Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Realisierung eines Werkzeugs zum Segmentieren und Reassemblieren von H.264-codierten Videos

von

Nils Straßenburg

(Matr.#: 6648499)

Bachelorarbeit

in der Arbeitsgruppe Telekommunikation und Rechnernetze (TKRN)
Prof. Dr. rer. nat. Bernd E. Wolfinger

Fachbereich Informatik
Universität Hamburg

Hamburg
26. Oktober 2017

Erstgutachter: Prof. Dr. rer. nat. Bernd E. Wolfinger
Zweitgutachter: Alexander Beifuß

Danksagungen

Ich möchte an dieser Stelle all jenen danken, die mich im Rahmen dieser Bachelorarbeit, aber auch während meines gesamten Studiums unterstützt haben.

Für das Korrekturlesen möchte ich mich insbesondere bei meinen Eltern Sabine und Frank Straßenburg, aber auch bei meiner Tante Claudia Lillinger bedanken.

Ebenfalls möchte ich Herrn Prof. Dr. Wolfinger und Herrn Alexander Beifuß für wertvolle Anregungen beim Verfassen dieser Arbeit danken.

Abschließend danke ich meiner Familie für die moralische und finanzielle Unterstützung während meines gesamten Studiums, wobei ich auch hier insbesondere meine Eltern hervorheben möchte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACP	Auxiliary Coded Picture
AU	Access Unit
CSDP	Coded Slice Data Partition
CVS	Coded Video Sequence
DON	Decoding Order Number
DONB	Decoding Order Number Base
DOND	Decoding Order Number Difference
FU	Fragmentation Unit
IDR	Instantaneous Decoding Refresh
ISO/IEC	International Standardization Organization/International Electrotechnical Commission
ITU	International Telecommunications Union
MSE	Mean Squared Error
MTAP	Multi-time Aggregation Packet
MTU	Maximum Transmission Unit
MVC	Multiview Video Coding
NAL	Network Abstraction Layer
NALU	Network Abstraction Layer Unit
PCP	Primary Coded Picture
PPS	Picture Parameter Set
PSNR	Peak-Signal-to-Noise-Ratio
QoE	Quality of Experience
RST264	Reassemblation and Segmentation Tool for H.264 encoded Vidoes
RTCP	RTP Control Protocol
RTP	Real-time Transport Protocol
SDP	Session Description Protocol
SEI	Supplemental Enhancement Information
SPS	Sequence Parameter Set
SSIM	Structural Similarity
STAP	Single-time Aggregation Packet
SVC	Scalable Video Coding
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VCL	Video Coding Layer

1 Einleitung

1.1 Motivation

In den letzten Jahren hat das Streamen von Videos immer mehr an Bedeutung gewonnen. 2016 waren 73% des IP-Verkehrs Videodaten, bis 2021 wird diese Zahl nach Schätzungen von Cisco auf 82% steigen [7]. Dieser starke Zuwachs ist auch darauf zurückzuführen, dass der Trend immer weiter weg vom gewöhnlichen Fernsehen und hin zu Video-on-Demand und IPTV geht. Längst haben sich in diesem Bereich Anbieter wie YouTube, Netflix und Amazon etabliert und selbst die meisten TV-Sender stellen mittlerweile einen Großteil ihrer Inhalte auf Abruf in eigenen Mediatheken zur Verfügung. Bei all diesen Angeboten ist es für die Dienstleister besonders wichtig, eine angemessene Bildqualität beim Empfang der Videos zu ermöglichen. Ein Begriff, der in diesem Zusammenhang oft fällt, ist *Quality of Experience* (QoE). Er beschreibt etwas weiter gefasst die Nutzerzufriedenheit beim Wahrnehmen eines Multimediainhaltes.

Bei den oben beschriebenen Streaminganwendungen liegen keine Echtzeitanforderungen vor, da das Video gepuffert und über TCP übertragen werden kann, ohne dass sich dies schlecht auf die QoE auswirkt. Es nimmt jedoch auch die Nutzung von Diensten zu, bei denen Echtzeitanforderungen vorliegen, wie z.B. Videotelefonie. Über die Anwendung Skype ist dies schon lange einer Vielzahl von Nutzern möglich und wurde vor allem auf Desktop-Computern eingesetzt. Seit ein paar Jahren findet aber auch die Videotelefonie über Mobilgeräte immer mehr Anwendung. So hat z.B. Apple 2010 FaceTime eingeführt und seit 2016 ist die Videotelefonie auch über Messengerdienste wie WhatsApp möglich. 2021 werden voraussichtlich 13% des Internet-Video-Verkehrs Live-Verkehr sein [7]. Eine Garantie einer angemessen QoE ist in diesem Anwendungsgebiet wesentlich schwieriger, aber nicht weniger wichtig. Die Videodaten müssen mit sehr geringer Verzögerung übertragen werden [9] und gleichzeitig ist die Garantie einer angemessenen QoE gerade bei der Kommunikation mit anderen Personen von großer Bedeutung [14].

Aufgrund dieser Entwicklung wurden und werden eine Vielzahl an Ansätzen entwickelt, um die QoE beim Videostreaming zu gewährleisten. Neben adaptiven Verfahren, bei denen z.B. die Datenrate angepasst wird, wie in [6], existieren auch Herangehensweisen, bei denen die zu versendenden Pakete auf mehrere Pfade aufgeteilt werden [12]. Um solche Ansätze zu realisieren und vor allem auch zu analysieren, ist es notwendig, Videodaten für das Streamen aufzubereiten und sie dann je nach gewähltem Verfahren zu versenden. Hierbei müssen die Videodaten segmentiert werden. In jedem Fall erfolgt eine Segmentierung, um den Versand der Daten z.B. via RTP und UDP zu ermöglichen. Zusätzlich kann oder muss zuvor eine Segmentierung nach Frames erfolgen, wie z.B. in [5] oder [16].

In den meisten Arbeiten wird bei der Beschreibung der Durchführung des Experiments zur Analyse des jeweiligen Verfahrens nur beschrieben, dass die Videodaten in bestimmter Weise segmentiert wurden oder dass z.B. die Paketgrößen für ein Testszenario möglichst klein oder groß zu wählen sind. Obwohl bekannte Multimedia-Frameworks wie *ffmpeg* nicht in der Lage sind beim Streaming Paketgrößen individuell zu variieren oder Frame-

grenzen bei der Segementierung zu berücksichtigen, wird hierbei jedoch nicht dargelegt, wie die versendeten Videodaten passgenau für die jeweilige Untersuchung erzeugt werden. Eventuell werden bei den gewählten Ansätzen auch noch nicht alle Möglichkeiten ausgenutzt, wie z.B. der Einsatz individueller Paketgrößen für jede einzelne Sendung.

Hauptziel dieser Bachelorarbeit ist es, ein Werkzeug zu entwickeln, das Videodaten aus einer pcap-Datei einliest und hieraus passgenaue Pakete für einen Videostrom generiert. So kann für die Anwendung und Leistungsbewertung bestimmter Verfahren zur Sicherstellung der QoE auf das Werkzeug zurückgegriffen werden. Bei der internen Verarbeitung der Videodaten sollen hierbei die Grenzen von Frames berücksichtigt werden, so dass bei Bedarf jedes Frame individuell im Bezug auf die gewünschte Paketgröße verpackt und bereitgestellt werden kann. Zudem soll das Werkzeug die Möglichkeit bieten, zu Analysezwecken gewünschte Paketverluste mit einzubeziehen. Der Fokus dieser Arbeit soll darauf liegen, die Daten für ein Streaming mit Echtzeitanforderungen aufzubereiten. Hierbei kommt RTP zum Einsatz.

Der momentan noch am häufigsten verwendete Standard für Videocodierung ist H.264, aufgrund dessen er auch bei der Entwicklung des Werkzeugs herangezogen wird. Da zunächst eine Segmentierung in Frames vorgenommen werden soll und dies wie in [5] beschrieben eine nicht triviale Aufgabe darstellt, wird die Entwicklung eines solchen Verfahrens zur Sementierung der Daten in Frames ein zentraler Bestandteil des Werkzeugs und auch dieser Arbeit sein.

1.2 Gliederung der Arbeit

In Kapitel 2 werden zunächst grundlegende Inhalte erläutert, welche im Verlauf der Arbeit immer wieder aufgegriffen werden. Nach einer zeitlichen Einordnung und Erläuterung relevanter Begrifflichkeiten der H.264-Videocodierung wird auch ein Überblick über den Aufbau von Videos gegeben, die nach diesem Standard codiert sind. Zudem stellt dieses Kapitel die im Rahmen dieser Arbeit wichtigen Kommunikationsprotokolle im Zusammenhang mit Videostreaming dar.

Kapitel 3 beschreibt den Entwurf und die Entwicklung des Werkzeugs und führt die eingesetzten Hauptkomponenten auf. Zu jeder Hauptkomponente werden darauf folgend einzeln die für die Implementation relevanten theoretischen Inhalte dargelegt und zudem erläutert, wie die Komponente softwareseitig realisiert wurde. Die Vorstellung der Komponente, mit der Videos eingelesen werden können, erfolgt in Abschnitt 3.3. Im Rahmen dessen wird auch RFC 6184, der das RTP-Payloadformat für H.264-codierte Videos spezifiziert, behandelt. In Abschnitt 3.4 folgt die Komponente des Werkzeugs, mit der das Finden von Framegrenzen realisiert wird. Hierzu wird vertieft auf Inhalte des H.264-Standards eingegangen, um das vorgestellte Verfahren genauer zu erklären und dessen Anwendbarkeit sicherzustellen. Der Aufbau und die Realisierung der Komponente zum Verpacken und Versenden der Videodaten ist Inhalt von Abschnitt 3.5. Zum Abschluss wird in Abschnitt 3.6 auf die Modellierung von Paketverlusten zu Analysezwecken eingegangen.

Kapitel 4 hat mehrere Fallstudien des entwickelten Werkzeugs zum Gegenstand. Insbesondere dienen diese dem Nachweis, dass die von dem Werkzeug generierten Daten wieder korrekt decodiert werden können, wie auch zur Analyse von Paketverlusten geeignet sind. In diesem Kapitel erfolgt sowohl die Begutachtung der Laufzeit des Werkzeugs als auch eine Betrachtung des entstehenden Overheads. Zudem erfolgt die Untersuchung der Auswirkung von verschiedenen Paketgrößen bei der Übertragung eines Videos mit Paketverlusten.

Abschließend wird in Kapitel 5 ein Fazit gezogen und ein Ausblick gegeben, wie das Werkzeug weiter entwickelt und verbessert werden kann.

2 Grundlagen

Dieses Kapitel behandelt die Grundlagen, die für die Realisierung des Werkzeugs erforderlich sind. Hierzu gehören die für diese Arbeit relevanten Inhalte des H.264-Standards [19]. Weiterhin wird ein kurzer Einblick in grundlegende Kommunikationsprotokolle gegeben, welche bei der Echtzeitübertragung von Medien zum Einsatz kommen.

2.1 Videocodierung nach H.264

Einen für diese Arbeit wichtigen und relevanten Aspekt stellt der Videocodec H.264 dar. Neben einer zeitlichen Einordnung des Standards wird ebenfalls dargelegt, wie H.264-codierte Videos grundsätzlich aufgebaut sind.

2.1.1 Zeitliche Einordnung

Die bedeutendsten Standards zur Videocodierung, wie z.B. MPEG-2, MPEG-4, H.263 und H.264, veröffentlichten in der Vergangenheit immer die *International Telecommunications Union* (ITU) oder die *International Standardization Organization/International Electrotechnical Commission* (ISO/IEC). Teilweise haben diese beiden Organisationen aber auch zusammengearbeitet, so auch bei der Standardisierung von H.264. Bei der Entwicklung der Standards standen insbesondere der Aspekt der Echtzeit-Videokommunikation und der Aspekt der Verbreitung und das Broadcasting von Videos im Vordergrund. Im Folgenden sind die wichtigsten Neuerungen in zeitlicher Abfolge dargestellt. Abbildung 1, sowie die weitere zeitliche Einordnung sind sinngemäß [22] S. 1-6 entnommen.

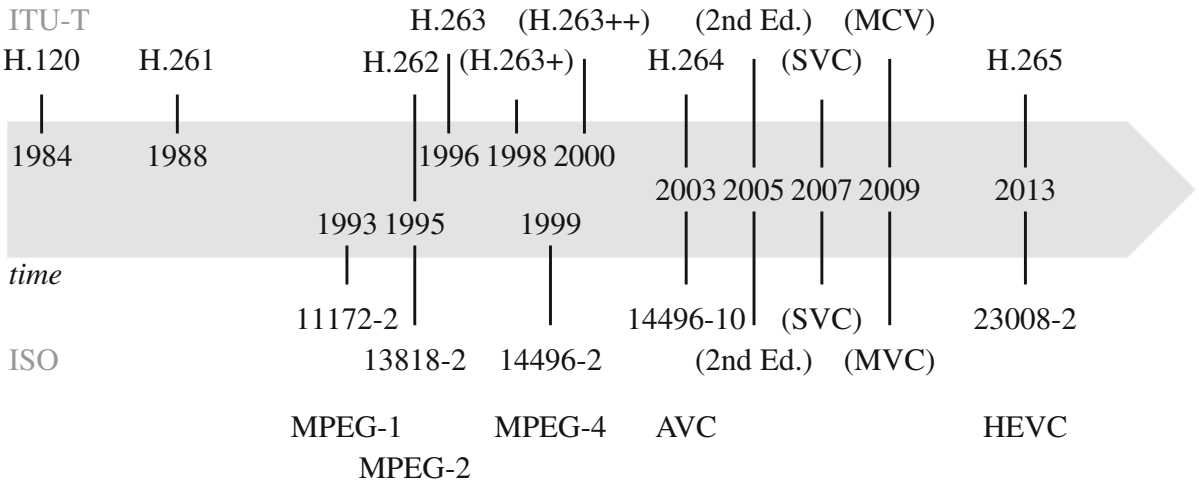

Abbildung 1: Zeitlicher Verlauf der Spezifikationen von ITU-T und ISO/IEC im Bereich Videocodierung

- **1984: H.120 (ITU)**:
  Dieser Standard stellt den ersten Standard für Videocodierung der ITU dar. Anwendung fand dieser vor allem im Bereich des „Standard-Fernsehens“ mit Bitraten im Bereich 1,5-2 *Mbit/s*.

- **1988: H.261 (ITU)**:
  Der H.261-Standard wurde unter dem Titel „Video codec for audiovisual services at p×64 kbit/s“ veröffentlicht und sollte zur Übertragung von Audio und Video über ISDN verwendet werden. Es ist der erste Standard, der auf dem *hybrid video coding scheme* basiert. Diese Struktur ist bis heute die Grundlage aller Videocodierungsstandards der ITU und der ISO/IEC. Über die Jahre wurden zwar einzelne Algorithmen angepasst, doch das grundsätzliche Vorgehen ist gleich geblieben und hat sich als eine geeignete Möglichkeit herausgestellt, Videos effizient in einen Bitstrom minimaler Größe umzuwandeln. Weitere Verbesserungen von H.261 gegenüber den vorherigen Standards waren eine effizientere Komprimierung, vor allem bei voneinander abhängigen Bildern und eine variable Bitrate, welche sich im Bereich von 40 *kbit/s* bis 2*Mbit/s* bewegt.

- **1993: MPEG-1 (ISO/IEC)**:
  Dieser Standard wurde insbesondere für die Verteilung und das Broadcasting von Videos entwickelt. Neben Komprimierungstechniken für Videos umfasst MPEG-1 auch die Standardisierung des sehr bekannten MP3-Formates.

- **1995: H.262 (ITU) / MPEG-2 (ISO/IEC)**:
  Dies ist der erste durch Zusammenarbeit der ITU und der ISO/IEC entwickelte Standard. Aus diesem Grund wurde er zum einen unter dem Namen H.262, aber auch unter dem Namen MPEG-2 von den jeweiligen Institutionen veröffentlicht. Einsatz fand dieser Standard bei DVDs, normalen TV-Übertragungen, aber auch bei HDTV. Ende 2012 wurde H.262 / MPEG-2 immerhin noch für 73% der TV-Übertragungen via Satellit verwendet.

