beendigung der Verbindung, da das Interface keine weiteren Methoden bereitstellt.

```
public class MyConnectionListener implements ConnectionListener {
    private Logger log;

    public MyConnectionListener(Logger logger) {
        this.log = logger;
    }

    @Override
    public void connected(ConnectionEstablishedEvent event) {
        log.info("connection established");
    }

    @Override
    public void disconnected(ConnectionTerminatedEvent event) {
        log.info("connection terminated: " + event.errorDescription());
    }
}
```

2.4 MyQuicConnection

Die Klasse `MyQuicConnection` stellt die zentrale Verbindungskomponente des Servers dar. Sie implementiert die Schnittstelle `ApplicationProtocolConnection`

des Kwik-Frameworks und wird für jede eingehende QUIC-Verbindung instanziiert.
Die Hauptaufgaben dieser Klasse sind:

- Das setzen des `MyConnectionListener`, die `connected`-Methode des Listeners wird dabei allerdings nicht aufgerufen, da die Verbindung bereits vorher aufgebaut wurde.
- Verwaltung des lokalen Verbindungszustands `ConnectionState` und Zugriff auf den globalen Zustand `MyGlobalState`.
- Akzeptieren von Peers initiierten Streams über `acceptPeerInitiatedStream`.
- Unterscheidung zwischen unidirektionalen und bidirektionalen Streams.
- Verarbeitung von Nachrichten aus bidirektionalen Streams, inklusive Befehlen zur Zustandsabfrage oder -änderung:
  - `get / set` für den lokalen Verbindungszustand
  - `gget / gset` für den globalen Zustand
  - Alle anderen Nachrichten werden einfach zurückgesendet (Echo)
- Versand von Nachrichten über die bidirektionalen Streams.

Die Verarbeitung jedes Streams erfolgt in einem eigenen Thread, um parallele Kommunikation zu ermöglichen.

```
public class MyQuicConnection implements ApplicationProtocolConnection {
    private final QuicConnection quicConnection;
    private final Logger log;
    private String ConnectionState;
    private MyGlobelState globelState;

    public MyQuicConnection(QuicConnection quicConnection, Logger logger,
        MyGlobelState state) {
        this.quicConnection = quicConnection;
        this.quicConnection.setConnectionListener(new
            MyConnectionListener(logger));
        this.log = logger;
        this.ConnectionState = "";
        this.globelState = state;
    }

    @Override
    public void acceptPeerInitiatedStream(QuicStream stream) {
        new Thread(() -> handleClient(stream)).start();
    }
```

```
private void handleClient(QuicStream stream) {
    if (stream.isUnidirectional()) {
        try (InputStream in = stream.getInputStream()) {

            byte[] buf = new byte[4096];
            int len;

            while ((len = in.read(buf)) != -1) {
                if (len == 0)
                    continue;

                String msg = new String(buf, 0, len,
                    StandardCharsets.UTF_8);
                log.info(this.hashCode() + ": stream ID: " +
                    stream.getStreamId() + ": " + msg);
            }

        } catch (Exception e) {
            log.error("stream failed", e);
        }
    }
    if (stream.isBidirectional()) {
        try (InputStream in = stream.getInputStream()) {

            byte[] buf = in.readAllBytes();
            String msg = new String(buf, StandardCharsets.UTF_8);
            log.info(this.hashCode() + ": stream ID: " +
                stream.getStreamId() + ": " + msg);

            String[] parts = msg.split(" ");

            switch (parts[0]) {
                case "get":
                    this.send("state: " + this.ConnectionState, stream);
                    break;

                case "set":
                    try {
                        this.ConnectionState = parts[1];
                        this.send("state: " + this.ConnectionState,
                            stream);
                    } catch (IndexOutOfBoundsException e) {
                        this.send("value is missing: 'set <value>' ",
                            stream);
                    }
                    break;
                case "gget":
                    this.send("global state: " +
                        this.globelState.getState(), stream);
                    break;

                case "gset":
                    try {
                        this.globelState.setState(parts[1]);
                        this.send("global state: " +
                            this.globelState.getState(), stream);
```

```
                        } catch (IndexOutOfBoundsException e) {
                            this.send("value is missing: 'gset <value>' ",
                                stream);
                        }
                        break;
                    default:
                        this.send("echo: " + msg, stream);
                }

            } catch (Exception e) {
                log.error("stream failed", e);
            }
        }

    }

    private void send(String msg, QuicStream stream) {
        try (OutputStream out = stream.getOutputStream()) {
            out.write(msg.getBytes());
            out.close();
        } catch (Exception e) {
            log.error("could not send msg: ", e);
        }

    }
}
```

