

H.264/MPEG-4 SVC

Skalierbare Videokodierung

Dieses Dokument beschreibt den Einsatz der Referenzsoftware „JSVM“ zur Dekodierung & Übertragung von skalierbaren H.264 – enkodierten Inhalten via Netzwerk.

Fachhochschule Düsseldorf – Philipp Ludwig – 538382
Wintersemester 2010/2011



INHALTSVERZEICHNIS

1. Grundlegendes.....	1
1.1 Aufbau eines H.264/MPEG-4 SVC-Streams.....	1
1.2 Nützliche Programme.....	2
2. Installation	3
2.1 Systemvoraussetzungen	3
2.2 Modifikation der JSVM-Referenzsoftware	3
2.3 Kompilierung unter Linux	4
2.4 Kompilierung unter Windows	5
3. Verwendung der Programme.....	7
3.1 Decoder-Testanwendung	7
3.2 Server-Anwendung	8
3.3 Client-Anwendung.....	10
4. Entwickeln auf Basis von JSVM	12
4.1 Grundlegendes.....	12
4.2 Datentypen.....	12
4.3 Wichtige Klassen	13
4.3.1 – Die Klasse „CreatorH264AVCDecoder“	13
4.3.2 – Die Klasse „H264PacketAnalyzer“	14
4.3.3 – Die Klasse „ReadbitStreamFile“	15
5. Der interne Aufbau der Programme.....	16
5.1 Die Server-Anwendung	16
5.2 Die Client-Anwendung	18
6. Funktionsweise der Programme.....	19
6.1 Kommunikation zwischen Server und Client.....	19
6.1.1 Verbindungsaufbau.....	19
6.1.2 Client-Behandlung.....	20
6.1.3 Übertragung der Daten.....	20
6.1.4 Wechsel der Qualität während der Übertragung	21
7. Integration des Quellcodes in neue Projekte	24
7.1 Verwendung der Klasse C_Decoder.....	24
7.2 Verwendung der Klasse C_ReadFile	25
7.3 Verwendung der Serverklassen	27
7.3.1 Starten des Servers.....	27

7.3.2 Verwendung von alternativen Quellen	28
7.4 Verwendung der Klasse C_Client	29
Anhang A: Abbildungsverzeichnis.....	30
Anhang B: Inhalt der beiliegenden CD	31
Anhang C: Nützliche Links.....	32

H.264/MPEG-4 SVC

Skalierbare Videokodierung

Das Ziel dieses Projektes war, zu untersuchen, in wie weit die Möglichkeiten der skalierbaren Videokodierung des H.264-Standards für den praktischen Anwendungsfall im Rahmen einer Netzwerkübertragung geeignet ist. Als Basis wurde die Referenzsoftware „JSVM“ verwendet, welche eine beispielhafte Implementierung bietet.

Mit der im Rahmen dieses Projektes entwickelten Software ist es gelungen, H.264-kodiertes Videomaterial via Netzwerk zu übertragen und zu dekodieren. Die dabei erzielte Performance war allerdings alles andere als ausreichend, was man wohlmöglich nur mit massiven Verbesserungen der Referenzsoftware erreichen könnte.

Nichtsdestotrotz wurden konkrete Ergebnisse erzielt und Teile des Quelltextes können ohne großen Aufwand für weitere Projekte wiederverwendet werden.

Hinweise für Quelltextleser:

Sämtliche Bezeichnungen der einzelnen Objekte, Klassen und Variablen folgen einer gewissen Konvention:

- Klassennamen beginnen immer mit „C_“.

Beispiel:

C_MainWindow

- Member-Variablen von Klassen beginnen immer mit einem kleinen „m“.

Beispiel:

*C_Server *mServer;*

- Namen von Funktionsargumenten beginnen immer mit einem kleinen „a“.

Beispiel:

void addEntry(QString aText, int aValue, QColor aColor);

In gewissen Fällen, wie z.B. bei einstelligen Namen, werden diese Konventionen nicht immer befolgt. Diese Ausnahmen sind jedoch äußerst selten und insgesamt wird die Lesbarkeit des Codes deutlich erhöht.

1. GRUNDLEGENDES

Um große Teile dieser Dokumentation sowie die damit verbundene Software bzw. deren Quelltext verstehen zu können, ist es notwendig, zu wissen, wie ein H.264-kodierter Videostream aufgebaut ist. Zwar kann man sich diese Grundlagen auch mit Hilfe von anderen Quellen, wie z.B. Wikipedia, aneignen (und dies in einem wesentlich umfangreicheren Maße, als es im Rahmen dieses Textes möglich ist), allerdings sollen der Einfachheit halber zumindest die Grundzüge erläutert werden. Wörter wie *frame*, *bitrate* und weitere gängige Begriffe aus der Videokodierung werden als bekannt vorausgesetzt.

1.1 Aufbau eines H.264/MPEG-4 SVC-Streams

Ein H.264/MPEG-4 SVC-Stream kann grundsätzlich in eine Menge von Frames unterteilt werden. Diese umfasst wie üblich B-, P- oder I-Frames. Ein Bild (entsprechend ein oder zwei Frames) wird innerhalb des Bitstreams durch ein oder mehrere sog. *Slices* repräsentiert. Ein Slice wiederum ist eine Menge von einem oder mehreren Makroblöcken.

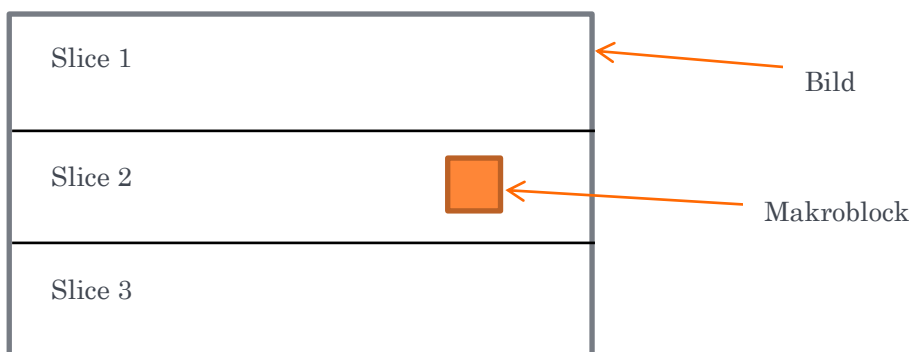


Abbildung 1: Aufbau eines Bildes mit mehreren Slices

Beinhaltet die Bytemenge eines Slices neben der reinen Bildinformation zusätzlich Metadaten mit Angaben über die Bitrate, die Qualitätsstufe oder den Frametyp, handelt es sich um eine sog. *NALU*, eine *Network Abstraction Layer Unit*. Eine Menge von NALUs, die einen ganzen Frame bilden, wird *AccessUnit*, kurz *Unit*, genannt.

Eine Unit kann in eine Menge von Paketen unterteilt werden. Diese werden durch einen sog. *Startcode* voneinander getrennt, welcher aus einer beliebigen Anzahl von Nullen (mindestens aber zwei) und einer 1 zusammengesetzt wird. Beispiele für einen Startcode wären 00001 oder 001 oder 00000001. Abbildung 2 zeigt den Ausschnitt einer hexadezimalen Darstellung eines H.264-Bitstreams; der Startcode ist deutlich zu sehen.

```
FC 61 AE 14 F2 46 81 DC 97 47 E0 00 00 00 01 41 9A 20 86 49
LC 0T WE T+ L5 +0 0T DC 2\ +\ 50 00 00 00 01 +T 2W 50 00 +2
```

Abbildung 2: Startcode im H.264-Bitstream

1. Grundlegendes

Jedes Paket beinhaltet neben den eigentlichen Bildinformationen zusätzliche Metadaten, wie beispielsweise die Nummer des zugehörigen *Layers*. „Layer“ bezeichnet eine Qualitätsstufe einer H.264/MPEG-4 SVC-kodierten Datei, wobei das sog. *BaseLayer* mit der Nummer 0 zwingend erforderlich ist, alle weiteren sind optional¹.

Um das Layer n aus einem Stream zu extrahieren, benötigt man alle Pakete, deren $id \leq n$ ist. Sind die Pakete innerhalb des Streams beispielsweise wie in Abbildung 3 angeordnet (die Zahlen geben die Layer-ID an) und das Layer 2 soll extrahiert werden, müssen das vierte und das siebte Paket verworfen werden. Aus den restlichen wird der endgültige Stream erzeugt.



Abbildung 3: Beispielhafte Verteilung von Paketen in einem Stream.

Dies stellt den Vorteil der skalierbaren Codierung dar: Statt mehrere Dateien mit verschiedenen Qualitäten bereitstellen zu müssen, was eine große Redundanz erzeugen würde, werden verschiedene Layer eingesetzt, welche jeweils nur die Videodaten enthalten, welche nicht in den niedrigeren Layern vorgekommen wären.

1.2 Nützliche Programme

Folgende Programme waren bei der Arbeit an diesem Projekt äußerst nützlich und werden es sicherlich auch bei ähnlichen Problemstellungen sein:

- *pyUV* – ein plattformunabhängiger Player für YUV-Rohdaten.
http://dsplab.diei.unipg.it/pyuv_raw_video_sequence_player
- *mp4box* – ein ebenfalls plattformunabhängiges Programm, welches H.264-Rohdaten in einen MP4-Container verpackt, welcher dann von allen gängigen Playern abgespielt werden kann.
<http://gpac.sourceforge.net>
- *Yamb* – eine GUI für mp4box.
<http://yamb.unite-video.com/>

¹ Das BaseLayer kann – muss aber nicht – AVC-kompatibel sein, wodurch alle gängigen Player dieses dann abspielen könnten. Im Rahmen dieses Projektes wurden nur Videosequenzen eingesetzt, welche AVC-kompatible BaseLayer enthielten.

2. INSTALLATION

2.1 Systemvoraussetzungen

Die in diesem Dokument vorgestellten Programme (Client und Server sowie Decoder-Testanwendung) wurden mit Qt 4.7.1 entwickelt, sollte aber auch mit jeder anderen Unterversion von Qt 4 lauffähig sein. Die Client-Anwendung ist aufgrund der Verwendung von Pipes nicht unter Windows ausführbar; der Server und die Decoder-Testanwendung können aber problemlos unter Windows verwendet werden.

Zur Kompilierung wird neben dem Qt-SDK zusätzlich die JSVM-Referenzsoftware benötigt. Während der Arbeit an diesem Projekt haben sich die möglichen Bezugsquellen mehrfach geändert, mit Stand vom 9. März 2011 kann die Software unter http://ip.hhi.de/imagecom_G1/savce/downloads/SVC-Reference-Software.htm bezogen werden.

Zur Laufzeit benötigen der Client und die Decoder-Testanwendung das Programm *ffmpeg* um die dekodierten Daten in einen AVI-Stream einzubinden, welcher dann von herkömmlichen Videoplayern abgespielt werden kann.

2.2 Modifikation der JSVM-Referenzsoftware

Obwohl dies zunächst vermieden werden sollte, war es schließlich unumgänglich, die JSVM-Referenzsoftware zu modifizieren, um konkrete Ergebnisse zu erzielen. Wenn auch die entsprechende Version auf der beiliegenden CD zu finden ist, so sollen doch die wenigen Änderungen der Vollständigkeit halber hier aufgeführt werden.