- **1996: H.263 (ITU)**:
  H.263 wurde für den Anwendungsbereich der Videokommunikation entwickelt und war auf geringe Bitraten ausgelegt. Weiterentwickelt wurde er durch die Standards H.263+ und H.263++. Hierdurch war z.B. eine höhere Komprimierung möglich. Auch Mechanismen wie die Vorwärtsfehlerkorrektur wurden im Rahmen dieser Standards einbezogen. Ebenfalls wurde das erste Mal das Konzept der *Supplemental Enhancement Information* (SEI) eingeführt. Diese zusätzlichen Nachrichten sind nicht zwangsläufig zur Decodierung erforderlich, liefern jedoch Informationen, die die Decodierung des Videos vereinfachen. Dieses Konzept ist auch in den Folgestandards wiederzufinden.

- **1999: MPEG-4 (ISO/IEC)**:
  Dieser Standard wurde für rechteckige Bewegtbilder, aber auch für Standbilder konzipiert. Er weist viel Ähnlichkeit zu H.263 auf und wurde vor allem in der Variante *Advanced Simple Profile* verwendet. Dieses vereint Konzepte aus H.263 mit *global motion compensation* und *quarter-sample motion accuracy*. Beides sind Techniken, um Abhängigkeiten bezüglich Bewegungen zur besseren Komprimierung auszunutzen.

- **2003: H.264 (ITU) / AVC (ISO/IEC)**:
  Der H.264-Standard wurde von dem Joint Video Team entwickelt und entstand genau wie H.262 durch die Zusammenarbeit von ITU und ISO/IEC, daher wird auch die Abkürzung AVC (Advanced Video Coding) für H.264 verwendet. Da die Verteilung von Videos über das Internet und auch die Echtzeitvideokommunikation zum Zeitpunkt der Entwicklung stark zugenommen hatte und bis heute weiter zunimmt, standen die Ziele einer Verdopplung der Komprimierung und ein netzwerkfreundliches Design im Vordergrund. Angedachte Einsatzbereiche zum Zeitpunkt der Veröffentlichung waren unter anderem DVDs und Blu-rays, aber auch Video-on-demand und Multimedia-Streamingdienste. Bei der Entwicklung stand immer die Frage im Vordergrund: „how to handle this variety of applications and networks.“[21]. Im Rahmen dessen wurden in H.264 zwei Schichten (Layer) eingeführt. Zum einen der *Video Coding Layer* (VCL), in dem es um die effiziente Repräsentation des Videos geht. Zum anderen der *Network Abstraction Layer* (NAL), der die Daten des VCL in bestimmter Weise strukturiert. So wird z.B. durch das Bereitstellen von Headerinformationen der Einsatz in verschiedensten Anwendungsfällen, sowohl im Netzwerkbereich aber auch im Bezug auf die Speicherung auf verschiedenen Medien, ermöglicht und vereinfacht.

  Das Konzept der Aufteilung in NAL und VCL wird in Abbildung 2 verdeutlicht, welche [16] entnommen ist. Hier sind schematisch mehrere Transportmöglichkeiten von H.264-codierten Videos über das Internet unter Einbezug von VCL und NAL gegeben. Der VCL stellt die Ebene dar, welche am nächsten bei dem realen Video liegt. Nach der Codierung des rohen Videos in komprimierte Videodaten durch den VCL-Encoder werden diese Videodaten durch den NAL-Encoder in NAL *Units* (NALUs) unterteilt. Zum Transport dieser Daten über das Internet gibt es verschiedene Möglichkeiten. In Abbildung 2 wird neben der Variante für die Übertragung für Empfänger, die MPEG-2 unterstützen, auch die Übertragung nach H.320, H.324/M oder eine Versendung der Daten im *file format* aufgeführt. Ebenfalls wird die Möglichkeit des H.264-Videostreamings über RTP und IP genannt, auf die in Abschnitt 3.3.3 eingegangen wird.

  In den Jahren 2007 und 2009 wurde der Standard erweitert. Der Zusatz mit dem Namen *Scalable Video Coding* (SVC) beschreibt, wie für die Übertragung von Videos mehrere Bitströme genutzt werden können. Ein zweiter Zusatz, mit der Bezeichnung *Multi-view Video Coding* (MVC), erweitert den Standard für den Einbezug von mehreren Blickwinkeln, was z.B. bei 3D-Videos eingesetzt wird.

- **2013: H.265 (ITU) / HEVC (ISO/IEC)**:
  Auch diesen Standard veröffentlichten sowohl die ITU als auch die ISO/IEC. Anlass für die Entwicklung dieses Standards war die erhöhte Nutzung und Verbreitung von Videos in hoher Auflösung (1080p) oder sogar in Ultra HD (4k). Somit lag der Fokus insbesondere auf der Erhöhung der Videoauflösung und einer weiteren Verbesserung der Komprimierungstechniken. Gegenüber H.264 konnte eine 59-prozentige Einsparung bei der Bitrate erreicht werden. Langfristig wird dieses Format H.264 ablösen und z.B. in MacOS High Sierra (10.13) zum Einsatz kommen [3].

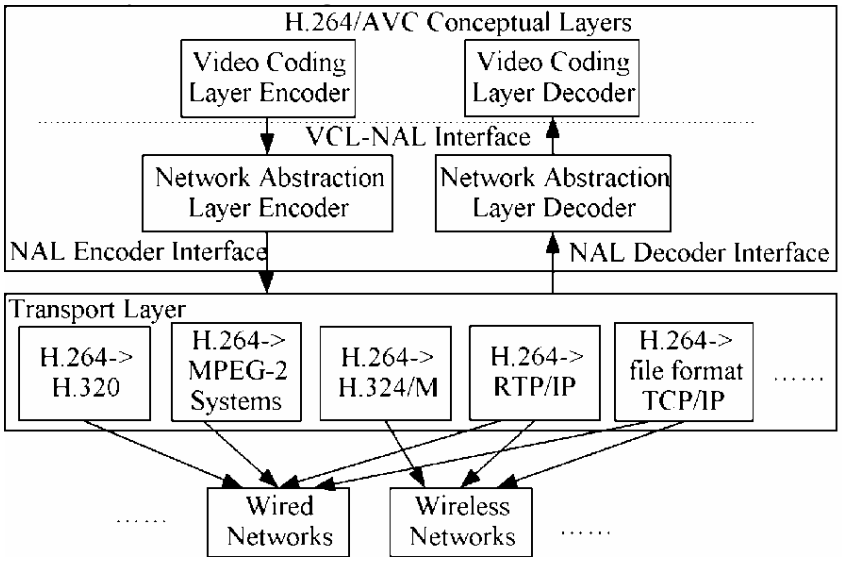

Abbildung 2: H.264 und Transportmöglichkeiten

Im Rahmen dieser Arbeit wird der Fokus auf H.264 gelegt. Grund hierfür ist vor allem, dass der H.264-Standard noch am weitesten verbreitet ist und somit aktuell häufig angewendet wird.

2.1.2 Hierarchischer Aufbau von Videodaten nach H.264

Im Folgenden wird ausgeführt, wie in H.264 codierte Videodateien grundsätzlich strukturiert sind (als Quelle wird [19] herangezogen). Hierbei bleiben die Besonderheiten von verschiedenen Videocontainerformaten oder Formaten für den Transport via RTP unberücksichtigt. Da im Kontext dieser Arbeit die Darstellung des Videos bis hin zu Blöcken oder sogar einzelnen Pixeln nicht relevant ist, liegt der Fokus auf dem oben beschriebenen NAL.

Ein Video, sei es als Datei abgelegt oder durch Streaming empfangen, besteht immer aus einer oder mehreren *coded video sequences* (CVSs). Diese bestehen aus mehreren *Access Units* (AUs), welche sich wiederum aus einer oder mehreren NALUs zusammensetzen. Diese NALUs enthalten dann entweder die eigentlichen Videodaten oder zur Decodierung benötigte Zusatzinformationen. Ein beispielhafter Aufbau eines Videos mit den oben genannten Komponenten ist Abbildung 3 zu entnehmen. Die einzelnen Komponenten werden im Folgenden genauer erläutert.

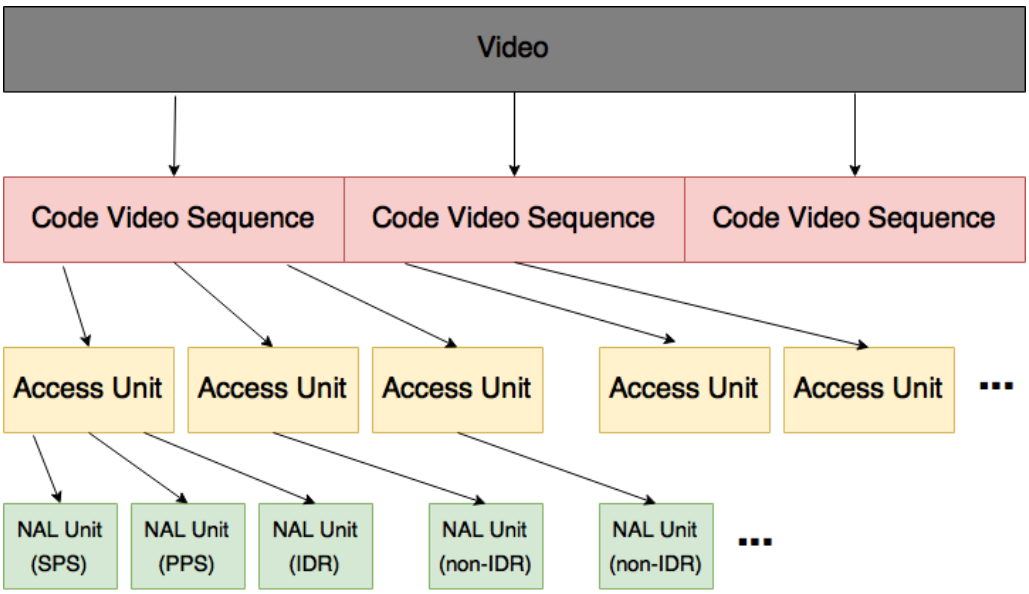

Abbildung 3: Hierarchischer Aufbau von Videodaten nach H.264

2.1.2.1 Coded Video Sequence

Eine einzelne CVS beinhaltet mehrere AUs. Hierbei startet sie immer mit einer bestimmten Form einer AU, die in jedem Fall eine *Instantaneous Decoding Refresh* (IDR) NALU enthält. Aufgrund dessen wird eine AU dieser Form als IDR AU bezeichnet. Gefolgt wird diese von mehreren AUs, die keine IDR NALUs enthalten und somit als non-IDR AU bezeichnet werden.
Eine IDR NALU kann decodiert werden ohne hierbei andere NALUs, die codierte Videodaten enthalten, zu referenzieren. Nachfolgende AUs derselben CVS (immer non-IDR AUs) referenzieren die erste AU mit der IDR NALU.

2.1.2.2 Access Units

AUs fassen mehrere NALUs zusammen. Hierbei wird eine festgelegte Reihenfolge der NALUs eingehalten. Ein für den Kontext dieser Arbeit zentraler Sachverhalt im Bezug auf AUs ist folgender: **„The decoding of each access unit results in one decoded picture.“** [21] Mit anderen Worten: Das Bestimmen von Framegrenzen in einer Folge aus NALUs ist gleichzusetzen mit dem Finden von Grenzen von AUs. Aufgrund dieser Tatsache wird der Aufbau von AUs und das Separieren dieser in Abschnitt 3.4.2 behandelt.

2.1.2.3 NAL Units

NALUs stellen die kleinste Einheit auf dem NAL dar und können Videodaten oder Zusatzinformationen enthalten. Diesen Nutzdaten der NALU ist ein NALU-Header vorangestellt (vgl. Abbildung 4), der die Größe von einem Byte hat.

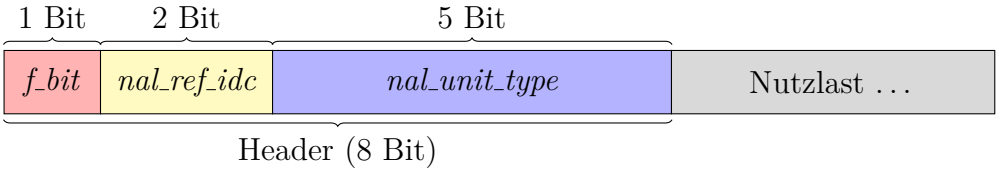

Abbildung 4: Aufbau einer NAL Unit

Er setzt sich folgendermaßen zusammen (die Reihenfolge entspricht auch der im Bitmuster):

- ***forbidden_zero_bit*** (1 Bit):
  Über dieses Feld wird im ITU-Standard [19] nur gesagt: „shall be equal to 0“. Ist dies nicht der Fall, weist es in der Regel auf einen Fehler bei der Codierung hin. In Abbildung 4 ist dieses Feld mit *f_bit* abgekürzt.
- ***nal_ref_idc*** (2 Bit):
  Dieses Feld trifft eine Aussage darüber, wie relevant die aktuelle NALU ist. Dem Standard nach sollte eine NALU vom Typ *Coded slice of an IDR picture* als *nal_ref_idc* nicht den Wert 0 gesetzt haben. Gleiches gilt z.B. für NALUs vom Typ *Sequence Parameter Set* (SPS) und *Picture Parameter Set* (PPS). Die besondere Bedeutung dieser NALUs ergibt sich daraus, dass sie in der Regel von vielen anderen NALUs referenziert werden.
- ***nal_unit_type*** (5 Bit):
  Dieses Feld spezifiziert die Art der Nutzlast. Da die Größe dieses Feldes fünf Bit beträgt, ergeben sich 32 mögliche Werte, die folgendermaßen unterteilt sind:
  - Die Werte 0 und 24 bis 31 sind *unspecified*.
  - Die Werte 17,18, 22 und 23 sind *reserved*.
  - Die Werte 1 bis 21 (ohne 17 und 18) sind im Standard mit speziellen Typen belegt.

2.1.3 NAL Unit Typen

Nachdem der grundsätzliche Aufbau eines nach H.264 codierten Videos und dessen Unterkomponenten beschrieben wurde, wird im Folgenden auf die verschiedenen Typen von NALUs und deren Verwendung eingegangen. Als Quelle dient hierfür ebenfalls [19].

Eine NALU kann Daten vom Typ *VCL* oder *non-VCL* enthalten. Im Standard heißt dieses Kriterium NALU *type class*. Anhand dessen lässt sich eine feinere Unterteilung des Wertebereichs von 1 bis 21 vornehmen (vgl. Abbildung 5):

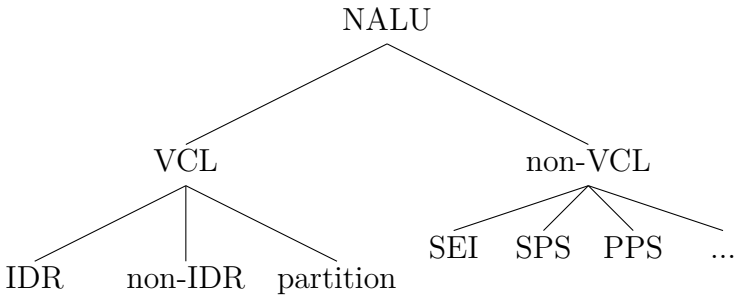

Abbildung 5: Übersicht: NALUs

- ***nal_unit_type*** $\in \{1, \ldots, 5\}$:
  NALUs aus diesem Wertebereich sind *VCL NAL units*. *NAL units* dieser Art enthalten Videodaten in codierter Form. Unter diesen Typen lässt sich eine weitere Unterteilung vornehmen:
  - ***nal_unit_type*** $= 1$:
    Dies ist eine NALU vom Typ *Coded slice of an non-IDR picture*. Eine solche NALU enthält codierte Videodaten. Ein komplettes Frame kann nur aus einer oder mehreren NALUs diesen Typs bestehen. Hierbei referenzieren diese NALUs aber andere NALUs, z.B. NALUs mit *nal_unit_type* $= 5$.
  - ***nal_unit_type*** $\in \{2, 3, 4\}$:
    Hier liegt eine NALU vom Typ *Coded Slice Data Partition* (CSDP) vor. Diese NALUs enthalten nur eine Untermenge von Daten, die ein *slice*, also einen Teil eines Bildes, repräsentieren. CSDP NALUs enthalten also im Endeffekt eine Teilmenge an codierten Videodaten, die für die Decodierung eines Frames benötigt werden.
  - ***nal_unit_type*** $= 5$:
    In diesem Fall handelt es sich um eine NALU vom Typ *Coded slice of an IDR picture*. *IDR* steht hierbei für *instantaneous decoding refresh*. Demnach kann sie, im Gegensatz zu allen anderen VCL NALUs, ohne das Referenzieren anderer VCL NALUs decodiert werden und wird zudem in der Regel von nachfolgenden NALUs in der gleichen CVS referenziert.