2.5 QUIC-Factory

Die Klasse `MyQuicFactory` implementiert das Interface `ApplicationProtocolConnectionFact`
und ist für die Erzeugung neuer Verbindungen zuständig. Sie wird vom Kwik-Framework verwendet, sobald eine neue QUIC-Verbindung akzeptiert wird.
Innerhalb der Factory wird ein einzelnes Objekt vom Typ `MyGlobalState` erzeugt und gespeichert. Dieses globale Zustandsobjekt wird bei jeder neu erstellten `MyQuicConnection` an den Konstruktor übergeben, sodass alle Verbindungen denselben globalen Zustand teilen.

```
public class MyQuicFactory implements ApplicationProtocolConnectionFactory {
    private final Logger log;
    private MyGlobelState globelState;

    MyQuicFactory(Logger logger) {
        this.log = logger;
        this.globelState = new MyGlobelState();
    }

    @Override
    public ApplicationProtocolConnection createConnection(String protocol,
        QuicConnection quicConnection) {
        return new MyQuicConnection(quicConnection, this.log,
            this.globelState);
```

```
    }
}
```

2.6 App

Die Klasse `App` dient als Einstiegspunkt der Serveranwendung und übernimmt die Initialisierung des QUIC-Servers.
Zu Beginn wird der zuvor erzeugte PKCS12-Keystore geladen.

```
String KEYSTORE_PATH = "../keystore.p12";
String KEYSTORE_PASSWORD = "keystorepass";
String KEY_ALIAS = "kwikserver";

KeyStore keystore = KeyStore.getInstance("PKCS12");

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

Anschließend wird ein `Logger` konfiguriert, der Laufzeitinformationen wie Verbindungsereignisse, Stream-Daten und Warnungen ausgibt. Ohne einen Logger kann der Server nicht gestartet werden.

```
Logger log = new SysOutLogger();

log.timeFormat(Logger.TimeFormat.Long);
log.logWarning(true);
log.logInfo(true);
log.logStream(true);
```

Die unterstützten QUIC-Versionen werden explizit festgelegt. In diesem Demonstrator wird QUIC Version 1 verwendet.

```
List<QuicConnection.QuicVersion> supportedVersions = new ArrayList<>();
supportedVersions.add(QuicConnection.QuicVersion.V1);
```

Über das `ServerConnectionConfig`-Objekt werden Verbindungsparameter konfiguriert, darunter Zeitlimits, maximale Stream-Anzahlen sowie die Verwendung von Retry-Paketen.

```
ServerConnectionConfig config = ServerConnectionConfig.builder()
        .maxIdleTimeoutInSeconds(5000)
        .maxOpenPeerInitiatedUnidirectionalStreams(10)
        .maxOpenPeerInitiatedBidirectionalStreams(100)
        .retryRequired(true)
        .connectionIdLength(8)
        .build();
```

Der eigentliche QUIC-Server wird anschließend über den `ServerConnector` erstellt. Dabei werden unter anderem der Serverport, die unterstützten Protokollversionen, das Schlüsselmaterial sowie der Logger registriert.

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

Über `registerApplicationProtocol` wird schließlich das Anwendungsprotokoll `quic` mit der zuvor definierten `MyQuicFactory` verknüpft.

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

Mit dem Aufruf von `connector.start()` wird der Server gestartet.

```
connector.start();
```

2.7 Build und Ausführung

Der Server kann problemlos direkt über Gradle mit dem Task `run` gestartet werden:

```
./gradlew server:run
```

Alternativ kann der Server als ausführbare Fat JAR gebaut werden. Hierzu wird das Gradle-Plugin `com.github.gooogler.shadow` verwendet, das sämtliche Abhängigkeiten in eine einzelne ausführbare JAR integriert.

```
./gradlew shadowJar
```

Anschließend kann der Server unabhängig von Gradle direkt über die Java-Laufzeitumgebung gestartet werden:

```
java -jar server/build/libs/server-all.jar
```

3 Client

3.1 MyConnection

Die Klasse `MyConnection` kapselt die QUIC-Client-Funktionalität. Sie ermöglicht den Aufbau einer Verbindung zu einem QUIC-Server, das Öffnen von Streams sowie das Senden und Empfangen von Nachrichten. Dabei werden bidirektionale und unidirektionale Streams unterstützt und es können mehrere parallele Streams gleichzeitig verwaltet werden.