In der Datei *GOPDecoder.cpp*, zu finden im Unterverzeichnis *H264Extension/src/lib/H264AVCDecoderLib*, müssen die Zeilen 848 und 3008 auskommentiert werden:

```

844.     ErrVal
845.     DecodedPicBuffer::xDeleteData()
846.     {
847.         //===== check picture buffer list =====
848.         //ROF( m_cPicBufferList.empty() );
849.         //===== delete DPB units =====
850.         (...)
3002.         if( m_apabBaseModeFlagAllowedArrays[1] )
3003.         {
3004.             delete [] m_apabBaseModeFlagAllowedArrays [1];
3005.             m_apabBaseModeFlagAllowedArrays[1] = 0;
3006.         }
3007.
3008.         //ROF( m_cSliceHeaderList.empty() );
3009.
3010.         return Err::m_nOK;
3011.         (...)

```

2. Installation

Diese Änderung ist notwendig, damit die Qualität während der Übertragung problemlos gewechselt werden kann.

Eine optionale Veränderung kann außerdem in der Datei *Typedefs.h* vorgenommen werden, zu finden in *H264Extension/include*. Wird hier der Datentyp *double* durch *float* ersetzt, kann die Dekodierung etwas schneller durchgeführt werden:

```
188.     #if !(defined MSYS_TYPE_DOUBLE)
189.     #define MSYS_TYPE_DOUBLE float
190.     #endif
191.     #if !(defined MSYS_NO_TYPE_DOUBLE )
192.         typedef MSYS_TYPE_DOUBLE Double;
193.     #endif
```

An dieser Stelle sei noch anzumerken, dass durch diese Veränderung keine Verschlechterung der Bildqualität zu bemerken ist. Allerdings kann das bei JSVM mitgelieferte Testprogramm „BitStreamExtractor“ nicht mehr kompiliert werden.

2.3 Kompilierung unter Linux

Sollte auf Basis des Quelltextes ein neues Qt-Projekt angelegt werden, so sind dabei einige Dinge zu beachten. Neben den üblichen Qt-Modulen *gui* und *core* werden außerdem *network* und *phonon* benötigt, letzteres lediglich zur Wiedergabe der dekodierten Daten in der Client-Anwendung und der Decoder-Testanwendung. Des Weiteren muss für manche Versionen des gcc zur erfolgreichen Kompilierung des JSVM-Anteils der Parameter *-std=c++0x* gesetzt sein, zusätzlich ist eine Definition der Makros *MSYS_LINUX*, *_LARGEFILE64_SOURCE*, *_FILE_OFFSET_BITS=64* sowie *MSYS_UNIX_LARGEFILE* erforderlich. Außerdem müssen die entsprechenden JSVM-Bibliotheken eingebunden werden.

Zum besseren Verständnis folgt ein Auszug aus der *.pro*-Datei:

```
QT      += core gui network phonon
DEFINES += MSYS_LINUX \
    _LARGEFILE64_SOURCE \
    _FILE_OFFSET_BITS=64 \
    MSYS_UNIX_LARGEFILE

LIBS = -lH264AVCVideoIoLibStatic -lH264AVCDecoderLibStatic -lH264AVCCommonLibStatic

QMAKE_CFLAGS_DEBUG += -std=c++0x
QMAKE_CXXFLAGS_DEBUG += -std=c++0x
```

2.4 Kompilierung unter Windows

Aufgrund der Tatsache, dass der gdb beim Debuggen von Programmen, welche die JSVM-Bibliotheken verwenden, sehr stark dazu neigt, CPU-Auslastungen jenseits der 98% zu erreichen und zur Durchführung eines einzelnen Programmschritts ca. 5 Minuten braucht, ist es anzuraten, nach Möglichkeit das Microsoft Visual Studio zumindest zur Entwicklung des Servers sowie der Decoder-Testanwendung zu verwenden. Das Debuggen ist dort mit gewohnter Geschwindigkeit möglich, was vermutlich auch damit zusammenhängt, dass JSVM mit dieser IDE entwickelt wurde. Mit Hilfe des entsprechend Addins für das Visual Studio² kann außerdem Qt bequem eingebunden werden. Sollte ein neues Projekt mit Verwendung der JSVM-Bibliotheken angelegt werden, sind einige Einstellungen notwendig, die hier erläutert werden sollen.

Zunächst ist die notwendige Konfiguration bezüglich der zu verwendenden Bibliotheken sowie der Include-Verzeichnisse zu tätigen. Dies sollte bekannt sein. Danach müssen – wie unter Linux – einige Makros definiert werden, was in den Projekteigenschaften unter *Konfigurationseigenschaften* → *C/C++* → *Präprozessor* vorgenommen werden kann.

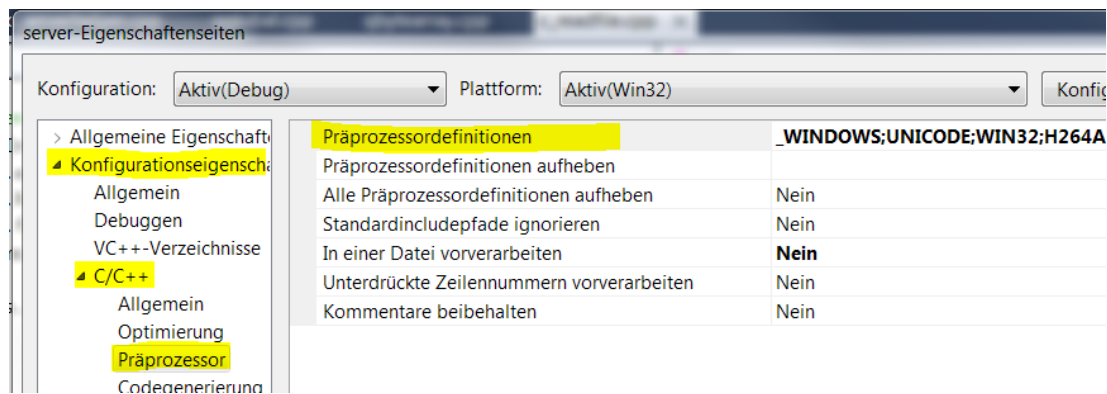


Abbildung 4: Einstellen der Präprozessordefinitionen im Visual Studio

Folgende Makros müssen in die Liste aufgenommen werden:

- H264AVCVIDEOIOLIB_LIB
- H264AVCCOMMONLIB_LIB
- H264AVCDECODERLIB_LIB
- H264AVCENCODERLIB_LIB (sofern die Encoder-Bibliothek verwendet wird)

Eine weitere notwendige Einstellung befindet sich in der Eigenschaftenseite des Linker. Die Standardbibliotheken msvcprtd.lib sowie libcmtd.lib müssen vom Linkingprozess ausgeschlossen werden (s.a. Abbildung 5), da es sonst zu Konflikten kommt.

² Zu beziehen unter <http://qt.nokia.com/downloads/visual-studio-add-in>

2. Installation

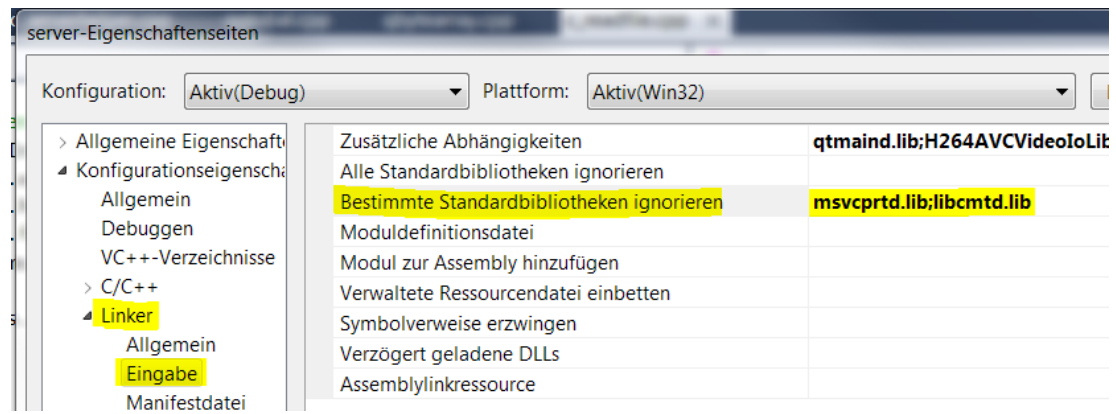


Abbildung 5: Linker-Einstellungen im Visual Studio.

3. VERWENDUNG DER PROGRAMME

In diesem Abschnitt sollen nun die einzelnen Programme vorgestellt werden, die im Rahmen dieses Projektes entwickelt wurden. Dabei geht es vor allem die Bedienung und den optischen Aufbau; die interne Funktionsweise wird in Kapitel 5, Seite 16 ff. erläutert.

3.1 Decoder-Testanwendung

Die Decoder-Testanwendung war das erste Programm, welches im Rahmen dieses Projektes mit der JSVM-Software entwickelt wurde. Es ermöglicht das Dekodierung & Abspielen eines bestimmten Layers einer H.264-SVC-kodierten Datei.

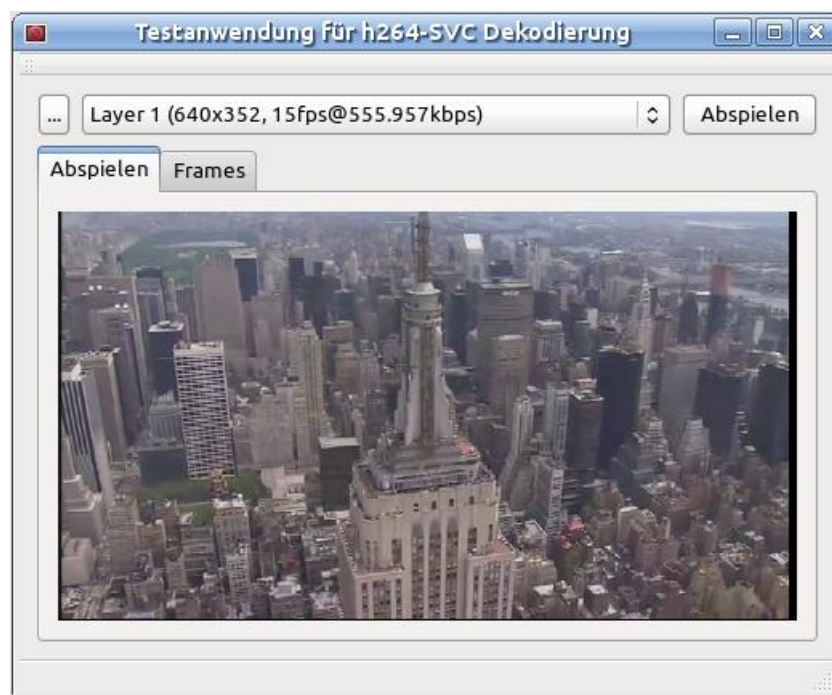


Abbildung 6: Die Decoder-Testanwendung, ausgeführt unter Ubuntu 10.10

Über den Button mit der Aufschrift „...“ kann die Quelldatei ausgewählt werden, in der daneben befindlichen Liste werden die einzelnen Layer aufgelistet. Mit einem Klick auf „Abspielen“ beginnt das Programm mit der Dekodierung und spielt das Video anschließend ab. Zusätzlich ist es möglich, zu Debuggingzwecken über den Unterpunkt „Frames“ eine Liste mit den im Video enthaltenen Frames inklusive der jeweiligen Byteposition abzurufen. Diese Funktion wurde im Rahmen der Implementation des Qualitätswechsels während des laufenden Videos eingeführt und wird später noch deutlicher erklärt.