  Die für diese Arbeit wichtigsten NALUs in diesem Wertebereich sind vom Typ *Coded slice of a non-IDR picture* (*nal_unit_type = 1*) und *Coded slice of an IDR picture* (*nal_unit_type* $= 5$).
- ***nal_unit_type*** $\in \{6, \ldots, 21\}$
  NALUs aus diesem Wertebereich sind non-VCL NALUs. Diese NALUs enthalten Zusatzinformationen und keine codierten Videodaten. Die für diese Arbeit wichtigen NALUs des obigen Wertebereiches werden im Folgenden aufgeführt:

- ***nal_unit_type*** = 6
  NALUs mit diesem Typ nennen sich SEI. Sie beinhalten Zusatzinformationen, die nicht zwangsläufig zur Decodierung des Videos benötigt werden. Sie liefern jedoch Informationen, die die Decodierung des Videos vereinfachen.
- ***nal_unit_type*** = 7
  Es handelt sich hierbei um eine NALU vom Typ SPS. Sie enthält Syntaxelemente, auf die sich eine oder mehrere CVS beziehen.
- ***nal_unit_type*** = 8
  Diese NALU vom Typ PPS enthält Syntaxelemente, auf die sich ein oder mehrere codierte Bilder beziehen.
- ***nal_unit_type*** = 9:
  NALUs diesen Typs werden mit *Access unit delimiter* benannt. Vor allem wird diese NALU dazu verwendet, den Anfang einer neuen AU zu kennzeichnen (vgl. Abschnitt 7.4.2.4 in [19]). Eine Verwendung dieser beim Videostreaming erfolgt allerdings in der Regel nicht. Hierdurch wird die Detektion der Grenzen zwischen zwei AUs zu einer nicht trivialen Aufgabe und wird in Abschnitt 3.4.2 behandelt.

2.1.3.1 Zuordnung von VCL NALUs, PPSs und SPSs

Im Allgemeinen gilt, dass eine VCL NALU ein PPS referenziert. Ein PPS referenziert wiederum ein SPS. Die Referenzierung wird jeweils durch IDs ermöglicht. Das Konzept der *parameter sets* und die damit verbundene Referenzierung mittels IDs, wird in [21] als eines der neuen Features von H.264 genannt, welche für mehr Robustheit und höhere Effizienz sorgen.

2.2 Überblick über relevante Kommunikationsprotokolle

Hier wird ein kurzer Einblick zu den jeweiligen Protokollen gegeben, welche für die Übertragung von Echtzeitmedien besonders relevant sind. Im Rahmen dieser Arbeit werden die Informationen zu diesen Protokollen hauptsächlich für die Generierung eines Videostroms benötigt. Als Quellen für diesen Abschnitt werden neben den RFCs [18] und [10], hauptsächlich [13] und [4] herangezogen.

2.2.1 User Datagram Protocol

Das *User Datagram Protocol* (UDP) ist ein Protokoll, welches nach dem TCP/IP-Referenzmodell der Transportschicht zugeordnet wird. Es zeichnet sich vor allem durch seine Leichtgewichtigkeit und einen geringen Overhead bezüglich Verwaltung und Kontrollinformationen aus. Es ist verbindungslos, weshalb kein Handshake benötigt wird.

Die benannte Leichtgewichtigkeit zeigt sich vor allem bei dem Protokollheader (vgl. Abbildung 6). In diesem sind lediglich die Felder zur Angabe des Ziel- bzw. Quellports, eine Längenangabe und eine Prüfsumme enthalten, wodurch sich eine Headergröße von 8 Byte ergibt. Zum Vergleich: Der Protokollheader des zuverlässigen Pendants, dem *Transmission Control Protocol* (TCP) hat eine Größe von 20 Byte.

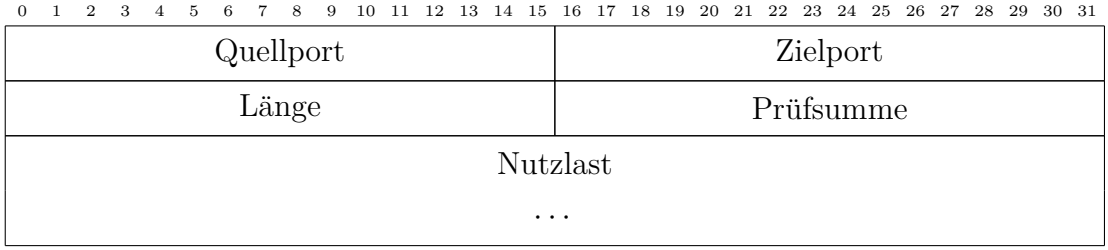

Abbildung 6: UDP-Header

Die Leichtgewichtigkeit und der geringe Overhead sind nur möglich, weil der Datenverkehr via UDP unzuverlässig ist und so weniger Kontrollinformationen und Headerfelder benötigt werden. Es gibt weder eine Garantie, dass das versendete Datenpaket beim Empfänger eintrifft, noch werden Zusicherungen bezüglich der Reihenfolge gemacht. Auch auf eine Überlast- oder Flusskontrolle wie bei TCP wird bei UDP verzichtet.

Sollen Teile dieser Aspekte bezüglich der Zuverlässigkeit trotz Übertragung mittels UDP zugesichert werden, so müssen diese auf der Anwendungsebene realisiert werden. Dass diese Möglichkeit teilweise wahrgenommen wird, zeigt sich in Abschnitt 2.2.3.

Aufgrund der oben beschriebenen Eigenschaften wird UDP insbesondere im Bereich der Echtzeitübertragung von Multimediadaten genutzt. Beispiele hierfür sind Voice over IP, aber auch Videostreaming mit Echzeitanforderungen, wie z.B. Videokonferenzen. In diesen Gebieten findet die Leichtgewichtigkeit von UDP optimale Anwendung und auch die oben beschriebenen Abstriche, welche bei dessen Verwendung gemacht werden müssen, haben so gut wie keinen Einfluss. Grund hierfür ist, dass es bei der Übertragung von Echtzeitmedien hauptsächlich darauf ankommt, die Daten möglichst schnell und mit geringem Overhead zu übertragen. Paketverluste sind in gewissen Maßen akzeptabel, aber vor allem die Neuübermittlung von verloren gegangenen Datenpaketen, wie es bei TCP der Fall ist, ist in diesem Einsatzgebiet wenig sinnvoll. Denn die Zeit, die für diesen Vorgang benötigt wird, ist im Kontext der Echtzeitübertragung ggf. zu lang. Zwar wird z.B. bei Youtube oder verbreiteten Streamingdiensten wie AmazonPrime HTTP bzw. TCP für die Übertragung verwendet, doch handelt es sich hier nicht um wirkliche Echtzeitanforderungen, denn eine kurze Wartezeit vor Beginn des Videos stört den Nutzer nicht. Diese Verzögerung reicht allerdings aus, um evtl. verloren gegangene Pakete neu zu übertragen. Diese Wartezeit und die damit verbundene Verzögerung in der Übertragung ist z.B. bei einer Videokonferenz nicht akzeptabel, weshalb hier UDP das Mittel der Wahl ist.

2.2.2 Real-time Transport Protocol

Das *Real-time Transport Protocol* (RTP) wird im TCP/IP-Schichtenmodell der Anwendungsschicht zugeordnet. Entwickelt wurde es für die Übertragung von Echtzeitmedien und ist heute weit verbreitet (siehe [13] S. 657). Definiert wurde es in RFC 3550 [18], welcher für diesen Abschnitt ebenfalls als Quelle herangezogen wurde.

RTP wird in den meisten Fällen zusammen mit UDP genutzt, was im Kontext von Echtzeitübertragung von Medien sehr sinnvoll ist. Ähnlich wie bei UDP wird auch durch RTP wenig zugesichert. So wird z.B. nicht zugesichert, dass die RTP-Pakete in einem konstanten zeitlichen Abstand eintreffen oder bis zu einem bestimmten Zeitpunkt zur Verfügung stehen. Auch andere QoS-Anforderungen werden seitens RTP nicht zugesichert. Es wird ebenfalls nicht zugesichert, dass überhaupt alle versendeten RTP-Pakete ankommen und auch nicht, dass die Pakete in der richtigen Reihenfolge eintreffen.

Wie dennoch eine angemessene Darstellung der zu übertragenden Medien sichergestellt wird, geht zum einen aus dem Header hervor (vgl. Abbildung 7), zum anderen aus Abschnitt 2.2.3.

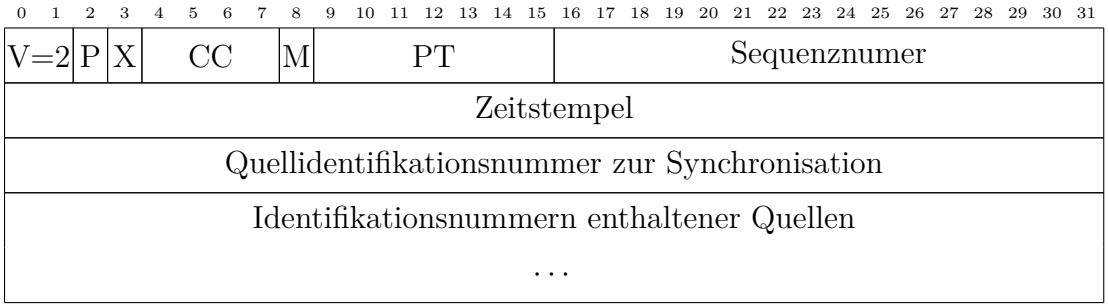

Abbildung 7: RTP-Header

Die wichtigsten Felder des Headers und dessen Zweck wird im Folgenden kurz erläutert:

- **Nutzlasttyp (*PT*)** (7 Bit):
  Durch dieses Feld wird die für die Nutzlast verwendete Codierung angegeben. Zum einen ist der Empfänger so in der Lage, die Daten korrekt zu decodieren. Zum anderen kann so auch während der Übertragung von Medien die Codierung geändert werden und somit die Qualität der Übertragung an die zur Verfügung stehende Datenrate angepasst werden.

- **Sequenznummer** (16 Bit):
  Die Sequenznummer wird seitens des Senders bei jedem neuen RTP-Paket inkrementiert. Zum einen ist der Empfänger durch dieses Feld in der Lage die korrekte Reihenfolge der Pakete wieder herzustellen. Zum anderen kann ein RTP-Paket mit einer gewissen Sequenznummer, wenn es nach einem festgelegten Timeout nicht angekommen ist, als verloren gegangen interpretiert werden.

- **Zeitstempel** (32 Bit):
  Der Zeitstempel bezieht sich jeweils auf das erste Byte der RTP-Nutzlast. Eingesetzt wird dieses Feld vor allem dazu, zeitliche Schwankungen bei der Übertragung auszugleichen. So können z.B. die Bilder eines Videostroms trotz eventueller Übertragungsschwankungen in den richtigen zeitlichen Abständen abgespielt werden.
- **Quellidentifikationsnummer zur Synchronisation** (32 Bit):
  Dieses Feld dient zur Identifikation der Quelle der Nutzlast. Dies ist z.B. wichtig, wenn für die Übertragung von Audio und Video zwei RTP-Ströme verwendet werden. Der Wert für dieses Feld wird jeweils zufällig vom Sender zugeordnet.

2.2.3 RTP Control Protocol

Das *RTP Control Protocol* (RTCP) kann ergänzend zu RTP eingesetzt werden, um die Qualität der Übertragung zu kontrollieren. Findet die Übertragung von Audio- oder Videodaten auf Port x statt, so werden die RTCP-Pakete auf Port $x+1$ übertragen. Diese enthalten statistische Daten, welche auf der Anwendungsschicht zur Verbesserung der Qualität genutzt werden können. Gesendet werden solche Pakete in der Regel sowohl vom Sender, als auch vom Empfänger.

Die Pakete, die der Sender versendet, werden *Sender Reports* genannt und enthalten neben der Quellidentifikationsnummer vor allem Informationen wie die Anzahl der bisher gesendeten Pakete und Bytes. Ebenfalls sind Informationen zum Timing, wie Timestamps der Medien und Informationen zur *wall clock time*, enthalten. Die *wall clock time* ist ein für alle Medien gemeinsamer Zeitstempel, der synchron zu der Realzeit läuft. Medien, die real zur gleichen Zeit aufgenommen wurden, bekommen so die gleiche *wall clock time*. So können diese Angaben genutzt werden, um z.B. jeweils separate Audio- und Videoströme zu synchronisieren. Die genaue Struktur eines solchen Pakets mit allen Feldern ist in [18] im Abschnitt 6.4.2 beschrieben.

Die Pakete, die von Empfängern versendet werden, werden *Receiver Reports* genannt. Diese enthalten ebenfalls die Quellidentifikationsnummer. Darüber hinaus beinhalten sie aber z.B. auch Informationen darüber, wie viele Pakete verloren gegangen sind, die letzte empfangene Sequenznummer und einen Wert, welcher die Variation der Paket-Ankunftsraten beschreibt. Die genaue Struktur eines solchen Pakets mit allen Feldern ist in [18] im Abschnitt 6.4.1 beschrieben.

2.2.4 Session Description Protocol

Das *Session Description Protocol* (SDP) wurde zum Einladen und zur Beschreibung von Multimediasessions entwickelt (vgl. [10] S. 1) und findet auch im Bereich des Videostreamings Anwendung. Die Beschreibung einer Session nach SDP ist in drei Teile aufgeteilt: *Session-Level*, eine Zeile für eine Zeitangabe und *Media-Level*.

In den meisten Fällen sind die folgenden Zeilen enthalten. Zum besseren Verständnis ist zu jeder Zeile deren Syntax angegeben. Die Erläuterung jedes einzelnen Syntaxelements ist jedoch nicht zielführend. Es wird somit nur erläutert, welche Informationen in den einzelnen Zeilen grundsätzlich enthalten sind:

- ***v*** *(Protocol Version)*: *v=<version>*
  Diese Zeile enthält Angaben zur verwendeten Version, wobei diese nach dem hier referenzierten RFC immer die Form *v=0* hat.
- ***o*** *(Origin)*: *o=<username> <sess-id> <sess-version> <nettype> <addrtype> <unicast-address>*
  In dieser Zeile wird die Quelle des zu übertragenden Inhalts spezifiziert.
- ***s*** *(Session Name)*: *s=<session description>*
  Diese Zeile beschreibt den Namen der Session als String.
- ***c*** *(Connection Data)*: *c=<nettype> <addrtype> <connection-address>*
  In dieser Zeile wird angegeben, wohin die Medien übertragen werden sollen.
- ***t*** *(Timing)*: *t=<start-time> <stop-time>*
  Diese Zeile gibt Start- bzw. Endzeitpunkt der Session an. Oft werden für die beiden Zeitangaben die Werte 0 0 verwendet. Auf diese Weise wird spezifiziert, dass die Session direkt nach Aufbau beginnt und zu beliebiger Zeit beendet werden kann.
- ***m*** *(Media Descriptions)*: *m=<media> <port> <proto> <fmt> ...*
  Für jedes zu übertragende Medium der Session existiert eine Zeile dieser Art, in der selbiges spezifiziert wird.
- ***a*** *(Attributes)*: *a=<attribute> :<value>* oder *a=<attribute>*
  Auch von dieser Zeile kann es beliebig viele geben. Sie können genutzt werden, um Attribute des *Session-Levels* oder des *Media-Levels* zu beschreiben. Oft wird durch solche Zeilen spezifiziert, in welcher Form die in den *m*-Zeilen angegebenen Medien übertragen werden können.

In dieser reduzierten verallgemeinerten Darstellung gehören die Zeilen *v,o,s* und *c* zum *Session-Level* und die Zeile mit der Bezeichnung *m* zum *Media-Level*. Zeilen mit der Bezeichnung *a* können, wie oben schon erwähnt, sowohl die Session als auch die Medien genauer spezifizieren.

Ein Beispiel für den typischen Aufbau einer SDP-Datei im Zusammenhang mit Videostreaming ist in Abbildung 8 dargestellt.