```
public class MyConnection {

    private final URI uri;
    private QuicClientConnection connection;
    private List<QuicStream> streams;

    public MyConnection(String url) {
        this.uri = URI.create("quic://" + url);
        streams = new ArrayList<QuicStream>();
    }

    public void connect() throws Exception {
        connection = QuicClientConnection.newBuilder()
                .noServerCertificateCheck()
                .uri(this.uri)
                .applicationProtocol("quic")
                .maxIdleTimeout(Duration.ofMinutes(5))
                .build();

        connection.connect();
    }

    public String sendResponse(String msg) {
        try {
            QuicStream stream = this.connection.createStream(true);
            OutputStream out = stream.getOutputStream();
            InputStream in = stream.getInputStream();
            out.write(msg.getBytes());
            out.close();
            byte[] buf = in.readAllBytes();
            return new String(buf, StandardCharsets.UTF_8);
        } catch (IOException e) {
            return "could not send Message: " + e.toString();
        }
    }

    public void sendUnidirectional(String msg) {
        try {
            QuicStream stream = this.connection.createStream(false);
            OutputStream out = stream.getOutputStream();
            out.write(msg.getBytes());
            out.close();
        } catch (IOException e) {
            System.out.println("could not send Message: " + e.toString());
        }
    }

    public void send_msg_over(int stream_id, String msg) {
        QuicStream stream = this.streams.get(stream_id);
        try {
            OutputStream out = stream.getOutputStream();
            out.write(msg.getBytes());
            out.flush();
        } catch (Exception e) {
            System.out.println("could not send msg: " + e.toString());
        }
    }
```

```
    public int open_stream() throws IOException {
        QuicStream stream = connection.createStream(false);
        this.streams.add(stream);
        return this.streams.size() - 1;
    }

    public void close_stream(int stream_id) {
        QuicStream stream = this.streams.get(stream_id);
        try {
            OutputStream out = stream.getOutputStream();
            out.close();
        } catch (IOException e) {
            System.out.println("could not close stream id: " + stream_id +
                "\n" + e.toString());
        }
    }

    public void close() {
        for (int i = 0; i < this.streams.size(); i++) {
            this.close_stream(i);
        }
        this.connection.close();
    }

    public void close(long errorCode, String msg) {
        for (int i = 0; i < this.streams.size(); i++) {
            this.close_stream(i);
        }
        this.connection.close(errorCode, msg);
    }

    public String getStats() {
        return this.connection.getStats().toString();
    }
}
```

3.2 MyConnectionHandler

Die Klasse `MyConnectionHandler` verwaltet mehrere QUIC-Clientverbindungen. Sie abstrahiert die Verwaltung mehrerer `MyConnection`-Instanzen und erlaubt das gleichzeitige Erstellen, Verwenden und Schließen mehrerer Verbindungen.

```
public class MyConnectionHandler {
    private List<MyConnection> clients;

    public MyConnectionHandler() {
        this.clients = new ArrayList<MyConnection>();
    }

    public void createConnection(String url) throws Exception {
        MyConnection client = new MyConnection(url);
```

```
        client.connect();
        this.clients.add(client);
    }

    public String sendResponse(int connection_id, String msg) {
        return this.clients.get(connection_id).sendResponse(msg);
    }

    public MyConnection getConnection(int connection_id) {
        return this.clients.get(connection_id);
    }

    public void closeAll() {
        for (MyConnection client : clients) {
            client.close();
        }
    }
}
```

3.3 Commands

Die Client-Anwendung nutzt Picocli, um die Benutzereingaben als Subcommands bereitzustellen. Jeder Command implementiert entweder das `Callable`- oder das `Runnable`-Interface und kann eigene Parameter definieren, die Picocli automatisch aus den eingegebenen Strings parsed.
Ein Beispiel hierfür ist der `OpenStream`-Command. Dieser erwartet als Parameter die `connection_id`, die angibt, über welche bestehende QUIC-Verbindung ein neuer Stream geöffnet werden soll. Innerhalb des Commands greift die Implementierung auf `MyConnectionHandler` zu um einen neuen Stream zu eröffnen. Auf diese Weise abstrahiert jeder Command die direkte Interaktion mit den QUIC-Verbindungen und ermöglicht eine übersichtliche, modulare Steuerung des Clients.