3. Verwendung der Programme

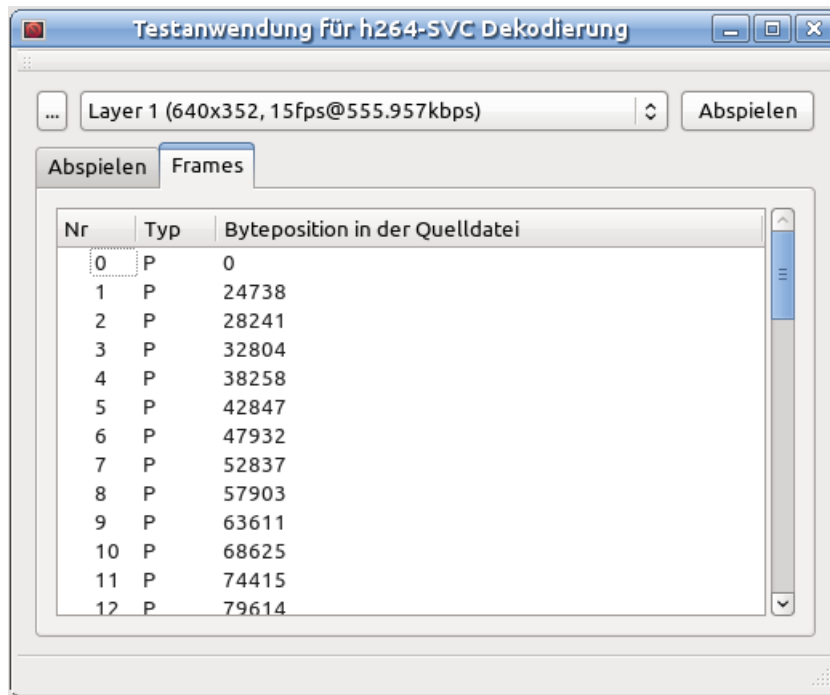


Abbildung 7: Liste der im Video enthaltenen Frames in der Decoder-Testanwendung

3.2 Server-Anwendung

Die Server-Anwendung erfüllt die Aufgabe, die H.264-Inhalte paketweise an die einzelnen Clients zu übertragen, ein Screenshot ist in Abbildung 8 zu sehen. Wie bereits weiter oben erwähnt, wurde die Server-Anwendung primär unter Windows entwickelt. Nichtsdestotrotz ist das Programm problemlos unter Linux lauffähig.

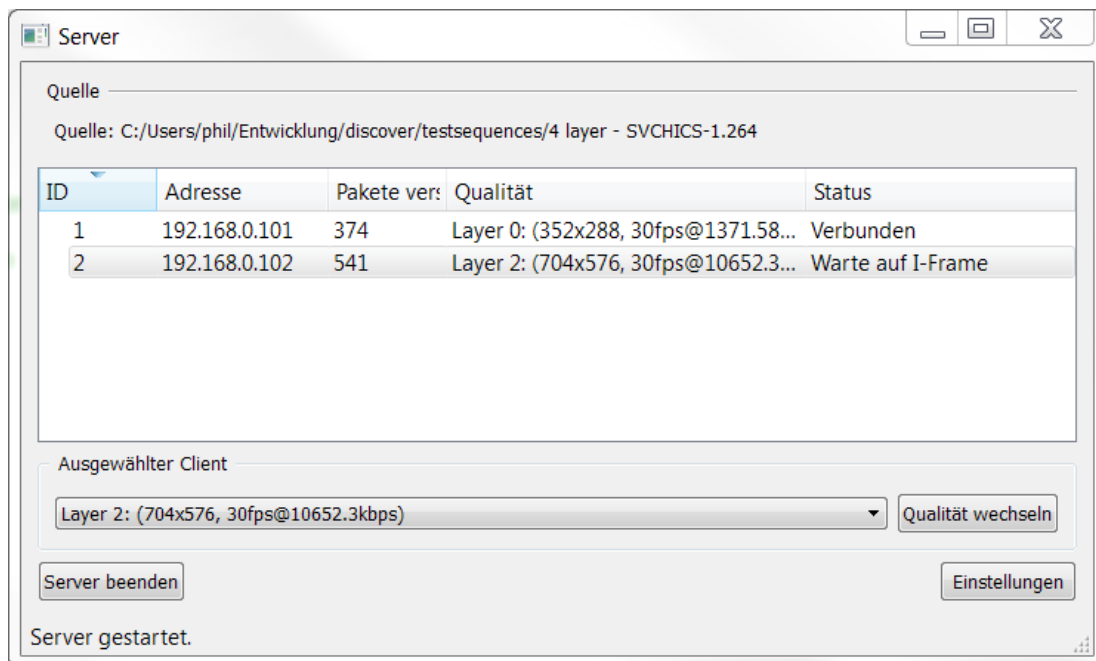


Abbildung 8: Ausführen der Server-Anwendung unter Windows.

In der allerersten Zeile steht der Pfad zur aktuellen Quelldatei, deren Inhalt an die Clients versendet wird. Direkt darunter befindet sich die Liste der Clients, die momentan mit dem Server verbunden sind, wobei folgende Attribute angegeben werden:

- *ID* – Ein numerischer Wert, über den sich der jeweilige Client gegenüber dem Server identifiziert.
- *Adresse* – Die IP-Adresse des Clients.
- *Pakete versendet* – Die Anzahl von Paketen des H.264-Bitstreams, die bereits an den Client versendet wurden.
- *Qualität* – Die Qualitätsstufe, die für diesen Client momentan aktiviert ist.
- *Status* – Eine Angabe über den Zustand der Übertragung.

In dem Feld „Ausgewählter Client“ kann über eine Drop-Down-Liste in Verbindung mit dem Button „Qualität zuweisen“ dem Client, der gerade in der Liste ausgewählt wurde, eine neue Qualität zugewiesen werden.

Mit Hilfe des Buttons in der unteren linken Ecke (auf dem Screenshot beschriftet mit „Server beenden“) kann der Server bei Bedarf beendet und neu gestartet werden. Dies ist beispielsweise notwendig, wenn eine andere Quelldatei gewählt wurde.

Der Button im unteren rechten Bereich des Fensters zeigt das Einstellungsfenster an.

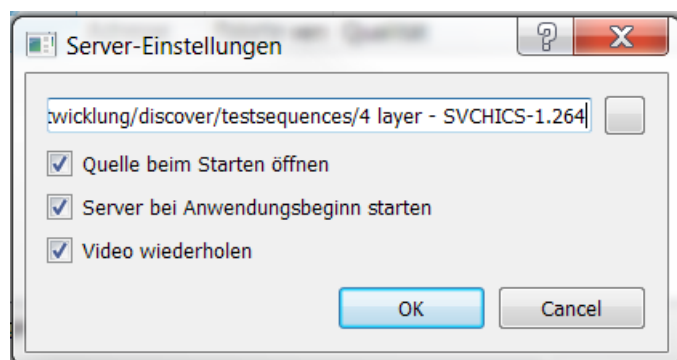


Abbildung 9: Einstellungsdialogfenster der Serveranwendung unter Windows

Neben der Möglichkeit, die Quelldatei auszuwählen, kann hier außerdem eingestellt werden, ob diese bei Programmstart geöffnet und/oder der Server gestartet werden soll. Mit der letzten Option kann angegeben werden, ob das Video wiederholt werden soll. Diese Möglichkeit wurde aus folgendem Grund in die Anwendung integriert: Während des Debuggens ist es von Vorteil, einen im Idealfall unendlich langen Videostream versenden zu können, da die Clientanwendung so ausgiebig getestet werden kann, ohne ständig den Server neustarten zu müssen. Da die allermeisten Testsequenzen allerdings nur wenige Sekunden lang sind und das Enkodierung einer Eigenen unverhältnismäßig viel Aufwand bedeutet hätte, wurde der Server um diese Funktion erweitert, welche es gestattet, ein beliebiges Video immer und immer wieder zu senden.

3.3 Client-Anwendung

Nun soll die Client-Anwendung erläutert werden. Wie bereits erwähnt, ist diese aufgrund der Verwendung von Pipes nur unter Linux lauffähig. Abbildung 10 zeigt einen Screenshot.



Abbildung 10: Ausführung der Client-Anwendung unter Ubuntu 10.10

In der obersten Zeile wird die IP-Adresse des Servers angegeben, mit dem sich der Client verbinden soll. Rechts daneben befindet sich der entsprechende Button, über den der Vorgang eingeleitet wird. War der Verbindungsversuch erfolgreich, wird dies – wie in Abbildung 10 zu sehen ist – dargestellt.

Direkt darunter sind einige hilfreiche Statistiken zum Fortschritt der Dekodierung zu finden. Zum einen lässt sich ablesen, wie viele Pakete vom Server empfangen wurden und wie viele bereits dekodiert wurden³. Direkt darunter ist angegeben, wie viele Frames bereits dekodiert werden konnten und wie viele Frames pro Sekunde die Software schafft. Gerade der letzte Wert ist von erhöhter Bedeutung, da er angibt, ob das empfangene Video in Echtzeit abgespielt werden kann: Im oberen Beispiel empfängt der Client ein Video mit einer Framerate von 30 fps, schafft allerdings nur 17 Frames pro Sekunde. Würde man nun mit dem Abspielen beginnen, würde die Wiedergabe früher oder später abbrechen. Wäre allerdings eine Framerate von z.B. 15 fps für das Video vorgesehen, gäbe es keine Probleme⁴.

³ Sofern das System, auf dem das Programm ausgeführt wird, auf einem halbwegs aktuellen Stand ist, sollten diese Zahlen immer sehr nah beieinander sein

⁴ Während meiner Tests des Servers mit mehreren Clients installierte ich Ubuntu 10.10 sowie Xubuntu 10.10 in jeweils zwei virtuellen Maschinen mit identischer Konfiguration. Interessanterweise war unter Xubuntu die Dekodierungsgeschwindigkeit durchschnittlich 5 Frames/Sekunde höher als unter Ubuntu. Leider konnte ich aus Zeitgründen nicht herausfinden, woran genau dies lag, aber da beide Distributionen größtenteils die gleichen Pakete verwenden, gibt es wenige denkbare Ursachen.

Die Information über die Framerate sowie einige andere Attribute lassen sich direkt unter dem Punkt „Videoparameter“ ablesen. Zusätzlich ist die Nummer des Layers angegeben, welches die Anwendung momentan vom Server erhält.

Der Button „Nächstes Paket anfordern“ wurde ursprünglich zu Debuggingzwecken integriert und tut genau das, was seiner Beschriftung entspricht, nämlich das nächste Paket vom Server anfordern. Mittlerweile wurde das entsprechende Problem behoben, nichtsdestotrotz kann seine Funktionsweise noch dazu dienen, sich gewisse Teile der Programmierung zu veranschaulichen.