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=No Name
c=IN IP4 127.0.0.1
t=0 0
a=tool:libavformat 57.56.101
m=video 4010 RTP/AVP 96
a=rtpmap:96 H264/90000
a=fmtp:96 packetization-mode=1
```

Abbildung 8: Beispiel einer typischen SDP-Datei

In diesem Beispiel wurde die Version 0 gewählt. Bei der Spezifikation der Quelle sind *username*, die *session-id* und die *session-version* nicht gesetzt. Als *nettype* ist mit IN Internet angegeben und als Quelladresse die IPv4 Adresse 127.0.0.1 (localhost). Diese Adresse ist auch das Ziel des Videostroms. Für den Start- und Endzeitpunkt wurden die oben schon erwähnten, oft genutzten Werte 0 0 gewählt. Durch die erste a-Zeile wird die Software bzw. die Bibliothek beschrieben, mit der der Videostrom erstellt wurde. Durch die letzten drei Zeilen wird spezifiziert, dass ein Video mittels RTP übertragen werden soll, wobei dies H.264 codiert und *packetization-mode* 1 verwendet werden soll.

3 Entwicklung eines Werkzeugs zum Segmentieren und Reassemblieren von H.264-codierten Videos

Im folgenden Kapitel wird zunächst der gewünschte Funktionsumfang des zu entwickelnden Werkzeugs erläutert. Im Anschluss folgt die Beschreibung der grundsätzlichen Strukturierung des Werkzeugs.

3.1 Funktionalität des Werkzeugs

Das Ziel dieser Arbeit ist es ein Werkzeug zu entwickeln, mit dem es möglich ist, H.264-codierte Videos einzulesen und zu streamen, hierbei aber auch die zu versendende Nutzlast angemessen und individuell zu segmentieren.

Beim Einlesen und Verarbeiten des Videos sowie beim Streamen sollen folgende Aspekte berücksichtigt werden:

- Das Werkzeug soll eine weitestgehend universelle Möglichkeit bieten, Videos einzulesen. Diese soll nicht auf ein bestimmtes Videocontainerformat, eine bestimmte Auflösung oder Framerate beschränkt sein. Um die Analyse der Daten zu vereinfachen, werden beim Einlesen des Videos die Audiodaten nicht berücksichtigt.
- Die Daten sollen zunächst so unterteilt werden, dass die Daten, die zu der Decodierung jeweils eines Frames benötigt werden, gemeinsam verarbeitet werden. Hintergrund dieser Anforderung ist, dass es beim Streaming über mehrere Pfade möglich sein soll, die Daten frameweise aufzuteilen.
- Der Stream bzw. die Paketierung erfolgt nach dem in [18] spezifizierten Paketierungsmodus 1 (*Non-Interleaved Mode*), da er sich am besten für die Übertragung von Videos mit Echtzeitanforderungen eignet. In Abschnitt 3.3.3.6 wird dies näher erläutert.
- Die Paketgröße der durch das Werkzeug generierten RTP-Pakete soll frei wählbar sein. Hierbei soll nicht einfach nur zu Beginn ein konstanter Wert für diese Größe festgelegt werden, vielmehr soll die Größe jedes einzelnen RTP-Paketes frei wählbar sein. Hintergrund dieser Anforderung ist, dass es beim Videostreaming über mehrere Pfade möglich sein soll, die Resourcen passgenau und unter Einbezug momentaner Umstände (z.B. Auslastung von Leitungen) zu nutzen. Hierzu ist eine für alle RTP-Pakete festgelegte Größe ungeeignet.
- Zu Analysezwecken bezüglich Paketverlusten soll es möglich sein, gezielt Pakete, z.B. nach einer stochastischen Verteilung oder einem anderen geeigneten Modell, zu vernichten.

Um im weiteren Verlauf dieser Arbeit von einem konkreten Werkzeug sprechen zu können und da die Hauptaufgaben des zu entwickelnden Werkzeugs die Reassemblierung, sowie Segmentierung von H.264-codierten Videos sind, wird diesem der Name *Reassemblation and Segmentation Tool for H.264 encoded Vidoes* gegeben. Als Abkürzung wird RST264 verwendet.

3.2 Hauptkomponenten des RST264

Das Werkzeug RST264 wird in folgende Komponenten unterteilt, um die einzelnen Anforderungen strukturiert zu erfüllen.

- **Komponente zum Einlesen des Videos (RST264-RV)**:
  Das RST264-Werkzeug soll eine möglichst universelle Möglichkeit bieten, Videos einzulesen und diese im Anschluss zu verarbeiten. Hierbei werden die Daten so eingelesen, dass eine Unterteilung in einzelnen NALUs vorliegt. Da die Hauptaufgabe dieser Komponente des RTS264-Werkzeugs das Einlesen der Videodaten ist, wird für diese die Bezeichnung RST264-RV verwendet, wobei RV für *read video* steht.
- **Komponente zum Erkennen von Framegrenzen (RST264-FD)** :
  Das Video liegt nach dem Einleseprozess bereits in NALUs unterteilt vor. Aufgabe dieser Komponente ist es, die NALUs frameweise zusammenzufassen, weshalb für diese die Bezeichnung RST264-FD, mit FD für *frameborder detection* verwendet wird. In Abschnitt 2.1 wurde festgestellt, dass diese vorzunehmende Unterteilung eine Unterteilung nach AUs ist.
- **Komponente zum Streamen eines Videos (RST264-SV)**:
  Die Aufgabe dieser Komponente ist es, die zuvor aufbereiteten Daten für einen Videostrom vorzubereiten und diese dann zu versenden. Als Bezeichnung für diese Komponente wird RST264-SV, mit SV für *stream video* gewählt.
- **Komponente zum Simulieren von Paketverlusten (RST264-PL)**:
  Aufgrund einer Anforderung an das Werkzeug RST264 kann die Komponente zu Analysezwecken gezielt Pakete vernichten. Es soll möglich sein, auf bestimmte Art und Weise Paketverluste auf Grundlage stochastischer Modelle zu simulieren. Dabei wird eine durchgeführte Simulation von Paketverlusten dokumentiert, oder aber eine bereits dokumentierte Simulation von Paketverlusten erneut durchgeführt. Für diese Komponente wird die Bezeichnung RST264-PL, mit PL für *packet loss* verwendet.

Wie die oben beschriebenen Komponenten zusammenarbeiten, ist in Abbildung 9 dargestellt. Zunächst werden durch die Komponente RST264-RV Videodaten eingelesen und in Form von NALUs ausgeben. Diese werden dann durch die Komponente RST264-FD in AUs zusammengefasst. Unter Berücksichtigung der Parametrisierung der Funktion zum Verpacken der AUs seitens des Nutzers, werden durch die Komponente RST264-SV RTP-Pakete generiert, welche anschließend versendet werden können. Optional können durch

die Komponente RST264-PL Paketverluste simuliert werden und unter Zuhilfenahme dieser Simulation eine Teilmenge der ursprünglichen RTP-Pakete erzeugt werden. Diese stellt die Menge an Paketen dar, die beim Empfänger eintreffen. Der optionale Teil ist in Abbildung 9 grau dargestellt.

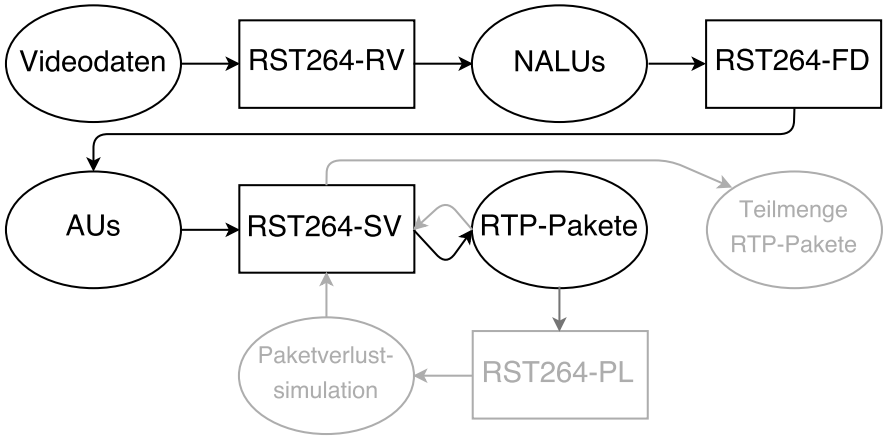

Abbildung 9: Zusammenarbeit der Hauptkomponenten des Werkzeugs RST264

3.3 Komponente zum Einlesen eines Videos (RST264-RV)

Durch diese Komponente des Tools RST264 wird ein Video eingelesen, als Resultat liegt das Video danach in Form von NALUs vor.

Dies soll grundsätzlich durch die folgenden Zwischenschritte erfolgen:

- Streamen des einzulesenden Videos
- Aufzeichnen des Videostroms
- Einlesen des gespeicherten Videostroms

3.3.1 Streamen des einzulesenden Videos

Im Rahmen dieser Arbeit wird das Video mittels *ffmpeg*, einem Multimedia-Framework, gestreamt. Hierbei wird folgender Befehl verwendet:

ffmpeg -re -i <quell_video> -vcodec libx264 -f rtp -an rtp://<Ziel_IP>:<Ziel_Port>

Zunächst wird mittels *-re* dafür gesorgt, dass das Video in seiner nativen Framerate eingelesen wird. Laut der Dokumentation *ffmpeg*s [1] soll dieser Teilbefehl eingesetzt werden, um einen *live input stream* zu simulieren, wenn das Video einer Datei entnommen wird. Anschließend wird mittels *-i <quell_video>* das Quellvideo angegeben. Durch die

Angabe von *-an rtp://<Ziel_IP>:<Ziel_Port>* wird das Ziel des Videostroms spezifiziert. Weiterhin wird durch *-vcodec libx264 -f rtp* erwirkt, dass das Video nach dem H.264-Standard codiert und der Videostrom in Form von RTP-Paketen realisiert wird.

3.3.2 Aufzeichnen und Einlesen des Videostroms

Während des Streamingvorganges wird eine Aufzeichnung gestartet und diese als Datei mit der Endung *pcap* gespeichert.

Die so erstellte Datei wird von der Komponente RST264-RV eingelesen. Hierbei werden nach und nach die IP-Pakete verarbeitet. Diese enthalten als Nutzlast UDP-Pakete, welche als Nutzlast wiederum RTP-Pakete enthalten. Um aus den RTP-Paketen einzelne NALUs extrahieren zu können, werden die RTP-Pakete analysiert und deren Nutzlast gemäß dem in Abschnitt 3.3.3 beschriebenen Nutzlastformat interpretiert.

3.3.3 RTP Payload Format nach RFC 6184

Im Folgenden werden die für diese Arbeit relevanten Inhalte aus RFC 6184 [20] dargelegt. In diesem ist spezifiziert, in welcher Form H.264-codierte Videodaten zwecks der Übertragung via Internet in RTP-Pakete gekapselt werden. Er stellt somit die Grundlage für das in der Komponente RST264-RV umgesetzte Verfahren zum Einlesen von Videodaten aus einem Videostrom dar.

3.3.3.1 Nutzung der RTP-Headerfelder

Zu Beginn des RFCs wird die Verwendung der RTP-Headerfelder behandelt. Hierbei werden die in Abschnitt 2.2.2 bereits erläuterten Felder, wie dort beschrieben, eingesetzt. Darüber hinaus wird aber z.B. im besagten RFC in Abschnitt 5.1 spezifiziert, dass das 1 Bit große M Feld des RTP-Headers dafür verwendet werden soll, das letzte Paket einer AU zu markieren. Auf den ersten Blick könnte dies also genutzt werden, um die Grenzen von AUs zu finden und so einen aufwendigen Parsingprozess zu vermeiden. Jedoch wird ausdrücklich darauf hingewiesen, dass sich der Empfänger bzw. Decodierer nicht auf dieses Feld verlassen darf. Somit kann auf ein aufwendiges Verfahren zum Finden von Framegrenzen nicht verzichtet werden.

3.3.3.2 Nutzlasttypen

Die Nutzlast der RTP-Pakete beginnt immer mit einem 1 Byte großen Feld, welches identisch zum NALU-Header aufgebaut ist. Ebenso wie bei herkömmlichen NALUs mit einem $nal\_unit\_type \in \{1, \ldots, 23\}$ wird hier die Art der Nutzlast durch den *nal_unit_type* beschrieben und im Zuge dessen der Wertebereich erweitert.

Es werden in dem RFC [20] drei grundlegende Typen vorgestellt, um die Nutzlast, in diesem Fall H.264-codierte Videos, zu strukturieren. Diese sind wiederum unterteilt, so dass sich insgesamt sieben Möglichkeiten ergeben, in welcher Form die RTP-Nutzlast strukturiert sein kann. Eine Übersicht über diese, sowie der jeweils dafür festgelegte numerische *nal_unit_type* sind Abbildung 10 zu entnehmen.

Welche dieser sieben Typen gleichzeitig eingesetzt werden, ist vom verwendeten Übertragungsmodus abhängig. Vor der Betrachtung der verschiedenen Übertragungsmodi sollen die aufgelisteten Nutzlasttypen erläutert werden.

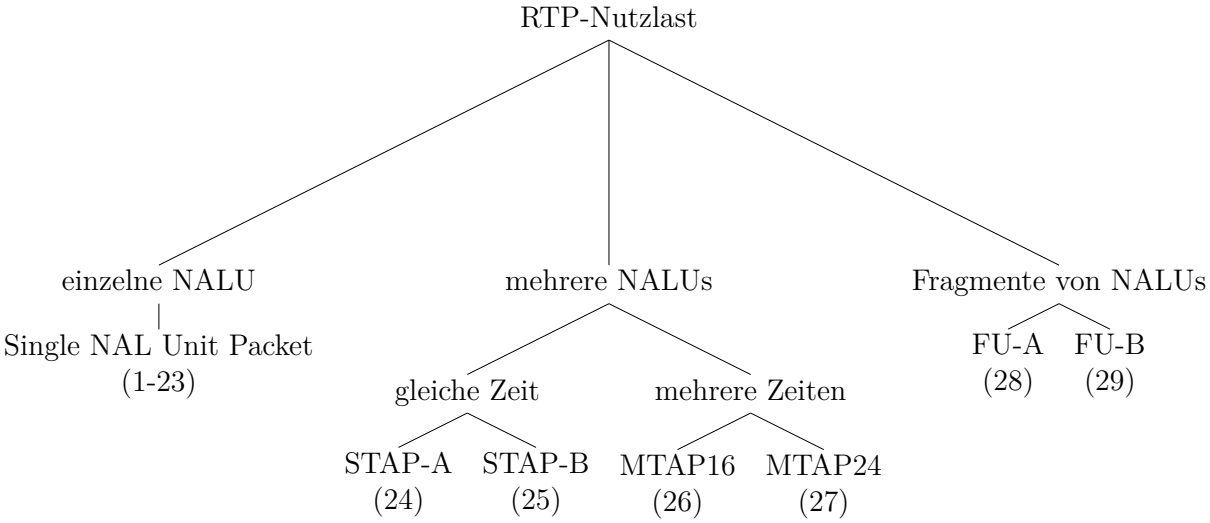

Abbildung 10: Übersicht über verschiedene RTP-Nutzlasttypen

3.3.3.3 Single NAL Unit Packet

Ein RTP-Paket, welches H.264-codierte Videodaten enthält, die nach diesem Typ strukturiert sind, hat folgenden Aufbau (der RTP-Header ist in dieser und auch in folgenden Abbildungen nur angedeutet, weshalb dessen Größe jeweils nicht der realen entspricht):

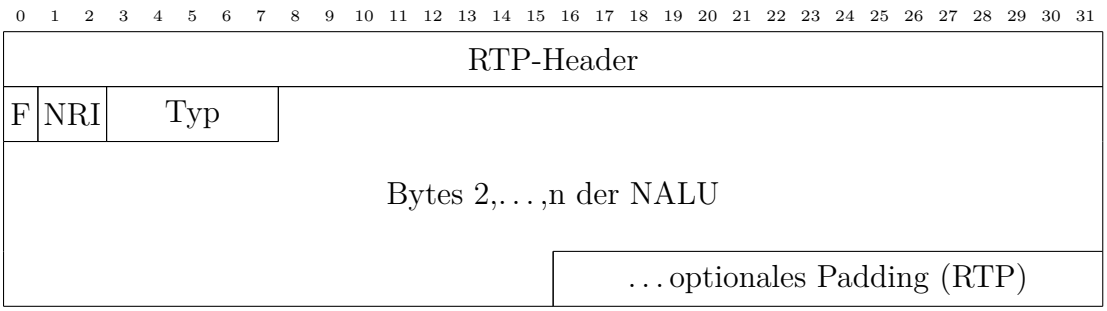

Abbildung 11: Beispiel für Single NAL Unit Packet

Wie Abbildung 11 zu entnehmen ist, enthält das RTP-Paket in diesem Fall als Nutzlast genau eine NALU. Die Nutzlast des RTP-Pakets kann also unverändert an den Decodierer weitergeleitet werden. Der Bereich des Feldes Typ, welcher den Wert von *nal_unit_type* darstellt, liegt bei 1-23. Theoretisch können alle NALUs auf diese einfache Art übertragen werden. In diesem Fall muss allerdings die Übertragungsreihenfolge mit der Decodierungsreihenfolge übereinstimmen.

3.3.3.4 Aggregation Packets

Zur Übertragung mehrerer kleiner NALUs ist es also theoretisch möglich, diese jeweils einzeln zu verpacken. Dieses ist aber mit einem unnötig großen Overhead verbunden, da für jede NALU eigene RTP-, UDP- und IP-Header generiert und mit übertragen werden müssten. Um dies zu vermeiden, werden in Abschnitt 5.7 des RFCs [20] mehrere *Aggregation Packets* definiert, welche das Verpacken mehrerer NALUs in ein RTP-Paket ermöglichen.

Beim Übertragen von NALUs via RTP kann die Decodierungsreihenfolge der NALUs mit der der RTP-Pakete übereinstimmen, sie kann aber auch abweichen. Geht die Decodierungsreihenfolge nicht aus der Reihenfolge der RTP-Pakete hervor, so muss diese beim Empfänger dennoch wieder herstellbar sein. Hierzu wird eine *Decoding Order Number* (DON) eingesetzt.

Es gibt insgesamt vier Varianten von *Aggregation Packets*.

3.3.3.4 (a) Single-time Aggregation Packet

Ein *Single-time Aggregation Packet* (STAP) enthält nur NALUs, welche sich alle auf den gleichen Zeitstempel beziehen. Es wird unterschieden zwischen STAP vom Typ A (STAP-A) und STAP vom Typ B (STAP-B). Bei einem STAP-A ist keine DON vorhanden, bei STAP-B hingegen schon.

STAP-As sind im RFC [20] in Abschnitt 5.7.1 definiert. Deren Struktur ist beispielhaft in Abbildung 12 dargestellt, wobei zwei NALUs enthalten sind.

Die Nutzlast beginnt mit dem STAP-A Header. Dieser enthält die gleichen Felder wie der Header einer herkömmliche NALU. Der *nal_unit_type* ist hierbei immer 24. Neben diesem Header ist zu jeder NALU deren Größe angegeben.

Ein STAP-B unterscheidet sich gegenüber einem STAP-A nur in zwei Punkten. Zum einen ist der *nal_unit_type* nicht 24 sondern 25. Zum anderen befindet sich zwischen dem STAP-Header und der Größenangabe der ersten NALU ein 16 Bit großes Feld, in der die DON angegeben wird. Ein Beispiel hierfür ist ebenfalls in Abschnitt 5.7.1 des RFCs [20] zu finden. Die DONs der einzelnen NALUs ergeben sich folgendermaßen: Der im DON Feld angegebene Wert entspricht der DON der ersten enthaltenen NALU. Die der folgenden ergibt sich durch: (DON der vorherigen NALU + 1) % 65536 (%: Modulo-Operation).

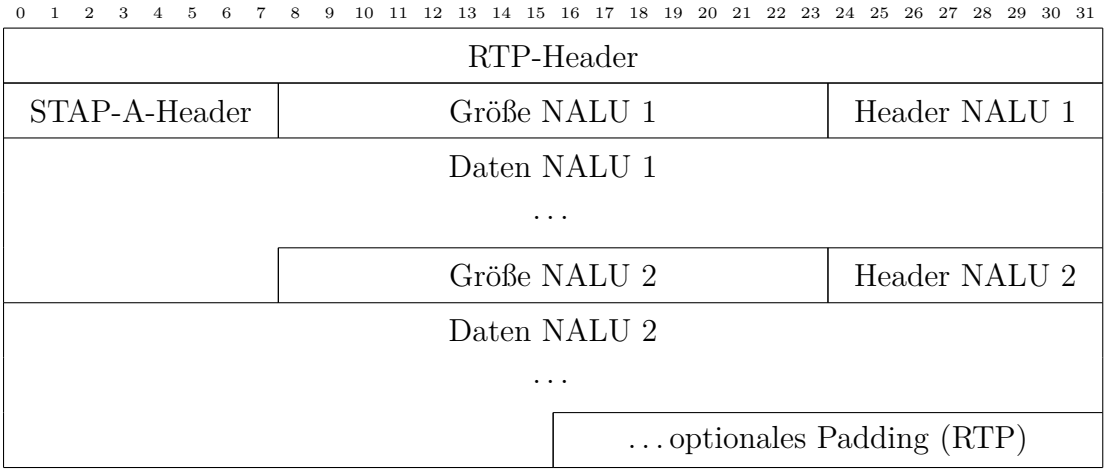

Abbildung 12: Beispiel für STAP-A mit zwei NALUs

3.3.3.4 (b) Multi-time Aggregation Packet

Ein *Multi-time Aggregation Packet* (MTAP) enthält NALUs, die nicht alle den gleichen Zeitstempel haben. Deshalb enthält dieser Typ Zeitoffsets. Je nachdem wie groß die zeitliche Differenz der NALUs ist, wird ein MTAP mit 16 Bit Zeitoffset (MTAP16) oder ein MTAP mit 24 Bit Zeitoffset (MTAP24) genutzt. Bei MTAP16 und MTAP24 kommen jeweils DONs zum Einsatz. Hier muss die Reihenfolge der NALUs innerhalb der Pakete also nicht der Decodierungsreihenfolge entsprechen.

MTAPs sind im RFC [20] in Abschnitt 5.7.2 definiert. Die Struktur einer MTAP16, welche zwei NALUs enthält, ist in Abbildung 13 dargestellt.

Auch hier besteht der MTAP16 Header aus genau den Feldern einer herkömmlichen NALU. Der *nal_unit_type* ist in diesem Fall 26. Neben dem Header ist eine DON *Base* (DONB) angegeben. Darüber hinaus ist für jede NALU die Größe, ein DON *Difference* (DOND) und ein Offset bezüglich des Zeitstempels angegeben. Die DON einer einzelnen NALU ergibt sich durch: (DONB + DOND) % 65536.

Ein MTAP24 unterscheidet sich gegenüber einem MTAP16 nur durch die Länge des Feldes für die Offsets der Zeitstempel. So können durch eine MATP24 größere Zeitspannen abgedeckt werden. Gleichzeitig wird aber auch der Overhead größer.

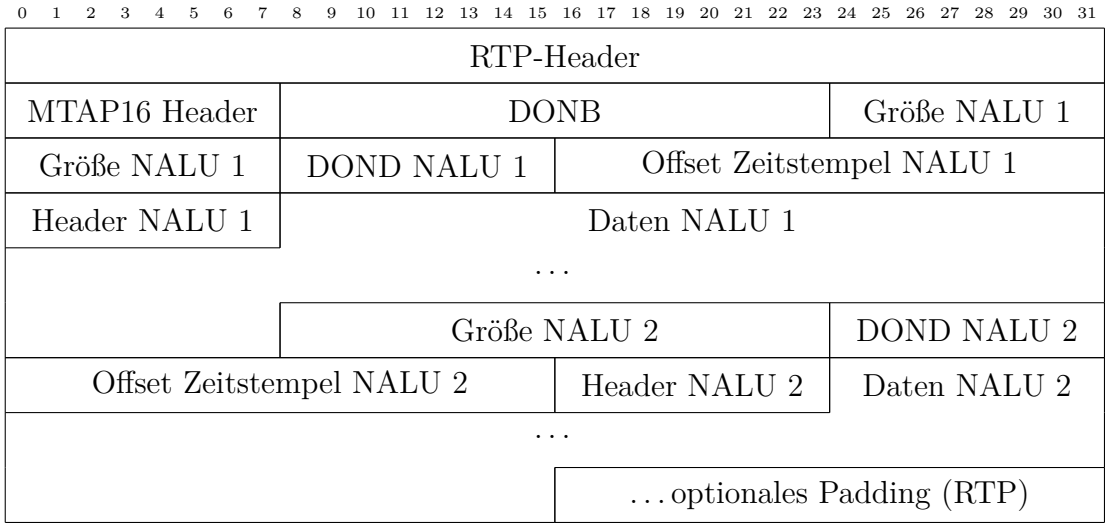

Abbildung 13: Beispiel für MTAP16 mit zwei NALUs

3.3.3.5 Fragmentation Units

Ist eine NALU sehr groß, so muss sie auf mehrere RTP-Pakete aufgeteilt werden. In diesem Fall kommen *Fragmentation Units* (FUs) zum Einsatz. Sie sind in Abschnitt 5.8 des RFCs [20] definiert. Es können FUs vom Typ A (FU-A) oder FUs vom Typ B (FU-B) verwendet werden. Die Struktur einer FU-A ist in Abbildung 14 dargestellt.

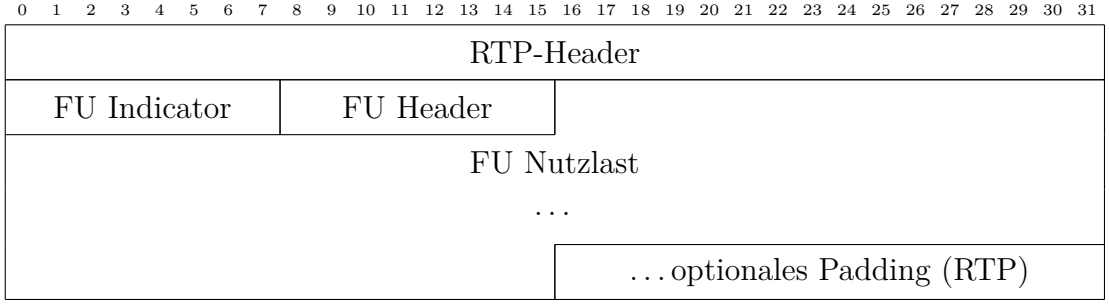

Abbildung 14: Beispiel für eine FU-A

Der FU *Indicator* hat folgenden Aufbau:

- **F** (1 Bit):
  Dieser wird wie gewohnt gesetzt und gibt an, ob Fehler vorliegen.
- **NRI** (2 Bit):
  Die Semantik dieses Wertes für eine NALU aus dem Bereich 1-23 ist bereits bekannt. Hier ist sie identisch. Der Wert dieses Feldes entspricht dem Wert der fragmentierten NALU.

- **Type** (5 Bit):
  Dieses Feld hat den Wert 28 (FU-A) oder 29 (FU-B).

Der FU Header hat folgenden Aufbau:

- **S** (1 Bit):
  Ist dieser Wert 1, so ist diese FU der Beginn einer neuen NALU. Andernfalls ist der Wert immer 0.
- **E** (1 Bit):
  Ist dieser Wert 1, so ist diese FU die letzte, die zu der aktuellen NALU gehört. Andernfalls ist der Wert immer 0.
- **R** (1 Bit):
  Dieses Bit ist immer 0 und wird vom Empfänger ignoriert.
- **Type** (5 Bit):
  Dieses Feld beschreibt den NALU Typ der fragmentierten NALU.

Die Nutzlast einer FU ist immer ein Teil einer NALU.

Eine FU-B enthält im Gegensatz zu einer FU-A ein 16 Bit großes Feld für die Angabe einer DON. Dieses befindet sich unmittelbar vor der Nutzlast.

3.3.3.6 Modi für die Paketierung

Welche der oben beschriebenen Typen von Paketen wann und in welcher Kombination genutzt werden, wird durch den jeweils verwendeten Modus für die Übertragung bestimmt.

Der RFC [20] definiert in Abschnitt 6 drei verschiedene Modi:

- **Single NAL Unit Mode**:
  In diesem Modus dürfen nur *Single NAL Unit Packets* verwendet werden. Hierbei muss die Übertragungsreihenfolge identisch zur Decodierungsreihenfolge sein. Er wird eingesetzt für Videoübertragung mit wenig Verzögerung. Jeder Empfänger muss wenigstens diesen Modus unterstützen.
- **Non-Interleaved Mode**:
  Dieser Modus ist ebenfalls für die Übertragung von Videos mit wenig Verzögerung gedacht. Es dürfen neben *Single NAL Unit Packets* auch STAP-As und FU-As verwendet werden. Da keine dieser Typen DONs enthalten, muss auch hier die Übertragungsreihenfolge identisch zur Decodierungsreihenfolge sein. Dieser Modus sollte von jedem Empfänger unterstützt werden.
- **Interleaved Mode**:
  In diesem Modus werden nur STAP-Bs, MTAPs, FU-As und FU-Bs verwendet. Da in diesem Modus mehrere NALUs auch mit verschiedenen Zeitstempeln gemeinsam übertragen werden, ist er für die Übertragung von Echtzeitvideos nicht einsetzbar. Empfänger können diesen Modus unterstützen.

In dieser Arbeit wird hauptsächlich der *Non-Interleaved Mode* eingesetzt. Der *Interleaved Mode* ist nicht anwendbar, da er nicht für die Echtzeitübertragung geeignet ist. Beim *Single NAL Unit Mode* kann es bei großen NALUs dazu kommen, dass diese größer sind als die *Maximum Transmission Unit* (MTU) und so eine Fragmentierung dieser unumgänglich ist.

3.3.4 Begründung der Vorgehensweise

Auf den ersten Blick wirkt das beschriebene Verfahren zum Einlesen eines Videos eventuell sehr umständlich. Vor allem auch, weil ein umfangreicher RFC herangezogen werden muss. Es wäre beispielsweise auch möglich, direkt ein gespeichertes Video durch die Komponente RST264-RV einzulesen. Bei diesem Ansatz ergeben sich allerdings folgende Schwierigkeiten:

Videodateien sind je nach verwendetem Videocontainerformat unterschiedlich aufgebaut. Zudem enthalten sie neben den Videodaten auch Audiodaten, welche nicht von der Komponente RST264-RV berücksichtigt werden sollen. Soll das Einlesen eines Videos also durch direktes Einlesen einer Videodatei erfolgen, so müsste die Komponente in der Lage sein, jedes gängige Videocontainerformat zu interpretieren. Darüber hinaus müsste auch ein Einlesen der Videos nach dem sog. Bytestreamformat erfolgen.

Wird das Video hingegen zunächst mittels *ffmpeg* gestreamt und dieser Videostrom dann eingelesen, ergeben sich folgende Vorteile:

ffmpeg ist sehr mächtig. Daher können verschiedenste Videocontainerformate und Videocodierungen verarbeitet und gestreamt werden. Ebenfalls ist es möglich, die Audiodaten eines Videos bewusst nicht mit zu übertragen.

Beim Streamen eines Videos mittels des bereits beschriebenen *ffmpeg*-Befehls wird immer der *Non-Interleaved Mode* (beschrieben in Abschnitt 3.3.3.6) verwendet. Somit ist das Einlesen eines Videos über den indirekten Weg eines Videostroms immer auf die gleiche Weise möglich. Darüber hinaus liegen die Videodaten durch die für das Streaming vorgenommene Paketierung bereits in einer sinnvollen Unterteilung vor. Um einzelne NALUs zu erhalten, wie sie in Abschnitt 2.1 vorgestellt wurden, ist es zwar notwendig, die Daten gemäß [20] zu interpretieren. Doch dieser RFC muss ohnehin für das spätere Streaming des Videos herangezogen werden.

3.3.5 Beispiel für den Einsatz der Komponente RST264-RV

Das folgenden Codebeispiel zeigt, wie eine pcap-Datei durch die Komponente RST264-RV eingelesen werden kann. Weiterhin zeigt es, wie die Komponente RST264-RV genutzt wird, um die eingelesenen Daten als NALUs zu erhalten.

Hierzu wird in Zeile 4 eine Instanz der Klasse *Pcap_nalu_parser* erstellt und als Parameter die gewünschte pcap-Datei angegeben. In Zeile 5 wird die Methode *parse_nalus()* des

Objekts aufgerufen, welche eine Liste der in dem aufgezeichneten Videostrom enthaltenen NALUs zurückliefert.