```
@Command(name = "open_stream", description = "open a stream")
public class OpenStream implements Callable<Integer> {
    private final MyConnectionHandler handler;

    @Parameters(index = "0", description = "Connection ID")
    public int connection_id;

    public OpenStream(MyConnectionHandler handler) {
        this.handler = handler;
    }

    @Override
    public Integer call() throws IOException {
        return this.handler.getConnection(this.connection_id).open_stream();
    }
}
```

3.4 App

Die Klasse `App` dient als Kommandozeilen-Interface für den QUIC-Client-Demonstrator. Sie nutzt `picocli` zur Verarbeitung von Subcommands.
Die Verwaltung der QUIC-Verbindungen erfolgt über `MyConnectionHandler`. Benutzereingaben werden in einer Endlosschleife eingelesen und anschließend an die registrierten Subcommands weitergeleitet. Die Zerlegung des Eingabestrings in einzelne Argumente erfolgt über die Methode `splitArgs`, die von dem KI-Modell Gemini generiert wurde und dafür sorgt, dass auch Argumente in Anführungszeichen korrekt als ein String behandelt werden.

```
@Command(name = "", subcommands = {})
public class App {
    public static void main(String[] args) throws Exception {
        MyConnectionHandler handler = new MyConnectionHandler();

        Scanner scanner = new Scanner(System.in);

        CommandLine cmd = new CommandLine(new App());
        cmd.addSubcommand("connect", new Connect(handler));
        cmd.addSubcommand("disconnect", new Disconnect(handler));
        cmd.addSubcommand("openStream", new OpenStream(handler));
        cmd.addSubcommand("closeStream", new CloseStream(handler));
        cmd.addSubcommand("sendOver", new SendOver(handler));
        cmd.addSubcommand("getState", new GetState(handler));
        cmd.addSubcommand("setState", new SetState(handler));
        cmd.addSubcommand("stats", new Stats(handler));
        cmd.addSubcommand("setGState", new SetGlobalState(handler));
        cmd.addSubcommand("getGState", new GetGlobalState(handler));

        while (true) {
            System.out.print("> ");
            String input = scanner.nextLine();

            if (input.equals("exit"))
                break;

            String[] parsedArgs = splitArgs(input);
            cmd.execute(parsedArgs);
        }

        handler.closeAll();
    }

    // von Gemini generierte Methode zum Parsen von Anführungszeichen
    public static String[] splitArgs(String string) {
        List<String> matchList = new ArrayList<String>();
        Pattern regex = Pattern.compile("[^\\s\"]+|\"([^\"]*)\"");
        Matcher regexMatcher = regex.matcher(string);
        while (regexMatcher.find()) {
            if (regexMatcher.group(1) != null)
                matchList.add(regexMatcher.group(1));
            else
                matchList.add(regexMatcher.group());
```

```
        }
        return matchList.toArray(new String[0]);
    }
}
```

3.5 Build und Ausführung

Da der Client als Kommandozeilenwerkzeug implementiert ist und kontinuierlich Benutzereingaben über `System.in` verarbeitet, kommt es zu Fehlern bei der ausführung mit `gradlew client:run`.
Aus diesem Grund muss der Client als ausführbare JAR-Datei gestartet werden. Hierfür wird ebenfalls das Gradle-Plugin `com.github.gooogler.shadow` verwendet.

```
./gradlew client:shadowJar
java -jar client/build/libs/client-all.jar
```

4 Fazit

Kwik abstrahiert die Logik des QUIC-Protokolls weitgehend und stellt der Anwendung eine höherwertige, stream-orientierte API zur Verfügung. Dadurch kann sich die Anwendungsentwicklung auf logische Datenströme konzentrieren, ohne sich mit Details wie Paketnummern, Retransmissions oder Congestion Control befassen zu müssen.
Diese starke Abstraktion vereinfacht die Nutzung erheblich, führt jedoch auch dazu, dass transportnahe Ereignisse – etwa Benachrichtigungen über Paketverluste oder interne Retransmissions – nicht über die öffentliche API zugänglich sind. Der `ConnectionListener` bietet lediglich grundlegende Verbindungsereignisse wie Verbindungsaufbau und -abbau. Für Analyse- oder Forschungszwecke könnte eine erweiterte Ereignisschnittstelle sinnvoll sein, die tiefere Einblicke in das Verhalten der Transportebene erlaubt.