Über den Button „Abspielen“ kann ein Fenster geöffnet werden, welches mit Hilfe der Phonon-Engine das dekodierte Material wiedergibt. Sofern die Dekodierungsgeschwindigkeit hoch genug ist (s.o.), kann dies in Echtzeit erfolgen.

Zusätzlich gibt es noch die Möglichkeit, das Einstellungsfenster aufzurufen.



Abbildung 11: Einstellungsfenster der Client-Anwendung unter Ubuntu 10.10

Hier kann der Benutzer das Verzeichnis, in dem temporäre Dateien angelegt werden, ändern.

4. ENTWICKELN AUF BASIS VON JSVM

4.1 Grundlegendes

Bevor beschrieben werden kann, wie die einzelnen Programme intern funktionieren, ist es zunächst notwendig, sich Klarheit über einen Teil der JSVM-Referenzsoftware zu schaffen. Daher sollen in diesem Abschnitt die notwendigen Grundlagen für eine Entwicklung auf Basis von JSVM dokumentiert werden⁵.

Ausgangspunkt sei eine fertig kompilierte Version von JSVM sowie ein neu angelegtes Projekt in einer bevorzugten Entwicklungsumgebung⁶.

Im Unterordner „include“ sind die jeweiligen Header-Dateien für die einzelnen Klassen zu finden. Diese müssen in der richtigen Reihenfolge eingebunden werden, damit es zu keinen Problemen kommt. Für die Verwendung der Decoder-Klassen lautet diese wie folgt:

```
#include <H264AVCDecoderLib.h>
#include <CreatorH264AVCDecoder.h>
#include <H264AVCVideoIoLib.h>
#include <ReadBitstreamFile.h>
#include <WriteBitstreamToFile.h>
```

Für andere Szenarien kann die Reihenfolge dem Quelltext der bei JSVM mitgelieferten Testanwendungen entnommen werden, im Zweifelsfall könnte auch ausprobieren notwendig sein.

An dieser Stelle sei noch zu bemerken, dass sich ein Teil der Objekte und Strukturen der JSVM-Bibliotheken innerhalb eines Namespaces mit dem Titel „h264“ befinden. Da dies allerdings mehr verwirrt als hilft, ist anzuraten, in jedem Projekt die Direktive `using namespace h264` zu verwenden; im weiteren Verlauf des Textes wird entsprechend davon ausgegangen.

4.2 Datentypen

Abgesehen davon, dass innerhalb der JSVM-Bibliotheken sämtliche Datentypen neudefiniert sind (zu finden in *TypeDefs.h*) gibt es einige wesentliche Strukturen und Klassen, die hier noch aufgelistet werden sollen:

- *BinData* – Speichert binäre Daten (unsigned char*), wird in der Regel für Pakete verwendet.
- *AccessUnit* – Eine Sammlung von NALUnits, die dekodiert werden kann.

⁵ Da die JSVM-Bibliotheken so gut wie gar nicht dokumentiert sind, basiert der Großteil der in diesem Abschnitt verfügbaren Informationen auf ausprobieren.

⁶ Notwendige Einstellungen wurden in den Abschnitten „2.3 Kompilierung unter Linux“ sowie „2.4 Kompilierung unter Windows“ besprochen.

4.3 Wichtige Klassen

Für die Dekodierung von Videomaterial sind vor allem folgende Klassen relevant:

- *CreatorH264AVCDecoder* – dekodiert Pakete und Units in YUV-Rohmaterial.
- *H264PacketAnalyzer* – analysiert Pakete und stellt so Informationen bereit.
- *ReadBitstreamFile* – liest aus einer .264-Quelldatei.
- *WriteYuvToFile* – schreibt YUV-Rohdaten in eine Datei.

Ein Objekt dieser Klassen lässt sich nicht über den üblichen *new*-Operator anlegen, da der *Constructor* nicht *public* ist. Stattdessen muss auf folgenden Umweg zurückgegriffen werden:

```
H264AVCPacketAnalyzer *analyzer;
H264AVCPacketAnalyzer::create( analyzer );
analyzer->init();
```

4.3.1 – Die Klasse „CreatorH264AVCDecoder“

Diese Klasse ist – wie bereits erwähnt – für die Hauptarbeit bei der Dekodierung zuständig. Mit Hilfe der Methode `CreatorH264AVCDecoder::initNALUnit(BinData*, AccessUnit)` können innerhalb einer `AccessUnit` (eines Bildes) solange `NALUnits` (Slices) initialisiert werden, bis ein Bild vollständig ist. Ist dies der Fall, kann das entsprechende Bild dekodiert werden. Da das Objekt, welches man der Methode `initNALUnit` übergibt, immer das gleiche sein muss (damit B- und P-Frames richtig dekodiert werden können), ist es leider nicht möglich, parallel in zwei Threads `AccessUnits` zu sammeln und zu verarbeiten.

Für die Dekodierung ist anschließend die Methode `CreatorH264AVCDecoder::processNALUnit(PicBuffer*, PicBufferList&, PicBufferList&, BinDataList&, &NALUnit)` zuständig. Wie zu sehen ist, werden als Argumente eine Reihe von Listen erwartet, die unter anderem dazu dienen, Bildelemente zur Verwendung in kommenden Frames zwischenzuspeichern (oder entsprechend zuzugreifen). Die genaue Funktionsweise würde den Rahmen dieses Abschnitts sprengen, lediglich der letzte Parameter soll erklärt werden:

Ist eine `AccessUnit` – wie oben erläutert – vollständig, kann mit Hilfe der Methode `AccessUnit::getAndRemoveNextNalUnit(NALUnit*)` ein Zeiger auf die nächste zu verarbeitende `NALUnit` (das nächste zu verarbeitende Slice) erhalten werden. Dieser wird entsprechend an die Methode `processNALUnit` übergeben.

In Abbildung 12 ist der Ablauf in vereinfachter Form dargestellt.

4. Entwickeln auf Basis von JSVM

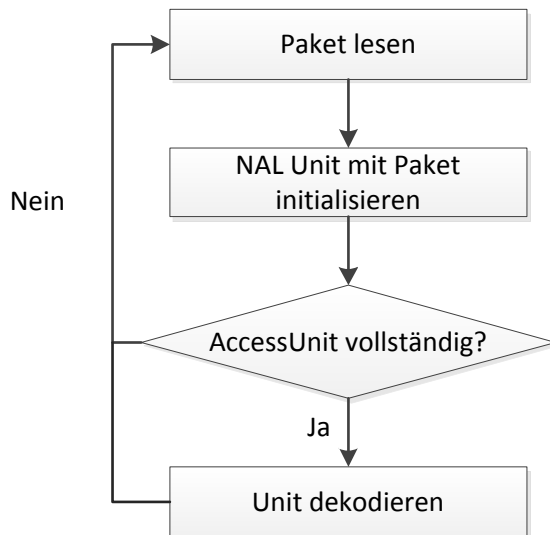


Abbildung 12: Dekodierung mit der Klasse „CreatorH264AVCDecoder“ (vereinfacht)

4.3.2 – Die Klasse „H264PacketAnalyzer“

Wie der Name schon vermuten lässt, verfügt diese Klasse über die notwendigen Methoden zur Analyse von Paketen. Durch die Analyse eines Paketes sind verschiedene Informationen zugänglich, z.B. SEI⁷, welche Angaben über die Bitrate, die Auflösung und ähnlichem enthält oder auch die Information, zu welchem Layer das jeweilige Paket gehört.

Soll eine bestimmte Qualitätsstufe dekodiert werden, ist es notwendig, dass die Klasse `CreatorH264AVCDecoder` nur Pakete verarbeitet, die dieser Stufe oder einer geringeren angehören. Mit Hilfe des Aufrufs `H264PacketAnalyzer::process(BinData*, PacketDescription&, SEI::SEIMessage*&)` wird das Paket zunächst analysiert. Anschließend können dem `PacketDescription`-Objekt die entsprechenden Informationen entnommen werden.

Berücksichtigen wir dies in unserem Decoding-Prozess, kann das Diagramm aus dem vorherigen Abschnitt wie folgt erweitert werden:

⁷ SEI: *Supplemental enhancement information*

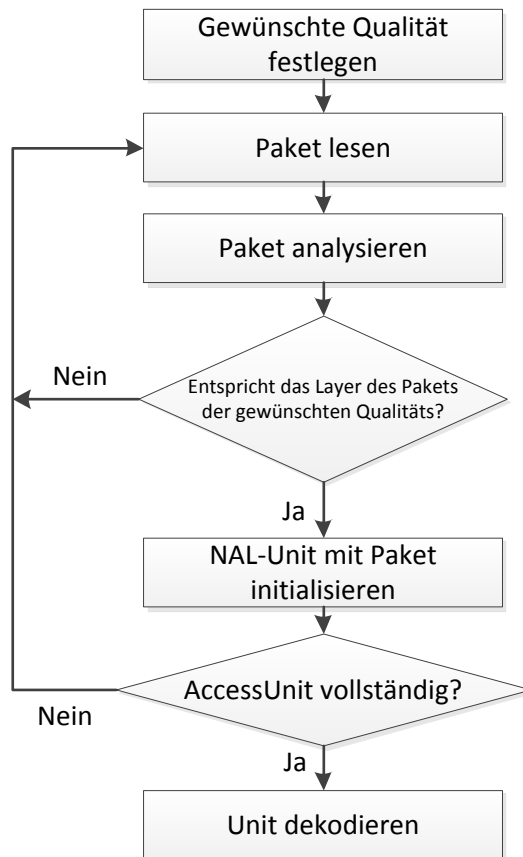


Abbildung 13: Dekodierungsvorgang unter Berücksichtigung der Qualität

4.3.3 – Die Klasse „ReadbitStreamFile“

Auch wenn es möglicherweise auf den ersten Blick trivial erscheint, bin ich mir sicher, dass es hilfreich ist, zu wissen, wie die Klasse zum Lesen von Daten aus einem Bitstream zumindest teilweise intern arbeitet. Die Besprechung sämtlicher Aspekte würde zu lange dauern, aber zumindest die wichtigsten Funktionen sollen erklärt werden, zumal der Quelltext fast nicht dokumentiert ist.

Die Verwendung gestaltet sich zunächst denkbar einfach: Mit der Methode `extractPacket (BinData*, Bool&)` kann das nächste Paket in der Quelldatei gelesen werden. Wird dieses nicht mehr benötigt, sollte es über `ReleasePacket (BinData*)` freigegeben werden.

Intern geschieht folgendes: Es werden Sequenzen mit einer Größe von 400 Byte aus der Datei gelesen. Beinhaltet diese Sequenz einen StartCode (s.A. Abschnitt 1.1), wird die Sequenz an dieser Stelle aufgeteilt: Der erste Teil wird zurückgegeben, der zweite zwischengespeichert. Beim nächsten Lesevorgang wird dieser Zwischenspeicher entsprechend mitverarbeitet.

Dies ist wichtig, weil die Klasse somit nicht bis zum nächsten Paket sondern evtl. darüber hinaus liest. Gibt es allerdings an dieser Stelle keine Daten, kommt es möglicherweise zu einem Problem.