```
1 import ...
2
3 default_pcap_filepath = 'default_stream.pcap'
4 parser = Pcap_nalu_parser(default_pcap_filepath)
5 nalu_list = parser.parse_nalus()
```

Das zweite Beispiel zeigt die volle Schnittstelle des Konstruktors von *Pcap_nalu_parser*. Neben dem Pflichtparameter, welcher den Dateipfad zur pcap-Datei darstellt, können zusätzlich folgende Angaben gemacht werden:

- *outoutput_mode* (standardmäßig *False*):
  Wird dieser auf *True* gesetzt, so werden die aus der pcap-Datei geparsten Typangaben der NALUs in eine txt-Datei geschrieben und können so nachvollzogen werden.
- *port* (standardmäßig 554):
  Über diesen Parameter kann angegeben werden, welcher Port beim Streamen des Videos gewählt wurde. Nur Pakete mit dieser Portangabe werden von der Komponente RST264-RV beim Parsingprozess berücksichtigt.

```
1 parser = Pcap_nalu_parser(default_pcap_filepath,output_mode=True,
      port=1234)
```

3.4 Komponente zum Erkennen von Framegrenzen (RST264-FD)

Wie bereits in Abschnitt 2.1 erwähnt, führt die Decodierung je einer AU zu einem decodierten Bild. Im weiteren Verlauf wird daher das Finden von Framegrenzen mit dem Finden von Anfängen von AUs gleichgesetzt.

In diesem Abschnitt wird zunächst der Aufbau einer AU beschrieben und ein Überblick über die einzelnen Bestandteile und deren Bedeutung gegeben. Anschließend wird erläutert, anhand welcher Kriterien Framegrenzen zu erkennen sind und wie die zum Prüfen dieser Kriterien benötigten Syntaxelemente aus vorliegenden Videodaten extrahiert werden können. Abschließend wird dargestellt, wie dieses Vorgehen softwareseitig durch die Komponente RST264-FD implementiert wird.

3.4.1 Aufbau einer Access Unit

Nach Definition 3.1 des H.264-Standards [19] besteht eine AU aus einer Menge von NALUs, die zusammen folgende Komponenten bilden: Genau ein *Primary Coded Picture* (PCP), n-viele *Redundant Coded Pictures* (RCPs), höchstens ein *Auxiliary Coded Picture* (ACP). Weiterhin können n-viele non-VCL NALUs enthalten sein. Hierbei ist n Element der natürlichen Zahlen inklusive 0.

Da die Begriffe PCP, RCP und ACP bisher noch nicht erläutert wurden, erfolgt dies nach Wiedergabe der entsprechenden Definitionen des Standards [19]:
Ein PCP enthält alle codierten Daten eines ganzen Bildes, dieses enthaltene Bild ist das einzige, das maßgeblichen Einfluss auf den Decodierungsprozess hat.
Ein RCP kann Teile eines Bildes oder ein ganzes Bild enthalten. Es sollte nicht bei dem Decodierungsprozess einbezogen werden und trägt auch nicht maßgeblich dazu bei.
Ein ACP ist ein Bild, welches das PCP ergänzen und in Verbindung mit zusätzlichen Daten beim Darstellungsprozess mit einbezogen werden kann. Auch ein ACP hat keinen maßgeblichen Einfluss auf den Decodierungsprozess.

Die Reihenfolge der NALUs innerhalb einer AU und deren Aufbau wird in [19] Abschnitt 7.4.1.2.3 beschrieben und ist dort in einer vereinfachten Form dargestellt. Abbildung 15, die der aus dem genannten Abschnitt entspricht, zeigt den Aufbau einer AU ohne Berücksichtigung der NALUs mit $nal\_unit\_type \in \{0, 7, 8, 12, \ldots, 18, 20, \ldots, 31\}$.

Der Abbildung ist folgendes zu entnehmen:
Ist eine NALU vom Typ *Access unit delimiter* vorhanden, so stellt sie den Start der AU dar. Eine solche NALU muss aber nicht Teil einer AU sein. Sind NALUs vorhanden, die SEI enthalten, so sollen diese in jedem Fall vor dem PCP auftauchen. Allerdings müssen auch solche NALUs nicht zwangsläufig Teil einer AU sein. Als nächstes in der Reihenfolge kommt das PCP. Dies ist auf jeden Fall enthalten und taucht immer vor NALUs auf, die zum RCP gehören. Diese wiederum kommen vor NALUs, welche zum ACP gehören. In einer AU müssen weder ein RCP noch ein ACP enthalten sein. Zum Ende einer AU können noch die NALUs *End of sequence* oder *End of stream* enthalten sein.

Es sei an dieser Stelle nochmals ausdrücklich darauf hingewiesen, dass die Darstellung nicht alle Typen an NALUs enthält. Insbesondere wird z.B. weder durch die Abbildung, noch durch weitere textuelle Beschreibung die genaue Position eines PPS oder SPS definiert. Es wird im Standard [19] lediglich festgelegt, dass sie in jedem Fall in der Reihenfolge vor der letzten NALU des PCP liegen müssen.

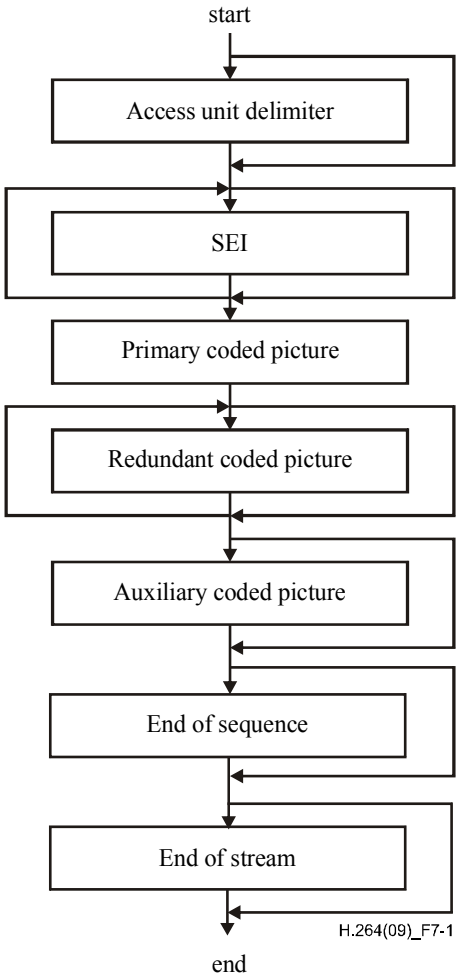

Abbildung 15: Aufbau einer Access Unit

3.4.2 Grundlegendes Vorgehen zum Erkennen von Framegrenzen

Ein konkretes Vorgehen zum Finden von Framegrenzen oder zum Finden von Grenzen von AUs wird in [19] nicht beschrieben. Das im Rahmen dieser Arbeit entwickelte Vorgehen soll im Folgenden vorgestellt werden. Es basiert auf Einschränkungen bezüglich der Reihenfolge von NALUs innerhalb einer AU und auf Merkmalen zum Finden von Anfang und Ende des PCP innerhalb einer AU.

Vorgehen

Als erstes werden innerhalb der NALU-Sequenz alle NALUs gekennzeichnet, welche den Beginn eines PCP darstellen. Als zweites wird jeweils die letzte NALU eines PCP markiert (in anderer Form als der Beginn des PCP). Hiernach können zunächst alle NALUs, welche jeweils zwischen den Grenzen eines PCP liegen, derselben AU zugeordnet werden.

In manchen Fällen liegen zwischen den PCPs weitere NALUs. Um in diesem Fall den Anfang bzw. die Grenzen von AUs zu bestimmen, wird Abschnitt 7.4.1.2.3 des Standards [19] angewendet, welcher unter anderem besagt: Tritt eine der aufgelisteten NALUs nach der letzten VCL NALU des PCP auf, so stellt diese den Start einer neuen AU dar. (Bei der Aufzählung ist der Übersicht halber immer der numerische Wert des jeweiligen *nal_unit_type* vorangestellt.)

- (9) *Access unit delimiter* NALU (wenn vorhanden)
- (7) SPS NALU (wenn vorhanden)
- (8) PPS NALU (wenn vorhanden)
- (6) *SEI* NALU (wenn vorhanden)
- NALUs für die gilt: $nal\_unit\_type \in \{14, \ldots, 18\}$ (wenn vorhanden)
- die erste VCL NALU eines neuen PCP

Die letzte NALU dieser Aufzählung ist immer präsent. Spätestens bei der Überprüfung dieser Bedingung wird also der Beginn einer neuen AU festgestellt.

Hinweise zum Vorgehen

Es sei erwähnt, dass NALUs, die erst durch den letzten Schritt zugeordnet werden können, in den Videosequenzen, die im Rahmen dieser Arbeit betrachtet wurden, so gut wie immer PPSs oder SPSs waren. Ist ein PPS oder ein SPS in einer AU enthalten, heißt dies nicht, dass sich zwangsläufig schon die NALUs des PCP, welche ebenfalls in derselben AU enthalten sind, auf diese PPS bzw. SPS beziehen.

Es sei ebenfalls darauf hingewiesen, dass dieses und das weiterhin beschriebene Vorgehen zum Finden der ersten bzw. letzten VCL NALU nur zuverlässig funktioniert, wenn sich alle CVSs des betrachteten Videos nach Annex A des Standards richten und durch einen Decodierer decodiert werden können, der sich nach den Abschnitten 2 bis 9 des Standards [19] richtet. Dies ist jedoch in allen im Rahmen dieser Arbeit betrachteten Fällen gegeben. Abweichungen ergeben sich lediglich in Annex G über SVC und Annex H über MVC.

Beispiel

Das allgemeine Vorgehen soll anhand eines Beispiels erläutert werden, welches in der Praxis vermutlich nie so auftreten würde, aber gut zur Verdeutlichung geeignet ist. Abbildung 16 zeigt eine Sequenz von NALUs, wobei der jeweilige Index der NALU und in Klammern der numerische NALU Typ angegeben ist. Die Anfänge bzw. Enden von PCPs seien bereits gegeben. Die **rot** gefärbten NALUs stellen den Anfang eines PCP dar, die **gelben** das Ende und die **orangenen** NALUs sind sowohl Anfang als auch Ende eines PCP.

Die Grenzen der AUs ergeben sich auf folgende Weise:

Dass AU 1 nur aus NALU 1 besteht, ist leicht zu sehen. Zum einen stellt sie sowohl den Beginn, als auch das Ende eines PCP dar. Zudem folgt direkt danach mit NALU

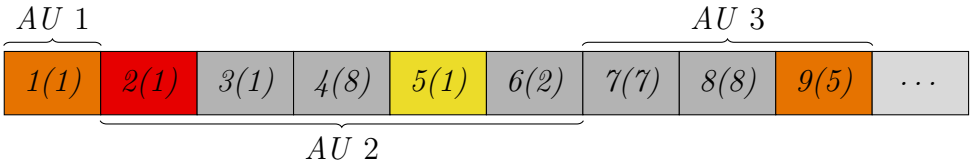

Abbildung 16: Sequenz von NALUs

2 der Anfang eines neuen PCP. NALU 2 kann also in keinem Fall zu AU 1 gehören. Somit gehört diese NALU zu AU 2. In jedem Fall auch zu AU 2 gehört das am nächsten liegende Ende eines PCP, in diesem Fall NALU 5. Es folgt sofort, dass alle NALUs, die zwischen diesen beiden liegen, ebenfalls zu AU 2 gehören. Dass NALU 9 zu AU 3 gehört, ist wieder leicht erkennbar, da sie den Beginn und auch das Ende eines PCP darstellt. Es bleibt noch zu klären, welche der NALUs (6, 7 oder 8) den Anfang einer neuen AU darstellt. Hierzu wird die oben beschriebene Auflistung herangezogen. Wir gehen also von der letzten NALU des PCP von AU 2 aus und betrachten die nächste NALU, also NALU 6. Dessen *nal_unit_type* ist weder 9, 8, 7, 6 oder aus dem Bereich 14-18, noch ist sie die erste NALU eines neuen PCP. Somit stellt sie nicht den Beginn einer neuen AU dar und gehört noch zu AU 2. NALU 7 hat jedoch einen *nal_unit_type* der entweder 9, 8, 7, 6 oder aus dem Bereich 14-18 ist. Somit stellt diese den Beginn einer neuen AU dar und ist AU 3 zuzuordnen. Da eine AU mindestens ein PCP enthalten muss, werden in diesem Fall mindestens alle NALUs ab dem Beginn einer neuen AU bis zum nächsten Beginn eines PCP der gleichen AU zugeordnet. In diesem Fall wird also NALU 8 ebenfalls AU 3 zugeordnet.

3.4.2.1 Finden der ersten NALU eines Primary Coded Pictures

Wie bereits beschrieben, ist das Finden der ersten NALU eines PCP ein notwendiger Zwischenschritt zum Identifizieren von Grenzen von AUs. Im Folgenden wird erläutert, wie hierzu vorgegangen wird.

In [19] in Abschnitt 7.4.1.2.4 wird beschrieben, dass sich alle NALUs mit *nal_unit_type* $\in \{1, 2, 5\}$ des PCP der aktuellen AU in mindestens einem der folgenden Punkte zu allen NALUs mit *nal_unit_type* $\in \{1, 2, 5\}$ des PCP der vorherigen AU unterscheiden. Die verwendeten Farben stellen hierbei dar, welchem Syntaxelement die einzelnen Werte zu entnehmen sind. Sie wurden der Übersichtlichkeit halber verwendet und sind nicht im Standard wiederzufinden. Die *blau* markierten Syntaxelemente sind in den VCL NALUs selbst enthalten. Die *grün* markierten sind den jeweiligen SPSs zu entnehmen.

- Die *frame_num* der NALUs unterscheidet sich.
- Die *pic_parameter_set_id* der NALUs unterscheidet sich.
- Die *field_pic_flag* der NALUs unterscheidet sich.
- Die *bottom_field_flag* ist in beiden NALUs vorhanden und unterscheidet sich.

- Der Wert für *nal_ref_idc* unterscheidet sich, wobei mindestens einer der Werte 0 ist.
- Der *pic_order_cnt_type* ist für beide NALUs 0 und entweder unterscheiden sich die Werte *pic_order_cnt_lsb*, oder die Werte von *delta_pic_order_cnt_bottom*.
- Der *pic_order_cnt_type* ist gleich 1 für beide NALUs und entweder unterscheiden sich die Werte im Array *delta_pic_order_cnt* an Index 0 oder 1.
- Der Wert der *IdrPicFlag* unterscheidet sich.
- Der Wert der *IdrPicFlag* ist 1 für beide NALUs und die *idr_pic_id* unterscheidet sich.

In dem Standard wird zu Beginn des Abschnittes erwähnt: „This clause specifies constraints on VCL NAL unit syntax that are sufficient to enable the detection of the first VCL NAL unit of each primary coded picture [...]“. Doch viel mehr als die oben genannten Punkte werden nicht erwähnt.

Dies ist aus folgenden Gründen kritisch:

- Genau genommen wird nur die Aussage getroffen: Liegt eine Grenze zu einem neuen PCP vor, so folgt, dass mindestens einer der oben aufgeführten Punkte zutrifft. Über die Implikation in die andere Richtung wird keine Aussage getroffen. Dass die Implikation auch in diese Richtung gilt, wäre wünschenswert, denn nur durch diese Rückrichtung lässt sich zum Erkennen der ersten VCL NALU des PCP einer AU so vorgehen, dass zunächst überprüft wird, ob eine der Bedingungen zutrifft und in Folge dessen auf den Beginn eines neuen PCP geschlossen werden kann.
- In der obigen Auflistung werden nur NALUs mit $nal\_unit\_type \in \{1, 2, 5\}$ einbezogen. Genau genommen wird durch Anwendung dieser Regel also nur die erste VCL NALU einer AU gefunden, die $nal\_unit\_type \in \{1, 2, 5\}$ hat. Es ist also theoretisch nicht ausgeschlossen, dass die tatsächlich erste VCL NALU der AU noch davor liegt. In diesem Fall wäre der *nal_unit_type* der ersten NALU des PCP 3 oder 4.

Durch das Behandeln weiterer Inhalte des Standards [19] soll zunächst die Rückrichtung der obigen Implikation gezeigt werden. Es ist zwar durch das angeführte Zitat sehr naheliegend, dass diese gilt, aber da das gesamte hier vorgestellte Vorgehen zum Finden von Framegrenzen auf dieser Annahme beruht, soll deren Richtigkeit sichergestellt werden. Weiterhin soll beschrieben werden, wie mit dem im zweiten Punkt erläuterten Spezialfall umzugehen ist.

Zeigen der Rückrichtung:

Es zu zeigen: Trifft für zwei NALUs mit $nal\_unit\_type \in \{1, 2, 5\}$ mindestens einer der oben genannten Punkte zu, so gehören diese nicht zum gleichen PCP einer AU und gehören somit unterschiedlichen AUs an.

Hierzu sollen zunächst wichtige Aspekte des Standards und Erkenntnisse daraus aufgeführt werden, die zu späterem Zeitpunkt genutzt werden:

1. Wenn die folgenden Felder gesetzt sind, dann haben sie für alle NALUs eines PCP die gleichen Werte: *frame_num*, *pic_parameter_set_id*, *field_pic_flag*, *bottom_field_flag*, *pic_order_cnt_lsb*, *delta_pic_order_cnt_bottom*, *delta_pic_order_cnt[0]*, *delta_pic_order_cnt[1]*, *idr_pic_id*, *IdrPicFlag* (Dies ist eine Schlussfolgerung aus Absatz 7.4.3 des Standards [19], in dem die Semantik des *Slice Header* definiert wird.)
2. Da das Feld *pic_order_cnt_type* im SPS enthalten ist, ist es in jedem Fall für alle AUs eines PCP gleich.
3. Ist der Wert von *nal_ref_idc* einer VCL NALU eines PCP einer AU gleich 0, so ist der Wert von *nal_ref_idc* aller anderen VCL NALUs derselben AU ebenfalls 0, (sinngemäß aus S.89 Absatz 4 in [19] entnommen).

Um obige Aussage zu zeigen, wird diese in eine äquivalente umformuliert:

Gehören zwei NALUs dem gleichen PCP und somit der gleichen AU an, so trifft keiner der oben genannten Punkte zu.

Werden die aufgelisteten Punkte betrachtet, so stellt man fest, dass fast alle Punkte allein aufgrund der Tatsache, dass die dort aufgeführten Syntaxelemente nach 1. und 2. gleich sein müssen, nicht zutreffen können.

Der einzige Punkt, über den bisher keine Aussage getroffen werden kann, besagt: Der Wert für *nal_ref_idc* unterscheidet sich, wobei mindestens einer der Werte 0 ist.

Dass auch dieser Punkt nicht zutrifft, wird im Folgenden erläutert:

Angenommen zwei VCL NALUs X und Y gehören dem gleichen PCP und somit der gleichen AU an.
Damit der obige Punkt zutreffen kann, muss in jedem Fall gelten, dass *nal_ref_idc* für eine der NALUs 0 ist. Sei also o.B.d.A. *nal_ref_idc* von X 0. Damit der Punkt zutrifft, müssen sich zudem die Werte bezüglich *nal_ref_idc* von X und Y unterscheiden. Aufgrund von 3. sind aber beide Werte 0 und unterscheiden sich somit nicht. Auch diese Bedingung trifft also nicht zu und somit keine der genannten.

Es folgt also die umgeformte Aussage und somit auch die Rückrichtung der Implikation.

Umgang mit dem Spezialfall

Um den oben dargelegten Spezialfall zu behandeln, werden weitere Inhalte des Standards [19] herangezogen.

In Abschnitt 7.4.1.2.5 wird die Reihenfolge von VCL NALUs in codierten Bildern definiert, also auch in PCPs. Hier wird erwähnt, dass NALUs mit *nal_unit_type* $\in \{3, 4\}$ immer nach NALUs mit *nal_unit_type* $\in \{1, 2\}$ auftreten, solange sie die gleiche *slice_id* aufweisen.

Der oben beschriebene Sonderfall, dass eine VCL NALU mit *nal_unit_type* $\in \{3, 4\}$ den Anfang des PCP darstellt, kann also nur eintreten, wenn sich deren *slice_id* von den *slice_ids* der VCL NALUs mit *nal_unit_type* $\in \{1, 2\}$ unterscheidet. Der Fall *nal_unit_type* =

5 muss hierbei nicht beachtet werden, da in einer IDR AU keine NALUs mit *nal_unit_type* $\in \{3, 4\}$ enthalten sind.

Dieser Sachverhalt lässt sich sogar noch weiter einschränken. Unter Einbezug der Beschreibung der Semantik der *slice_id* (siehe Abschnitt 7.4.2.9.1 in [19]) ist diese so definiert, dass für Annex A in jedem Fall gelten muss: Die *slice_id* der ersten VCL NALU des PCP muss 0 sein. Danach wird sie jeweils inkrementiert.

Das heißt: Der genannte Spezialfall kann nur eintreten, wenn die *slice_id* der ersten NALU mit *nal_unit_type* $\in \{1, 2\}$ ungleich 0 ist. Ist dies der Fall, so stellt eine NALU mit *nal_unit_type* $\in \{3, 4\}$ und *slice_id* $= 0$ den Beginn des PCP dar und kann anhand dieses Kriteriums gefunden werden.

Zusammenfassend lässt sich also folgendes Verfahren zum Finden der ersten VCL NALU einer AU anwenden: Zunächst wird mittels der oben beschriebenen Kriterien die erste NALU mit *nal_unit_type* $\in \{1, 2, 5\}$ des PCP gefunden. Treten vor dieser NALU noch weitere VCL NALUs auf, welche einen *nal_unit_type* $\in \{3, 4\}$ aufweisen, dann wird anhand der *slice_id* überprüft, ob diese schon zum PCP gehören oder noch der vorherigen AU zuzuordnen sind.

3.4.2.2 Finden der letzten NALU des Primary Coded Pictures

In diesem Abschnitt wird erläutert, wie die letzte NALU des PCP einer AU gefunden werden kann. Dass neben dem Finden der ersten NALU des PCP auch das Finden der letzten NALU des PCP relevant ist, liegt daran, dass die in Abschnitt 3.4.2 beschriebene Überprüfung der NALUs nach der letzten NALU des PCP startet.

Das entscheidende Syntaxelement in diesem Fall ist *redundant_pic_cnt*. *redundant_pic_cnt* hat nur einen definierten Wert, wenn *redundant_pic_cnt_present_flag* gesetzt ist. Hat es keinen definierten Wert, so ist *redundant_pic_cnt* nach Standard [19] mit 0 zu interpretieren. Ist das Feld *redundant_pic_cnt_present_flag* für eine VCL NALU des PCP gesetzt, so ist es für alle VCL NALUs des PCP gesetzt, da dieses Feld dem PPS zu entnehmen ist und alle NALUs eines PCP dasselbe PPS referenzieren. Weiterhin wird im Standard gesagt, dass das Feld *redundant_pic_cnt* für alle NALUs des PCP 0 und für alle NALUs eines RCP größer als 0 sein soll. Die letzte NALU mit *nal_unit_type* $\in \{1, 2, 3, 4, 5\}$ und *redundant_pic_cnt* $= 0$ vor dem nächsten Beginn eines neuen PCP ist also als letzte NALU des aktuellen PCP zu interpretieren.

3.4.3 Extrahieren der für das Finden von Framegrenzen benötigten Syntaxelemente

Nachdem im vorherigen Abschnitt erläutert wurde, anhand welcher Syntaxelemente es möglich ist, die Grenzen von AUs zu erkennen, wird jetzt beschrieben, wie diese Syntaxelemente zu extrahieren sind.

Zunächst sei angemerkt, dass das Extrahieren bestimmter Syntaxelemente durch folgende Punkte sehr aufwendig ist:

- Bei der Betrachtung aller benötigten Syntaxelemente fällt auf, dass nicht alle Syntaxelemente den jeweils zu vergleichenden NALUs zu entnehmen sind. Es werden auch Syntaxelemente des SPS benötigt, auf welchen indirekt über das PPS referenziert wird. Dies führt dazu, dass bei der Analyse der NALUs immer darauf geachtet werden muss, welche PPSs bzw. SPSs im Moment herangezogen werden müssen.
- Manche Syntaxelemente haben eine variable Länge. Dies führt dazu, dass zum Extrahieren dieser Syntaxelemente bestimmte, im Standard [19] definierte, Funktionen (Descriptoren) angewendet werden müssen. Teilweise benötigen diese aber wiederum andere Syntaxelemente als Parameter. So ist z.B. die Länge des Feldes *frame_num* im *slice header* (Teil bestimmter NALUs) von dem Sytaxelement *get_frame_num_length* abhängig, welches dem SPS zu entnehmen ist.
- Manche Syntaxelemente sind nur in bestimmten Fällen enthalten (z.B.: *colour_plane_flag*). Zusammen mit dem zweiten Punkt hat dies zur Folge, dass es unmöglich ist, nur die momentan benötigten Syntaxelemente zu extrahieren. Es müssen mindestens alle Syntaxelemente bis zum letzten benötigten extrahiert werden.

3.4.3.1 Definition der Syntax von NALUs und darin enthaltene Elemente

Um trotz der oben genannten Punkte die benötigten Syntaxelemente zu extrahieren, wird zunächst die Syntax dieser Elemente betrachtet.

Im Standard [19] wird die Syntax der jeweiligen Elemente in Tabellenform angegeben. Zu jeder NALU ist definiert, welcher Syntaxstruktur sie entspricht. Dies lässt sich in Abbildung 17 sehen. Hier ist ein Ausschnitt der Tabelle enthalten, in der alle NALUs und unter anderem deren zugeordnete Syntax aufgelistet ist.

So ist die Syntax einer NALU vom Typ *Coded slice of a non-IDR picture* beispielsweise durch die Syntaxdefinition mit der Bezeichnung *slice_layer_without_partitioning_rbsp()* definiert. Diese ist in Abbildung 18 dargestellt. Sie enthält keine konkreten Syntaxelemente, verweist aber auf weitere Syntaxdefinitionen, unter anderem auf *slice_data()*, diese wiederum auf *macroblock_layer()*. Ein Ausschnitt dieser Definition ist in Abbildung 19 zu sehen.

Es sei darauf hingewiesen, dass diese Syntaxelemente keine höhere Relevanz als andere haben. Sie wurden nur ausgewählt, da sie sich gut zur beispielhaften Erläuterung eignen.

Bei den einzelnen Tabellen zur Syntaxbeschreibung sind jeweils in der linken Spalte die einzelnen Felder aufgelistet. In der Spalte mit der Bezeichnung **C** wird eine Kategorie angegeben, deren Bedeutung in Abschnitt 3.4.3.1 (a) erläutert wird. In der Spalte mit der Bezeichnung **Descriptor** ist angegeben, wie das entsprechende Feld zu extrahieren ist.

In Abbildung 19 zeigt sich, dass es üblich ist, viele Felder mit variabler Länge zu verwenden. Dies ist an dem Parameter v des jeweiligen Descriptors zu erkennen. Ebenfalls sind hier Beispiele für Felder zu finden, welche nicht immer enthalten sind, was jeweils an den if-Bedingungen zu erkennen ist.

nal_unit_type	Content of NAL unit and RBSP syntax structure	C	Annex A NAL unit type class	Annex G and Annex H NAL unit type class	Annex I and Annex J NAL unit type class
0	Unspecified		non-VCL	non-VCL	non-VCL
1	Coded slice of a non-IDR picture slice_layer_without_partitioning_rbsp( )	2, 3, 4	VCL	VCL	VCL
2	Coded slice data partition A slice_data_partition_a_layer_rbsp( )	2	VCL	not applicable	not applicable
3	Coded slice data partition B slice_data_partition_b_layer_rbsp( )	3	VCL	not applicable	not applicable
4	Coded slice data partition C slice_data_partition_c_layer_rbsp( )	4	VCL	not applicable	not applicable
5	Coded slice of an IDR picture slice_layer_without_partitioning_rbsp( )	2, 3	VCL	VCL	VCL

Abbildung 17: Ausschnitt der Auflistung aller NALUs

slice_layer_without_partitioning_rbsp( ) {	C	Descriptor
slice_header( )	2	
slice_data( ) /* all categories of slice_data( ) syntax */	2 | 3 | 4	
rbsp_slice_trailing_bits( )	2	
}		

Abbildung 18: Ausschnitt Syntaxdefinition slice_layer_without_partitioning_rbsp()

macroblock_layer( ) {	C	Descriptor
mb_type	2	ue(v) | ae(v)
if(mb_type == I_PCM) {		
while(!byte_aligned( ))		
pcm_alignment_zero_bit	3	f(1)
for(i = 0; i < 256; i++)		
pcm_sample_luma[i **]**	3	u(v)
for(i = 0; i < 2 * MbWidthC * MbHeightC; i++)		
pcm_sample_chroma[i **]**	3	u(v)

Abbildung 19: Ausschnitt Syntaxdefinition macroblock_layer

3.4.3.1 (a) Interpretation der Kategorie

Wie bereits erwähnt, wird in der Spalte mit der Bezeichnung **C** jedem Syntaxelement eine Kategorie zugeordnet. Sie wird verwendet, um die Zuordnung von Syntaxelementen zu NALUs herzustellen. Deshalb werden auch in der Tabelle, in der die NALUs aufgelistet sind (vgl. Abbildung 17), jeder NALU eine oder mehrere Kategorien zugeordnet.

Ein bestimmtes Syntaxelement ist also nur dann enthalten, wenn deren Kategorie mit der der zu parsenden NALU übereinstimmt.

Zur Verdeutlichung soll folgendes Beispiel betrachtet werden:

Das Syntaxelement *slice_data()* wird von folgenden Syntaxelementen referenziert: *slice_layer_without_partitioning_rbsp()* (SLWP), *slice_data_partition_a_layer_rbsp()* (SDPa), *slice_data_partition_b_layer_rbsp()* (SDPb), *slice_data_partition_c_layer_rbsp()*.
Da *slice_data()* wie schon erwähnt *macroblock_layer()* (ML) referenziert, referenzieren die vier genannten Sytaxelemente ebenfalls ML. ML enthält Syntaxelemente der Kategorie 2 und 3. Greifen wir bespielhaft die Felder *mb_type* (Kategorie 2) und *pcm_alignment_zero_bit* (Kategorie 3) heraus. Da SLWP den Kategorien 2, 3 und 4 zugeordnet ist, sind, wenn ML von diesem Syntaxelement referenziert wird, beide Felder enthalten. Wird ML von SDPa referenziert, ist *mb_type* enthalten, da *mb_type* wie auch SDPa der Kategorie 2 angehören. *pcm_alignment_zero_bit* ist jedoch nicht enthalten, da dieses Element der Kategorie 3 angehört. Wird ML von SDPb referenziert, so ist *pcm_alignment_zero_bit*, aber nicht *mb_type* enthalten, da bei SDPb Kategorie 3 vorliegt.

3.4.3.1 (b) Descriptoren und deren Anwendung

Durch Descriptoren wird der Parsingprozess für Syntaxelemente beschrieben. Ausgangslage vor der Anwendung eines Descriptors ist immer eine Bitsequenz und ein Pointer, welcher auf ein bestimmtes Bit dieser Sequenz zeigt. Durch Anwendung des Descriptors werden je nach Definition eine gewisse Anzahl an Bits gelesen und gleichzeitig der Pointer immer mit verschoben. Durch die jeweilige Interpretation wird dem jeweiligen Syntaxelement sein Wert zugewiesen.

Um die für das Finden von Framegrenzen erforderlichen Syntaxelemente zu extrahieren, werden die Descriptoren $ue(v)$ und $u(n)$ und $se(v)$ benötigt. Sie sind ebenfalls in [19] (Abschnitt 9.1) definiert. Da sie elementarer Bestandteil des Parsingprozesses sind, werden diese und weitere im Folgenden erläutert.

- $read\_bits(n)$:
  Zwar ist dies keiner der oben genannten Descriptoren, doch stellt er die Grundlage aller drei oben genannten dar. Über $read\_bits(n)$ wird im Standard Folgendes erwähnt: „*read_bits(n) reads the next n bits from the bitstream and advances the bitstream pointer by n bit positions.*“ Letztendlich liest dieser Descriptor also die ersten n der noch nicht gelesenen Bits und versetzt den Pointer um n Stellen.

- $u(n)$:
  Der Wert dieses Descriptors ist definiert als das Ergebnis von $read\_bits(n)$ interpretiert als unsigned Integer.
  Ist n nicht direkt als Ganzzahl oder Konstante angegeben, sondern mit dem Bezeichner v, so hängt die Anzahl der zu lesenden Bits von anderen Syntaxelementen ab.

- $ue(v)$:
  Wenn das Syntaxelement als $ue(v)$ codiert ist, dann entspricht dessen Wert dem Wert mit der Bezeichnung *codeNum*. Es gilt:

$$codeNum = 2^{leadingZeroBits} - 1 + read\_bits(leadingZeroBits)$$

  $leadingZeroBits$ ist wiederum definiert durch folgendes Konstrukt:

```
1  leadingZeroBits = -1;
2  for(int b=0;b==0;leadingZeroBits+=1){
3   b = read_bits(1);
4  }
```

  $leadingZeroBits$ gibt also die Anzahl der voranstehenden Nullen an.

  Zum besseren Verständnis wird die Bestimmung von $codeNum$ anhand des Beispiels 00110 im Folgenden erläutert. Zudem sind in Abbildung 20 ([19] entnommen) die möglichen Wertebereiche und die dafür benötigten Bitmuster angegeben. Die hier dargestellten x_i stellen die jeweils variablen Stellen des Bitmusters dar, wobei gilt $x_i \in \{0, 1\}$.

  Zunächst wird gelesen, wie viele führende Nullen auftreten. In diesem Fall sind es zwei. Dies entspricht dem Wert von $leadingZeroBits$. Nach dieser Prozedur sind die ersten drei Bits (001) gelesen und der Pointer zeigt auf das 4. Bit.

  Es werden nun durch $read\_bits(leadingZeroBits)$ die restlichen zwei Bits eingelesen (10), da $leadingZeroBits = 2$. Die eingelesenen Bits werden als unsigned Integer interpretiert. In diesem Fall ist das Ergebnis dieser Teilfunktion also 2.

  Setzt man diese Werte in die obige Formel ein, so ergibt sich:

  $codeNum = 2^2 - 1 + 2 = 5$

- $se(v)$:
  Ebenso wie bei $ue(v)$ wird zunächst der Wert von $codeNum$ ermittelt. Der Wert von $se(v)$ ist dann wie folgt definiert:

$$(-1)^{codeNum+1} \cdot Ceil(\tfrac{codeNum}{2})$$

Bit string form	Range of codeNum
1	0
0 1 x_0	1..2
0 0 1 x_1 x_0	3..6
0 0 0 1 x_2 x_1 x_0	7..14
0 0 0 0 1 x_3 x_2 x_1 x_0	15..30
0 0 0 0 0 1 x_4 x_3 x_2 x_1 x_0	31..62
...	...

Abbildung 20: Wertebereiche von codeNum

3.4.3.2 Aktivierung verschiedener Parameter Sets

Beim Parsen der Syntaxelemente muss immer darauf geachtet werden, dass das richtige PPSs bzw. SPSs verwendet wird. Wann welche Parameter Sets aktiviert werden, wird im Standard [19] im Abschnitt 7.4.1.2 beschrieben und hier kurz erläutert:

Zu jeder Zeit kann maximal ein PPS aktiviert sein, wobei zu Beginn des Decodierungsprozesses kein PPS aktiviert ist. Die Aktivierung eines PPS erfolgt, sobald eine NALU mit *nal_unit_type* $\in \{1, \ldots, 5\}$ diese referenziert. Die Referenzierung passiert hierbei durch die *pic_parameter_set_id*. War zuvor ein anderes PPS aktiviert, wird dieses deaktiviert. Wird ein neues PPS mit der gleichen ID empfangen bzw. decodiert, so ersetzt dieses das vorherige PPS mit der gleichen ID.

Die Aktivierung der SPSs funktioniert sehr ähnlich.

3.4.4 Softwareseitiges Finden von Framegrenzen in H.264-codierten Videos

Wie in Abschnitt 3.3 beschrieben, liegen die Videodaten nach dem Einlesen durch die Komponente RST264-RV als NALUs vor. Die Aufgabe der hier betrachteten Komponente ist es, die NALUs in AUs zu unterteilen. Hierzu wird im wesentlichen das in Abschnitt 3.4.2 beschriebene Verfahren angewendet. Um dies zu ermöglichen, wurden bei der Entwicklung der Komponente RST264-FD alle benötigten Descriptoren zum Extrahieren der Syntaxelemente wie in Abschnitt 3.4.3.1 (b) beschrieben implementiert, sowie die Verwaltung und Aktivierung der PPSs bzw. SPSs softwareseitig umgesetzt. Für die wichtigsten übergeordneten Strukturen, hier SPS, PPS und *slice header*, wurden eigene Klassen erstellt, welche die für das beschriebene Vorgehen benötigten Syntaxelemente enthalten. So kann bei den vielen Vergleichen von Feldern, die vorgenommen werden müssen, einfach über den Namen des Syntaxelements über ein jeweiliges Objekt darauf zugegriffen werden.

Beim Streamen mit *ffmpeg* kommt standardmäßig das sogenannte *High Profile* zum Einsatz. Dies bestätigt sich auch durch die Analyse des im SPS enthaltenen Feldes *profile_idc*. Nach Abschnitt A.2.4 im H.264-Standard [19] ist dieser Wert bei Verwendung des *High Profile* 100.

Im selbigem Abschnitt wird auch definiert, dass bei Verwendung des *High Profile* NALUs mit $nal\_unit\_type \in \{2, 3, 4\}$ nicht auftreten. Somit kann der in Abschnitt 3.4.2.1 beschriebene Sonderfall, bei dem theoretisch NALUs mit $nal\_unit\_type \in \{3, 4\}$ den Anfang eines PCP darstellen können, nicht eintreten und wird deshalb in der Komponente RST264-FD nicht behandelt.

Ansonsten entspricht das in der Komponente RST264-FD angewendete Verfahren aber vollständig dem aus Abschnitt 3.4.2.

3.4.5 Beispiel für den Einsatz der Komponente RST264-FD

Bis Zeile 5 entspricht der folgende Code dem aus Abschnitt 3.3.5. Demnach enthält die Variable *nalu_list* eine Liste aller NALUs. Zum Einteilen dieser in AUs wird in Zeile 7 ein Objekt der Klasse *Access_unit_detector* erzeugt, wobei als Parameter die Liste der NALUs angegeben wird. In Zeile 8 wird die Methode *get_access_units()* des erzeugten Objektes aufgerufen, welche eine zweidimensionale Liste zurück gibt, in der die AUs jeweils als Listen von NALUs enthalten sind.