5. DER INTERNE AUFBAU DER PROGRAMME

5.1 Die Server-Anwendung

Da nun sämtliche Grundlagen bezüglich der JSVM-Bibliotheken bekannt sind, soll in diesem Abschnitt die interne Funktionsweise der einzelnen Programme besprochen werden.

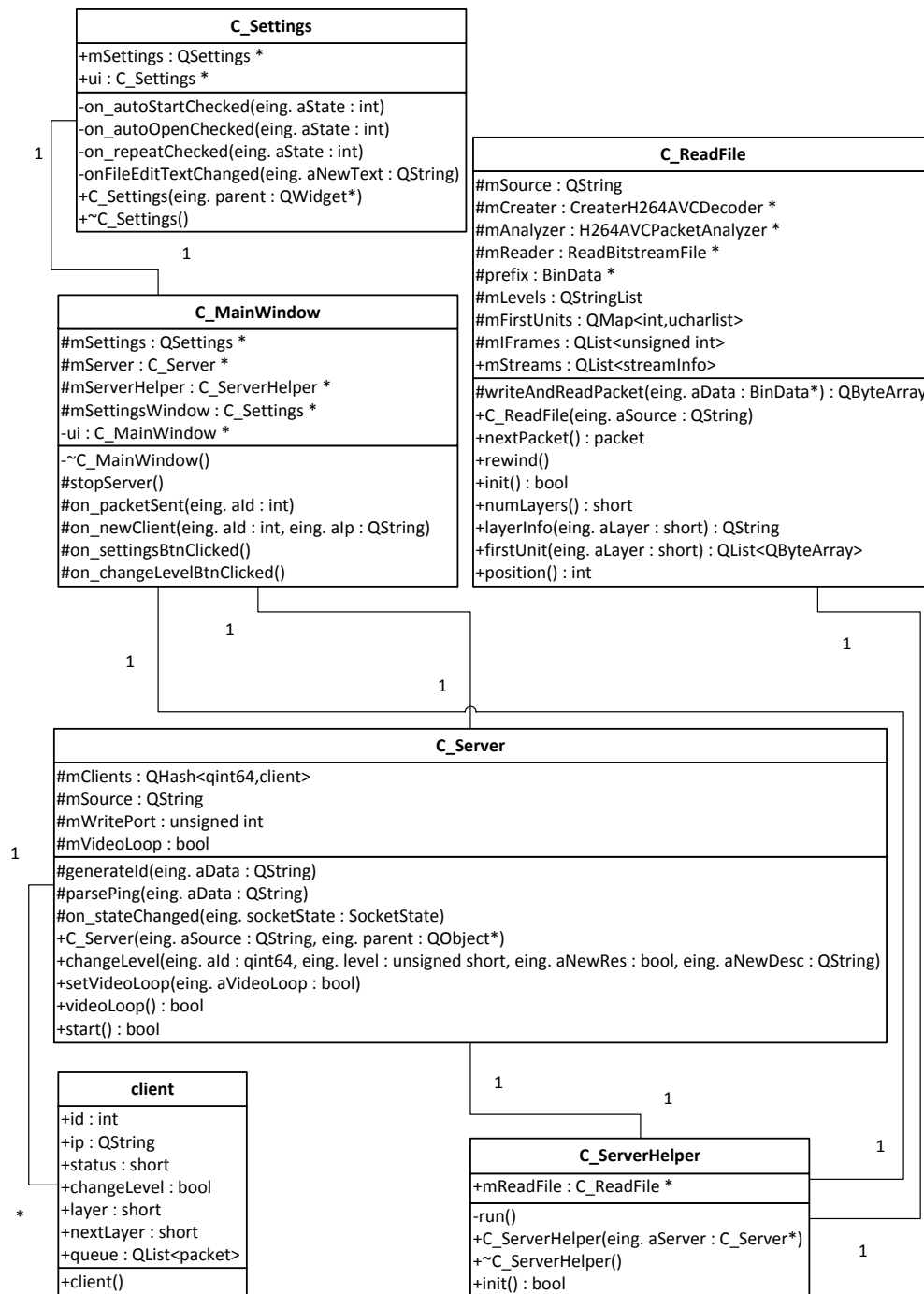


Abbildung 14: Klassendiagramm der Server-Anwendung

Abbildung 14 zeigt ein Klassendiagramm der Server-Anwendung. Die Klasse *C_Settings* ist für das Speichern der Einstellungen zuständig, die Klasse *C_MainWindow* für die Darstellung des Hauptfensters.

Die Hauptarbeit des Programms werden von den Klassen *C_Server* sowie *C_ServerHelper* übernommen; letztere erledigt einige Aufgaben, wie das Versenden von Paketen an die Clients, in einem separaten Thread, was die Antwortzeit des Servers deutlich verringert.

Die Klasse *Client* darf nicht mit der ähnlich benannten Klasse der Client-Anwendung verwechselt werden; vielmehr handelt es sich hier um eine Datenstruktur, mit deren Hilfe der Server die Informationen der einzelnen verbundenen Clients speichert, wie z.B. IP-Adresse, ID oder die gewünschte Qualität.

C_ReadFile wird primär von der Klasse *C_ServerHelper* verwendet und dient zum Lesen von Paketen aus einer Quelldatei. Die Methode „nextPacket ()“ gibt das nächste zu versendende Paket zurück, eingebettet in einem Element der Struktur „packet“, welche wie folgt aufgebaut ist:

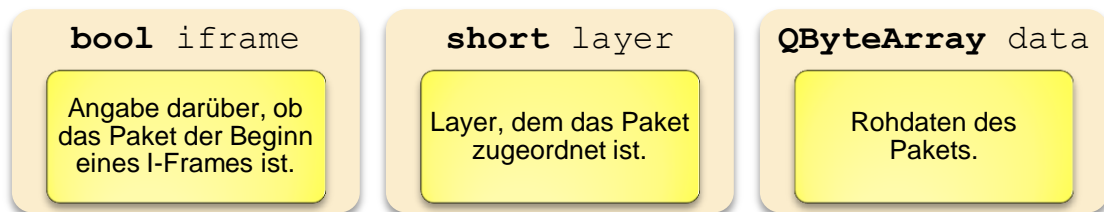


Abbildung 15: Einzelne Elemente der Struktur „packet“

5.2 Die Client-Anwendung

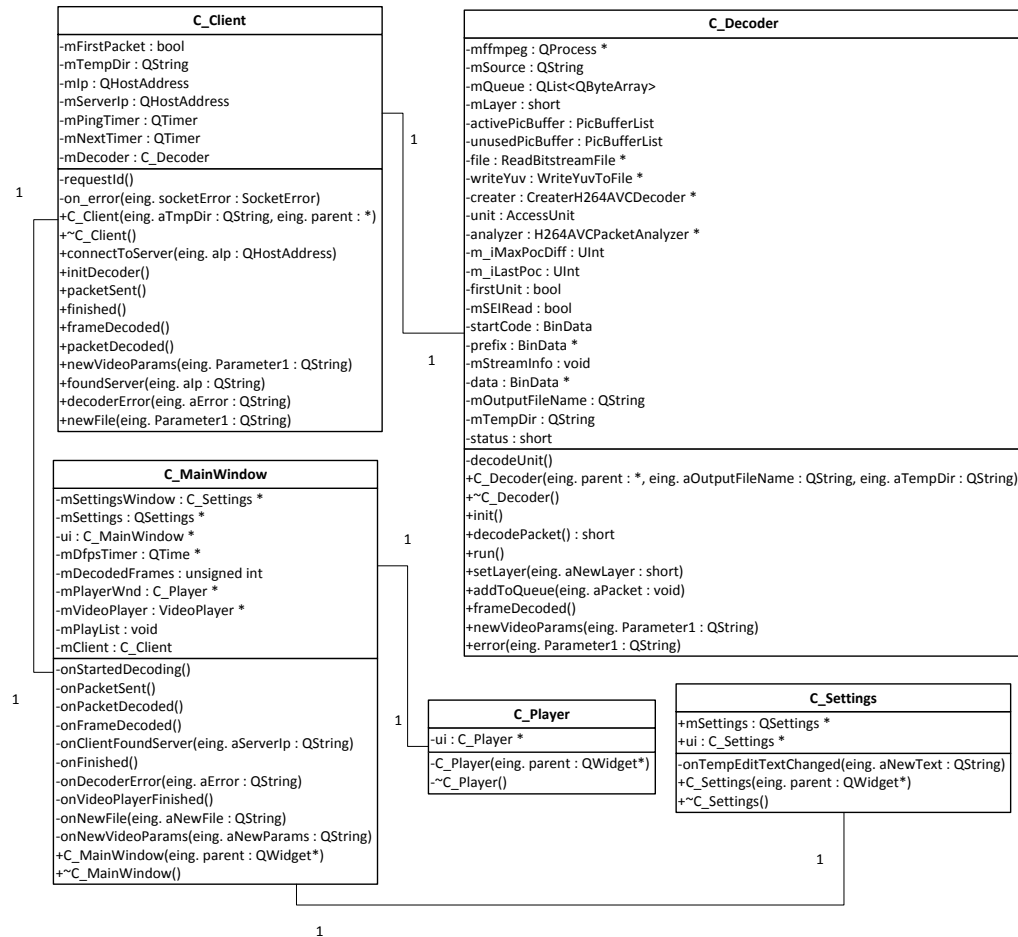


Abbildung 16: Klassendiagramm der Client-Anwendung

In Abbildung 16 ist ein Klassendiagramm der Client-Anwendung dargestellt. Die Primärklasse, *C_Client*, wird von der Hauptfenster-Klasse *C_MainWindow* erzeugt und verwaltet den Netzwerkverkehr mit dem Server. Eintreffende Pakete werden über die Funktion *C_Decoder::addToQueue* der Warteschlange des Decoders hinzugefügt, welcher von der Klasse *C_Decoder* repräsentiert wird.

Die Klasse *C_Settings* ist – ähnlich wie bei der Server-Anwendung – für die Darstellung des Einstellungsfensters zuständig.

6. FUNKTIONSWEISE DER PROGRAMME

6.1 Kommunikation zwischen Server und Client

6.1.1 Verbindungsaufbau

Die Kommunikation zwischen beiden Programmen ist relativ simpel gehalten, soll aber nichtsdestotrotz hier dokumentiert werden. Es beginnt zunächst damit, dass der Benutzer auf Clientseite die IP-Adresse des Servers in das entsprechende Eingabefeld eintippt⁸ und den Verbinden-Button betätigt. Die Clientanwendung schickt daraufhin einen PING an die angegebene IP an Port 5555 in folgendem Format:

PING 127.0.0.1,192.168.33.67,135.74.5.11

IP-Adressen der Netzwerkadapter
des Clients

Wie zu sehen ist, besteht das Kommando aus dem Schlüsselwort „PING“ sowie einer Liste der IP-Adressen, die den Netzwerkadaptern des Systems zugeordnet sind, auf dem die Clientanwendung ausgeführt wird. Der Server antwortet an jede einzelne dieser Adressen mit einem PONG an Port 5554 nach folgendem Format:

PONG aaa.bbb.ccc.ddd

Mögliche IP-Adresse
des Clients

Die Adresse „aaa.bbb.ccc.ddd“ ist dabei eine der möglichen drei Client-Adressen – in diesem Beispiel würde der Server also insgesamt drei PONGs versenden.

Da der Client unter mindestens einer dieser Adressen erreichbar sein muss (ansonsten wäre der PING nicht angekommen), wird er entsprechend mindestens (und in der Regel nicht mehr als) einen PONG erhalten. Der Client ist nun darüber informiert, über welche IP-Adresse er erreichbar ist.

Als nächstes benötigt er eine eindeutige ID vom Server, um mit diesem zu kommunizieren. Zu diesem Zweck schickt er eine Anfrage nach folgendem Format:

ID_REQUEST aaa.bbb.ccc.ddd

IP-Adresse des Clients

Da dem Server zu diesem Zeitpunkt die IP, unter der der Client eindeutig erreichbar ist, noch nicht bekannt ist, ist diese in der Anfrage enthalten. Der Server kann den Client nun eindeutig zuordnen, eine ID generieren und diese wie folgt mitteilen:

⁸ S.a. Abbildung 10, S. 10

NEW_ID 1

6.1.2 Client-Behandlung

Nun könnte mit der Übertragung des Videomaterials begonnen werden, bevor dies allerdings besprochen wird, zunächst einige erklärende Worte zur internen Handhabung auf Serverseite.

Wie bereits erwähnt, werden die Informationen über die einzelnen mit dem Server verbundenen Clients in Objekten der Datenstruktur „client“ abgelegt, welche u.a. eine Variable zur Beschreibung des Status des Clients enthält.

Der Inhalt dieser Variable ist maßgeblich für das Verhalten der *C_ServerHelper*-Klasse, welche nun beschrieben werden soll.

Wird der Server gestartet, legt das Programm daher nicht nur ein Objekt der Klasse *C_Server* an, sondern auch eines der Klasse *C_ServerHelper*. Diese Klasse arbeitet – wie erwähnt – als separater Thread und führt einen Großteil der Kommunikation durch; ein stark vereinfachter Ablauf ist in Abbildung 17 zusehen.

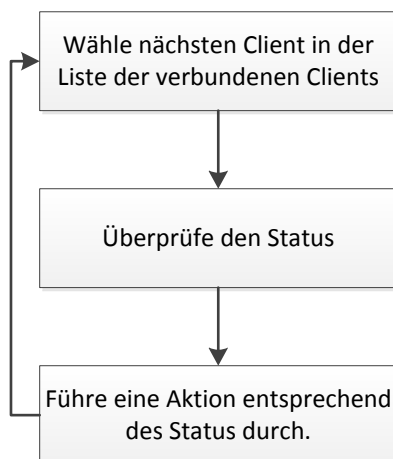


Abbildung 17: Stark vereinfachter Ablauf der *C_ServerHelper*-Klasse

Hat sich der Client mit dem Server verbunden und eine ID erhalten, wird ihm zunächst die Statusnummer 3 zugeteilt. Dies besagt, dass auf den Beginn der Übertragung gewartet wird. Auf Serverseite wird nun 0,5 Sekunden gewartet, um dem Client ein wenig Zeit zu verschaffen, sich auf den Empfang der Daten vorzubereiten; anschließend wird die Übertragung gestartet, wobei die Statusnummer auf 1 gesetzt wird.

6.1.3 Übertragung der Daten

Wurde ein Paket versendet, wechselt der Status auf den Wert 2. Dies bedeutet, dass der Server keine Pakete mehr versendet, bis der Client das nächste Paket mit Folgendem Schlüsselwort anfordert:

NEXT_PACKET 1

ID des
Clients

Daraufhin wird der Status auf 1 gesetzt, das nächste Paket versendet, der Status wird wieder auf 2 gesetzt usw. Abbildung 18 verdeutlicht das Prinzip noch einmal.

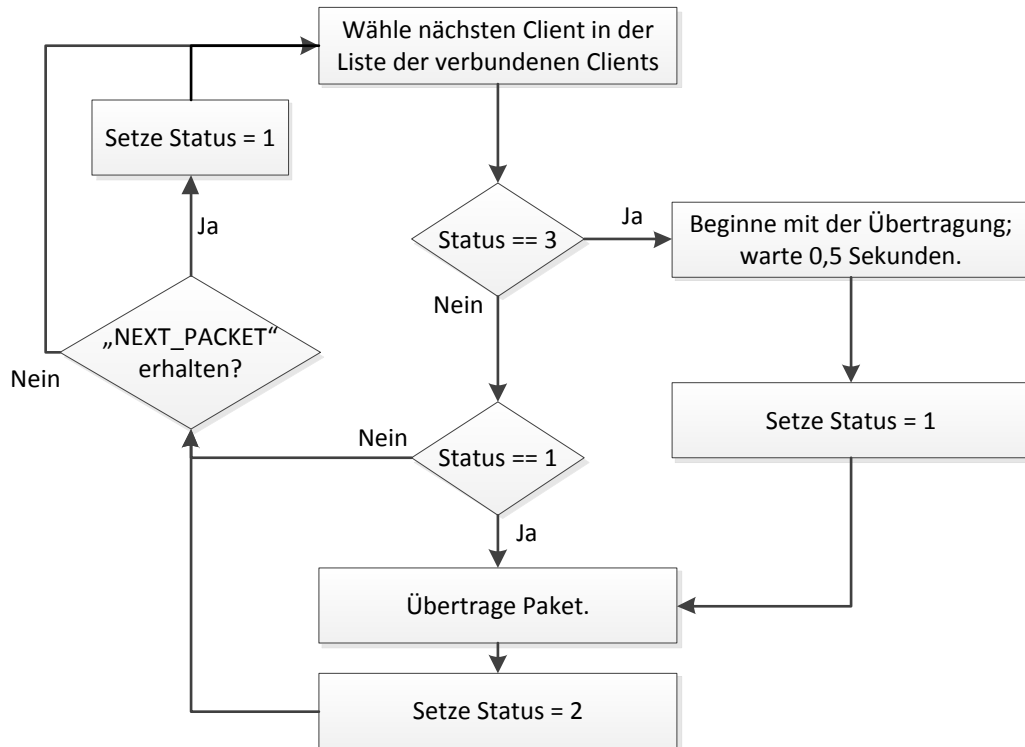


Abbildung 18: Statusveränderungen der Clients auf Serverseite

6.1.4 Wechsel der Qualität während der Übertragung

Kommen wir nun zum eigentlichen Hauptziel des Projektes: Das Wechseln der Qualität während der Übertragung.

Wie bereits erläutert, sind die Pakete eines H.264/MPEG-4 SVC-Bitstreams einzelnen Layern zugeordnet. Jeder Client wünscht eine andere Qualität, d.h. ein anderes Layer. Der Server muss dies bei der Übertragung entsprechend berücksichtigen; Abbildung 19 schlüsselt den Vorgang „Übertrage Paket“ aus Abbildung 18 entsprechend auf⁹.

⁹ Gilt nur, wenn nicht mehr als ein Client mit dem Server verbunden ist.

6. Funktionsweise der Programme

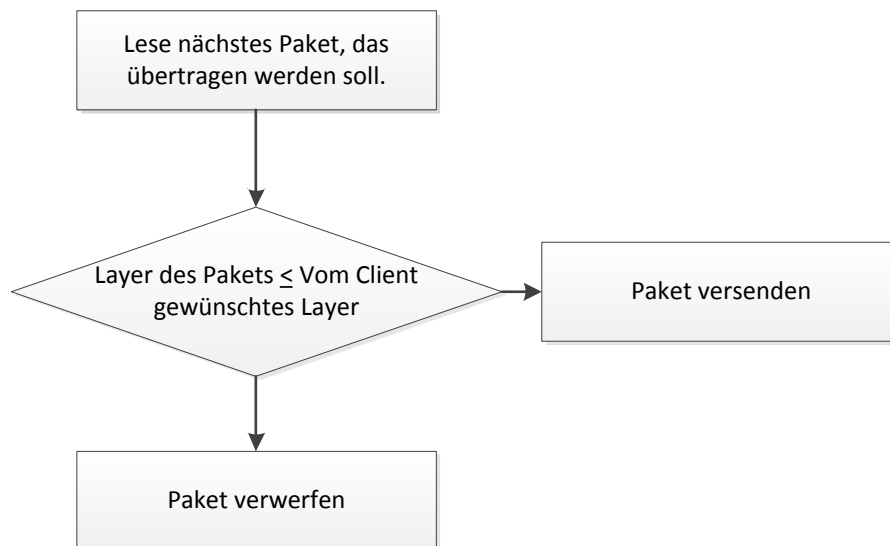


Abbildung 19: Sortierung der Pakete als Flussdiagramm

Soll nun die Qualität, die ein Client erhält, geändert werden, reicht es nicht aus, lediglich die zusätzlichen Pakete zu senden. Zunächst muss mit dieser Änderung auf den nächsten I-Frame gewartet werden, da ansonsten massive Probleme bei der Dekodierung auftreten, was dazu führt, dass die JSVM-Bibliothek das Programm umgehend beendet.

Sollte sich beim Wechsel der Qualität nicht nur die Bitrate, sondern auch die Auflösung ändern¹⁰, ist es notwendig, den Dekodierungsprozess neu zu initialisieren. Dies liegt daran, dass aus den H.264-Daten YUV-Rohdaten erzeugt werden, welche mit den entsprechenden Faktoren richtig interpretiert werden müssen. Eine Neuinitialisierung schließt eine Übertragung der ersten Unit der entsprechenden Qualitätsstufe mit ein, da nur diese erste Unit die notwendige *Supplemental enhancement information* beinhaltet. Um dies jederzeit möglich zu machen, speichert der Server sämtliche *firstUnits* aller Qualitätsstufen beim Öffnen der Quelldatei zwischen. Abbildung 20 stellt den Vorgang als Flussdiagramm dar.

¹⁰ Ein Wechsel der Framerate ist im Übrigen völlig unerheblich, da dies nur ein Parameter im jeweiligen Videocontainer darstellt und keinen weiteren Einfluss auf die Interpretation der Rohdaten seitens des Decoders hat.

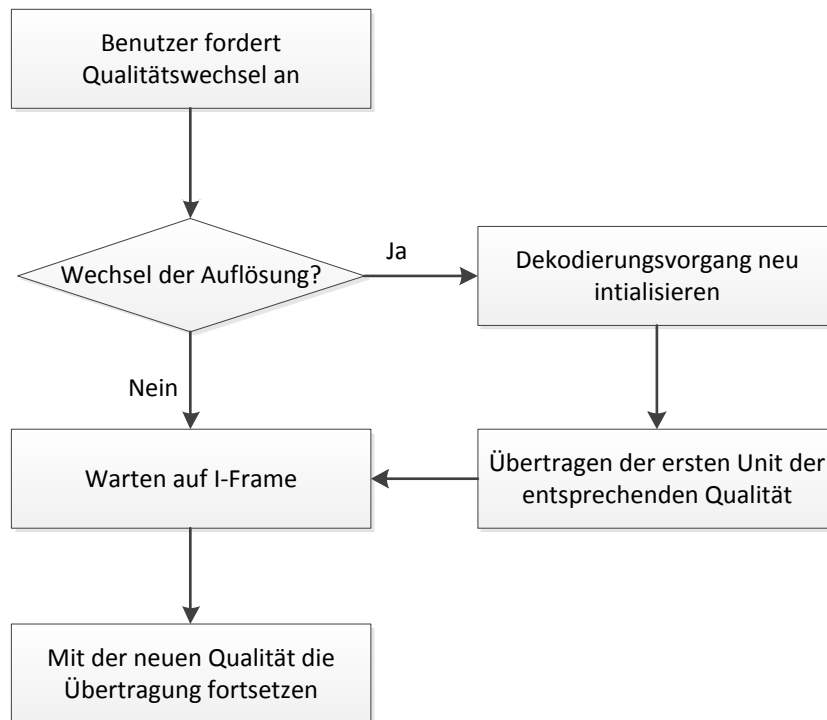


Abbildung 20: Vereinfachter Qualitätswechsel als Flussdiagramm

7. INTEGRATION DES QUELLCODES IN NEUE PROJEKTE

Viele der in diesem Projekt entwickelten Klassen wurden so angelegt, dass sie weiterverwendet werden können. In diesem Abschnitt sollen die Möglichkeiten besprochen werden.

7.1 Verwendung der Klasse C_Decoder

Die Klasse C_Decoder der Client-Anwendung kann wiederverwendet werden, um Daten eines H.264-Bitstreams zu dekodieren und in eine AVI-Datei zu schreiben. Dafür muss zunächst mit Hilfe des Constructors ein Objekt angelegt werden. Folgender Quelltextausschnitt stellt die Syntax dar:

```
C_Decoder(QObject *parent, QString aOutputFileName = "",
           QString aTempDir = "/tmp");
```

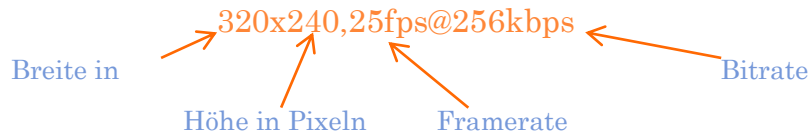
Der Inhalt des ersten Parameters wird an den übergeordneten Constructor der Klasse QThread, von der die Klasse C_Decoder erbt, weitergegeben. Der zweite Parameter gibt den Namen der Ausgabedatei an; ist dieser leer, wird „out.avi“ genommen. Der letzte Parameter ermöglicht die Änderung des Verzeichnisses für temporäre Dateien. Vor allem bei System, auf denen der Pfad „/tmp“ nicht existiert, wie beispielsweise Windows, sollte hier eine Angabe gemacht werden.

Nachdem das Objekt erzeugt wurde, ist noch die Durchführung einer Initialisierung notwendig. Durch die Methode `init()` kann dies veranlasst werden, hierbei wird auch der Thread gestartet.

Über die Methode `setLayer(int)` kann die Nummer des gewünschten Layers angegeben werden. Sollte dies im laufenden Decodierungsvorgang geschehen, müssen auf die entsprechenden Sachverhalte geachtet werden, die in Abschnitt „6.1.4 Wechsel der Qualität während der Übertragung“ auf Seite 21 besprochen wurden, da die Klasse selbst nicht darauf achtet, ob der Wechsel beispielsweise zu Beginn eines I-Frames erfolgt.

Schließlich werden mit Hilfe der Funktion `addToQueue(QByteArray)` die zu dekodierenden Pakete an das Objekt übergeben. Wie es der Name vermuten lässt, werden diese zunächst in eine Warteschlange eingereiht und nach und nach dekodiert (abhängig von der Leistung der CPU). Über die Signale `packetDecoded()` sowie `frameDecoded()` kann der Fortschritt entsprechend verfolgt werden.

Ein weiteres wichtiges Signal lautet `newVideoParams(QString)`. Dieses wird aktiviert, sobald die Attribute des Videos bekannt sind. Der Inhalt des Parameters hat folgenden Aufbau:



Sollten bei der Dekodierung Probleme auftreten, können diese mit Hilfe des Signals `error(QString)` empfangen und entsprechend darauf reagiert werden.

Wurden alle verfügbaren Pakete an den Decoder übergeben, kann das Objekt gelöscht werden. Dabei werden zunächst alle noch zwischengespeicherten Pakete dekodiert, bevor der Speicher freigegeben wird.

Zum besseren Verständnis ist ein beispielhafter Ablauf einer Verwendung der C-Decoder-Klasse in Abbildung 21 noch einmal als Flussdiagramm dargestellt.

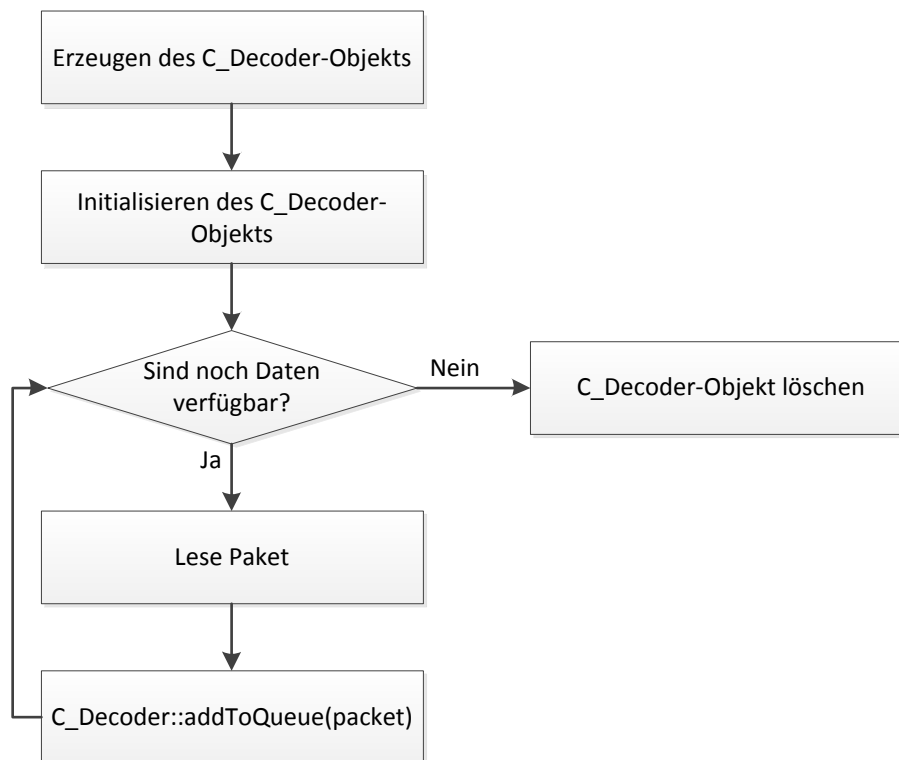


Abbildung 21: Verwendung der C_Decoder-Klasse

7.2 Verwendung der Klasse C_ReadFile

Es ist problemlos möglich, die Klasse `C_ReadFile` der Serveranwendung in neue Projekte zu integrieren, um Pakete aus H.264-Quelldateien zu lesen. Warum dies komfortabler als die Klasse „`ReadBitstreamFile`“ der JSVM-Bibliotheken ist, wird in Abbildung 22 dargestellt.

Um mit Hilfe der Klasse `C_ReadFile` Daten zu lesen, ist neben der Erzeugung des Objekts ebenfalls eine Initialisierung notwendig. Die Methode `C_ReadFile::init()`

7. Integration des Quellcodes in neue Projekte

gibt einen Wert des Typs `bool` zurück; ist dieser `true`, war die Initialisierung erfolgreich. Wird allerdings `false` zurückgegeben, ist ein Problem aufgetreten und es bedarf einer Untersuchung.

Wurde die Datei erfolgreich geöffnet, können nun die einzelnen Pakete mit Hilfe der Methode `C_ReadFile::nextPacket()` nacheinander ausgelesen werden. Der Rückgabewert dieser Funktion ist vom Typ `packet`, welcher bereits in Abbildung 15 auf Seite 17 erläutert wurde.



C_ReadFile	ReadBitstreamFile
<ul style="list-style-type: none">•Fügt den gelesenen Paketen Informationen hinzu, wie etwa die Nummer des Layers.•Speichert die Paketrohdaten als <code>QByteArray</code>.•Ermöglicht schnellen Zugriff auf die ersten Units der verschiedenen Layer.•Ermöglicht einfaches "Zurückspulen" des Videos.	<ul style="list-style-type: none">•Liest einzelne Pakete, speichert die Rohdaten als <code>BinData*</code>.•Liefert Pakete nicht immer in der Reihenfolge, in der sie dekodiert werden müssen.•Liefert keine Informationen über die Layerzugehörigkeit eines Pakets.•Informationen über den Bitstream bzw. die einzelnen Layer sind nicht verfügbar.

Abbildung 22: `C_ReadFile` und `ReadBitstreamFile` im Vergleich

Neben dieser Methode gibt es einige weitere Hilfreiche Funktionen, mit denen Informationen über die Quelldatei abgerufen werden können.

Die Funktion `numLayers()` gibt die Anzahl der verfügbaren Layer zurück. `layerInfo(short)` liefert einen `QString`, der das Layer mit der entsprechenden Nummer im bekannten Format beschreibt. Mit der Methode `position()` lässt sich die aktuelle Leseposition innerhalb der Quelldatei abfragen.

Soll ein Wechsel der Qualität mit Auflösungswechsel durchgeführt werden, ist es bekanntlich notwendig, die erste Unit des entsprechenden Layers erneut zu versenden¹¹. Die Klasse `C_ReadFile` macht diese Prozedur sehr einfach, da über die Methode `firstUnit(short)` auf eine Liste sämtlicher Pakete des entsprechenden Layers zugegriffen werden kann.

Zu guter Letzt ist es noch möglich, den H.264-Bitstream „zurückzuspulen“. Dazu existiert die Methode `rewind()`, welche die Leseposition innerhalb der Datei auf den Beginn des ersten I-Frames setzt.

¹¹ S.a. Abschnitt „6.1.4 Wechsel der Qualität während der Übertragung“, S. 22.

7.3 Verwendung der Serverklassen

Sollte es gewünscht sein, den Server innerhalb eines anderen Projektes erneut zu verwenden, so werden zunächst mindestens die Klassen *C_Server* sowie *C_ServerHelper* benötigt. Wird vorausgesetzt, dass die Quelldaten nicht aus einer Datei eingelesen werden, sind noch einige weitere Schritte notwendig. Dies soll nun erklärt werden.

7.3.1 Starten des Servers

Gehen wir zunächst vom einfachsten Fall aus: Innerhalb eines neuen Projekts soll der Server-Quellcode weiterverwendet werden, wobei der zu sendende H.264-Inhalt in einer Datei gespeichert ist.

Sowohl ein Objekt der Klasse *C_Server* als auch der Klasse *C_ServerHelper* müssen erstellt und gestartet werden. Das folgende Quelltextbeispiel veranschaulicht den Vorgang:

```

10. // Serverobjekt erstellen, dabei gleich die Quelle angeben.
11. C_Server *server = new C_Server("/tmp/stream.264");
12. if( !server->start() )
13. {
14.     // Beim Starten ist ein Fehler aufgetreten.
15.     delete server;
16.     return;
17. }
18.
19. // Hilfsobjekt anlegen.
20. C_ServerHelper *serverHelper = new C_ServerHelper(server);
21.
22. if( !serverHelper->init() )
23. {
24.     // Gibt C_ServerHelper::init() false zurück, konnte
25.     // die Quelle nicht gelesen werden.
26.     delete server;
27.     delete serverHelper;
28.     return;
29. }
30.

```

Wurden diese Schritte erfolgreich ausgeführt, ist der Server bereit und wartet auf Clients. Verbindet sich Einer, wird das Signal `C_Server::newClient(int aId, QString aIp)` aktiviert, worüber das entsprechende übergeordnete Objekt informiert wird. Sind erst einmal IDs von Clients bekannt, kann die Methode `changeLevel(quint64 aId, unsigned short level, bool aNewRes, QString aNewDesc)` verwendet werden, um einem Client mit der ID `aId` die Qualität mit der

7. Integration des Quellcodes in neue Projekte

Nummer `level` zuzuweisen. Sollte sich dabei um eine Qualität mit einer anderen Auflösung handeln, ist es notwendig, den Parameter `aNewRes` auf `true` zu setzen, damit die notwendigen Schritte eingeleitet werden¹².

7.3.2 Verwendung von alternativen Quellen

Ist das Quellmaterial, welches verwendet werden soll, nicht in einer lokalen Datei gespeichert, besteht die Möglichkeit, die Klasse `C_ReadFile` zu vererben und mit der Klasse `C_ServerHelper` neu zu verknüpfen, damit diese alternative Quelle genutzt werden kann.

Das Objekt `C_ServerHelper::mReadFile` ist zu diesem Zweck `public`. Zwischen dem Erzeugen des Objektes und der Initialisierung kann es entsprechend ausgetauscht werden:

```
10. (...)
11. C_ServerHelper *serverHelper = new C_ServerHelper(server);
12. serverHelper->mReadFile = new C_StreamReader();
13. if( !serverHelper->init() )
14. {
15. (...)
```

Die Klasse `C_StreamReader` sei in diesem Fall eine Kindklasse von `C_ReadFile`, welche die Daten von einer Netzwerkquelle liest. Um so etwas durch Vererbung von `C_ReadFile` zu implementieren, ist es notwendig, einige von Methoden zu überschreiben, welche in Abbildung 23 aufgelistet sind.

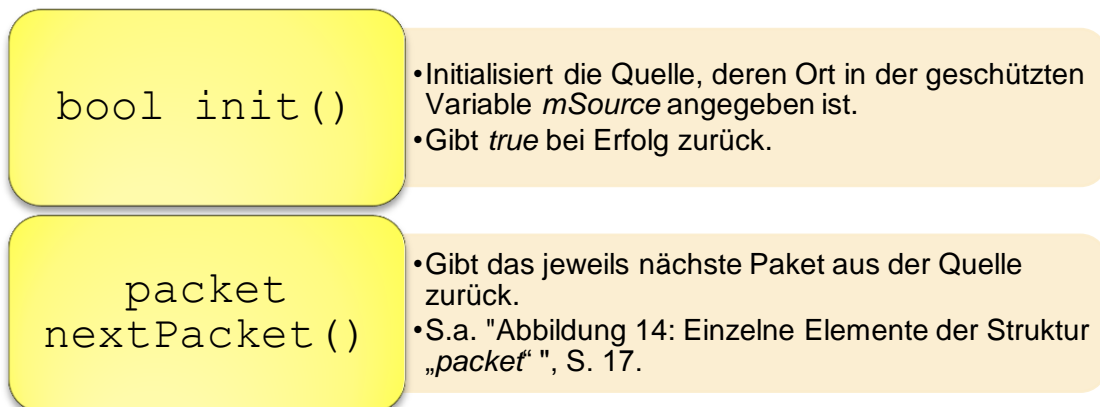


Abbildung 23: `C_ReadFile`-Methode, die bei Vererbung überschrieben werden müssen.

Die Methode `init()` sollte bestimmte Variablen Inhalte zuweisen, damit Methoden wie `numLayers()` oder `firstUnit(short)` sinnvolle Werte zurückliefern:

- `mLevels` – `QStringList` mit Beschreibungen von jedem Layer, wie z.B. „192x144@300kbps“

¹² S.a. Abschnitt „6.1.4 Wechsel der Qualität während der Übertragung“, S. 21

- *mFirstUnit* - *QMap<int, QList<QByteArray>>*, welche für jedes Layer die Rohdaten für die erste Unit der Quelle enthält. Diese werden für Qualitätswechsel während der Übertragung mit Auflösungswechsel benötigt.

Die Methode `nextPacket` liefert das jeweils nächste Paket zurück, welches an den Client geliefert werden soll. Ist kein Paket vorhanden, weil alle Daten gelesen wurden, reicht es, den Rohdaten-Anteil (`packet::data`) leer zu lassen. Das Objekt der Klasse *C_ServerHelper* reagiert entsprechend.

7.4 Verwendung der Klasse C_Client

Die Klasse *C_Client* kann mit relativ geringem Aufwand in ein neues Projekt integriert werden. Prinzipiell sind lediglich drei Schritte notwendig, welche in Abbildung 24 aufgezeigt werden.

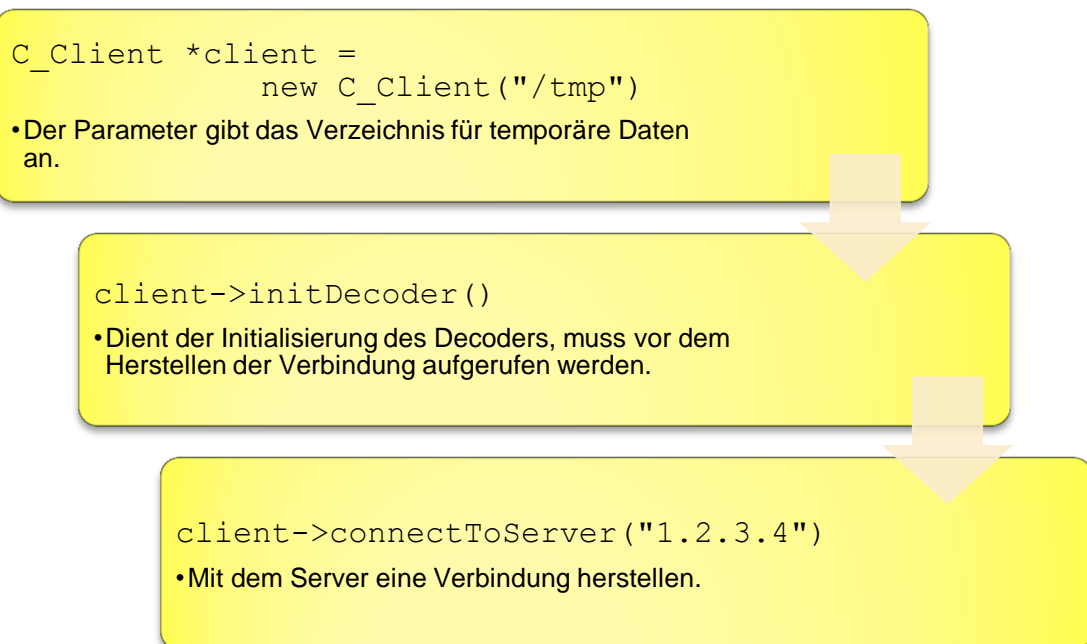


Abbildung 24: Notwendige Schritte bei Verwendung von C_Client.

Ein optionaler aber äußerst sinnvoller Schritt ist, einen Slot für das Signal `C_Client::newFile(QString)` anzulegen. Es wird aufgerufen, wenn der Decoder begonnen hat, Videodaten in die Datei, welche als Parameter angegeben ist, zu schreiben. Die Anwendung könnte in diesem Fall mit dem Abspielen beginnen.

Zusätzlich existieren noch einige weitere praktische Slots, die Aufschluss über die Dekodierungsgeschwindigkeit und ähnliches geben. Sie werden in der Quelltextdokumentation ausführlicher besprochen.

ANHANG A: ABBILDUNGSVERZEICHNIS

Abbildung 1: Aufbau eines Bildes mit mehreren Slices	1
Abbildung 2: Startcode im H.264-Bitstream.....	1
Abbildung 3: Beispielhafte Verteilung von Paketen in einem Stream.....	2
Abbildung 4: Einstellen der Präprozessordefinitionen im Visual Studio	5
Abbildung 5: Linker-Einstellungen im Visual Studio.....	6
Abbildung 6: Die Decoder-Testanwendung, ausgeführt unter Ubuntu 10.10.....	7
Abbildung 7: Liste der im Video enthaltenen Frames in der Decoder-Testanwendung....	8
Abbildung 8: Ausführen der Server-Anwendung unter Windows.....	8
Abbildung 9: Einstellungsdialogfenster der Serveranwendung unter Windows	9
Abbildung 10: Ausführung der Client-Anwendung unter Ubuntu 10.10	10
Abbildung 11: Einstellungsfenster der Client-Anwendung unter Ubuntu 10.10.....	11
Abbildung 12: Dekodierung mit der Klasse „CreaterH264AVCDecoder“ (vereinfacht)...	14
Abbildung 13: Dekodierungsvorgang unter Berücksichtigung der Qualität.....	15
Abbildung 14: Klassendiagramm der Server-Anwendung	16
Abbildung 15: Einzelne Elemente der Struktur „packet“	17
Abbildung 16: Klassendiagramm der Client-Anwendung.....	18
Abbildung 17: Stark vereinfachter Ablauf der <i>C_ServerHelper</i> -Klasse.....	20
Abbildung 18: Statusveränderungen der Clients auf Serverseite	21
Abbildung 19: Sortierung der Pakete als Flussdiagramm	22
Abbildung 20: Vereinfachter Qualitätswechsel als Flussdiagramm.....	23
Abbildung 21: Verwendung der <i>C_Decoder</i> -Klasse	25
Abbildung 22: <i>C_ReadFile</i> und <i>ReadBitstreamFile</i> im Vergleich.....	26
Abbildung 23: <i>C_ReadFile</i> -Methode, die bei Vererbung überschrieben werden müssen.	28
Abbildung 24: Notwendige Schritte bei Verwendung von <i>C_Client</i>	29

ANHANG B: INHALT DER BEILIEGENDEN CD

- Quelltext des Servers, des Clients, der Decoder-Testanwendung sowie der Encoder-Testanwendung.
- Diese Dokumentation in den Formaten PDF und DOCX.
- Die fertig modifizierte Version der JSVM-Software.
- Ausgewählte Testsequenzen.
- Diagramme dieser Präsentation im VSD-Format.

ANHANG C: NÜTZLICHE LINKS

- http://ip.hhi.de/imagecom_G1/savce/MPEG-Verification-Test/MPEG-Verification-Test.htm
Performancetest verschiedener Videosequenzen bei Enkodierung & Dekodierung mit JSVM.
- <http://www.itu.int/net/itu-t/sigdb/spevideo/VideoForm-s.aspx?val=102002641>
SVC-enkodiertes Videomaterial.
- <http://www.pudn.com/downloads211/sourcecode/multimedia/streaming/detail994055.html>
Chinesischer Mirror der JSVM-Software.
- <http://r2d2n3po.tistory.com/>
Koreanisch-englischer Blog mit stückhaften Informationen zum Thema.
- <http://r2d2n3po.tistory.com/attachment/ik090000000001.ppt>
Kleine Einführung in JSVM.