```
1 import ...
2
3 default_pcap_filepath = 'default_stream.pcap'
4 parser = Pcap_nalu_parser(default_pcap_filepath)
5 nalu_list = parser.parse_nalus()
6
7 au_detector = Access_unit_detector(nalu_list)
8 au_list = au_detector.get_access_units()
```

Das zweite Beispiel zeigt hier die volle Schnittstelle des Konstruktors der Klasse *Access_unit_detector*. Neben dem Pflichtparameter *nalu_list* kann auch hier der *output_mode*, welcher standardmäßig auf *False* gesetzt ist, auf *True* gesetzt werden. In Folge dessen werden die Typinformationen der NALUs nach AUs unterteilt in eine txt-Datei geschrieben.

```
1 au_detector = Access_unit_detector(nalu_list, output_mode=True)
```

3.5 Komponente zum Streamen eines Videos (RST264-SV)

Nach der Bearbeitung der Videodaten durch die vorherigen Komponenten (RST264-RV und RST264-FD) liegen diese jetzt der Komponente RST264-SV als Eingabe in Form von NALUs vor, wobei diese wiederum in AUs zusammengefasst sind. Die Aufgabe dieser Komponente ist es, die NALUs passgenau in RTP-Pakete zu verpacken und das Streamen dieser zu ermöglichen. Hierzu erfolgt eine grundsätzliche Beschreibung der Realisierung der Komponente RST264-SV und eine Erläuterung, wie ein Videostrom, welcher mit dem Werkzeug RST264 erzeugt wurde, empfangen und gespeichert werden kann.

3.5.1 Verpacken von NALUs in RTP-Pakete

Das Vorgehen zum Verpacken der NALUs in RTP-Pakete richtet sich nach dem bereits in Abschnitt 3.3.3 beschriebenen Payloadformat, wobei auch hier der *Non-Interleaved Mode* verwendet wird. Durch entsprechende Anweisungen kann für jede AU eine Liste an Größen übergeben werden, nach der sich die Größe der RTP-Nutzlast richtet. Ebenso kann einfach nur eine Maximalgröße angegeben werden. Je nach Größenangaben wird dann für die einzelnen NALUs einer AU entschieden, in welcher Form sie versendet werden. Ist sie klein genug, um mit einer anderen NALU versendet werden zu können, so wird ein STAP-A verwendet. Ist die NALU zu groß, um sie in einem Paket zu versenden, so wird eine FU-A genutzt. Passt die NALU genau in ein RTP-Paket der gewünschten Größe, so kommt ein *Single NAL Unit Packet* zum Einsatz.

3.5.2 Streamen und Empfangen eines Videos

Zum Streamen des Videos werden die Pakete unter Angabe von Quell-Port, Quell-IP-Adresse, Ziel-Port und Ziel-IP-Adresse mit Hilfe einer Pythonbibliothek auf einen Socket gesendet. Empfangen werden können diese Daten dann durch einen *ffmpeg*-Befehl der folgenden Form:

ffmpeg -protocol_whitelist file,crypto,rtp,udp -i <SDP-datei>.sdp
-vcodec libx264 -f mp4 -c copy <Dateiname>

Durch *-protocol_whitelist file,crypto,rtp,udp* wird nur spezifiziert, dass diese Protokolle bzw. Pakete zugelassen werden. Durch *-vcodec libx264 -f mp4 -c copy <Dateiname>* wird der Dateiname angegeben, sowie die Codierung (H.264) und das Containerformat (mp4) festgelegt. Durch *-i <SDP-datei>.sdp* wird auf eine SDP-Datei verwiesen, welche den empfangenen Videostrom spezifiziert. Diese ist hierbei wie in Abschnitt 2.2.4 beschrieben aufgebaut. Der Inhalt muss so angepasst werden, dass er zu den für das Versenden des Videos gewählten Angaben passt.

3.5.3 Beispiel für den Einsatz der Komponente RST264-SV

Bis Zeile 9 ist der Code identisch zu dem aus Abschnitt 3.4.5. Die Variable *au_list* enthält demnach eine zweidimensionale Liste mit AUs. In Zeile 13 wird ein Objekt der Klasse *Socket_sender* erstellt und als Parameter Quell- bzw. Ziel-Port und IP-Adresse angegeben. Darüber hinaus wird auch ein Objekt der Klasse *RTP_helper* mit übergeben. Dieses *RTP_helper*-Objekt ist intern dafür zuständig, beim Verpacken der Pakete durch die Klasse vom Typ *Socket_sender* Informationen wie z.B. Zeitstempel und Sequenznummer für die Generierung des RTP-Headers bereitzustellen. Als Liste für die Zeitstempel werden in diesem Fall dieselben wie aus dem aufgezeichneten Videostrom genutzt, weshalb eine Liste dieser Zeitstempel bei der Erzeugung des *RTP_helper*-Objekts übergeben wird. In den Zeilen 15 und 16 werden alle AUs in den Puffer des *Socket_sender*-Objekts eingefügt. Hierbei wird die Methode *access_unit_to_buffer(...)* aufgerufen, welche dafür sorgt, dass eine komplette AU zunächst in RTP-Pakete verpackt und die einzelnen Pakete dann in den Puffer gespeichert werden. Der erste Parameter gibt hierbei die aktuelle AU an, der zweite Parameter gibt die maximale Nutzlastgröße für die RTP-Pakete in Byte an und der dritte optionale Parameter gibt eine Liste an maximalen Nutzlastgrößen an. In diesem Fall werden also konkret die ersten drei RTP-Pakete so generiert, dass sie maximal 500, 600 bzw. 700 Byte an Nutzlast beinhalten. Danach kommt die Grenze von 1400 Byte zum Einsatz. Die angegebene Liste kann beliebig variiert werden, sowohl in der Länge, als auch was die Werte angeht. So ist es möglich, jede AU individuell in RTP-Pakete zu verpacken.

Durch die Methode *stream()* werden alle Pakete im Puffer des *Socket_sender*-Objekts versendet. Durch die Methode *receive()* werden die Sendungen empfangen und als mp4-Datei gespeichert. Diese Methoden werden in den Zeilen 29-32 zeitgleich aufgerufen.

```
1  import threading
2  import ...
3
4  default_pcap_filepath = 'default_stream.pcap'
5  parser = Pcap_nalu_parser(default_pcap_filepath)
6  nalu_list = parser.parse_nalus()
7
8  au_detector = Access_unit_detector(nalu_list)
9  au_list = au_detector.get_access_units()
10
11 rtp_timestamp_list = parser.rtp_timestamp_list
12 rtp_helper = RTP_helper(rtp_timestamp_list)
13 sender = Socket_sender("127.0.0.1", 3010, "127.0.0.1", 4010,
       rtp_helper)
14
15 for au in au_list:
16     sender.access_unit_to_buffer(au,1400,[500,600,700])
17
18 def stream():
19     sender.send_all_packets_without_loss()
```

```
20
21  def receive():
22      dir_path_for_video_save = '/output'
23      sdp_filepath = default_sdp_file_path
24      filename = 'streaming_main'
25      ffmpeg_receive_cmd ='cd %s && ffmpeg -protocol_whitelist file,
            crypto,rtp,udp -i %s -vcodec libx264 -f mp4 -c copy %s.mp4'
26      ffmpeg_cmd = ffmpeg_receive_cmd %(dir_path_for_video_save,
            sdp_filepath,filename)
27      subprocess.call(ffmpeg_cmd, shell=True)
28
29  t1 = threading.Thread(target=receive)
30  t2 = threading.Thread(target=stream)
31  t1.start()
32  t2.start()
```

Beim Erzeugen des *Socket_sender*-Objekts kann neben den oben bereits beschriebenen Parametern auch der Parameter *calc_overhead* auf *True* gesetzt werden. Ist dies der Fall, so wird beim Verpacken jedes RTP-Pakets zusätzlich der Overhead, der aus UDP- und RTP-Header entsteht, berechnet. Der Gesamtoverhead kann dann abgefragt werden.

```
1  sender = Socket_sender("127.0.0.1", 3010, "127.0.0.1", 4010,
       rtp_helper,calc_overhead=True)
2  overhead = sender.overhead
```

3.6 Komponente zum Simulieren von Paketverlusten (RST264-PL)

Durch diese Komponente des RST264-Werkzeugs ist es möglich, gezielt Pakete beim Sendevorgang nicht mit zu berücksichtigen und somit Paketverluste zu simulieren. Hierzu können entweder durch zwei zur Auswahl stehende stochastische Verfahren Paketverluste modelliert und in csv-Dateien gespeichert werden, oder aber ein gewünschter modellierter Paketverlust aus einer csv-Datei eingelesen werden.

3.6.1 Modellierung von Paketverlusten

Es gibt eine Vielzahl an Möglichkeiten und stochastischen Verteilungen, um Paketverluste zu modellieren. Im Rahmen dieser Arbeit und somit auch in der Komponente RST264-PL liegt der Fokus auf folgenden zwei Varianten:

- **Modellierung von Paketverlusten durch Normalverteilung der Verzögerung eines einzelnen Pakets:**
  Bei dieser Variante werden den einzelnen Paketen zunächst Sendezeitpunkte und Deadlines für die Ankunft zugeordnet. Anschließend wird für jedes Paket anhand

einer "abgeschnittenen" (engl. truncated) Normalverteilung zufällig bestimmt, um wie viel Zeit sich dieses verspätet. Überschreitet die Summe aus Sendezeitpunkt und Verzögerung die zuvor bestimmte Deadline für das Paket, so gilt dies als verloren und wird beim Sendeprozess nicht berücksichtigt. Neben der Normalverteilung könnten hierfür auch andere Verteilungen gewählt werden.

- **Modellierung von Paketverlusten durch das Gilbert-Elliott Modell:**
  Das in dieser Arbeit verwendete Modell stimmt mit dem aus [11] überein und hat den in Abbildung 21 dargestellten Aufbau (Abbildung ebenfalls aus [11] übernommen). Es besteht aus zwei Zuständen: Good (G) mit der dazugehörigen Verlustwahrscheinlichkeit von $(1-k)$ und Bad (B) mit der Verlustwahrscheinlichkeit $(1-h)$. Ebenso beinhaltet das Modell die Wahrscheinlichkeit p für den Übergang von Good zu Bad und die Wahrscheinlichkeit r für den Übergang von Bad zu Good. Ursprünglich war dieser Ansatz für die Modellierung von Bitfehlern gedacht, doch in [11] wird dieser z.B. auch für die Modellierung von Paketverlusten genutzt und so auch in dieser Arbeit bzw. in der Komponente RST264-PL.

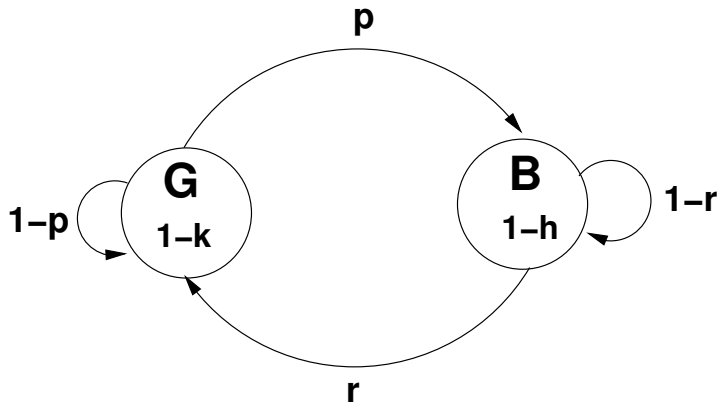

Abbildung 21: Gilbert-Elliott Modell

3.6.2 Beispiel für den Einsatz der Komponente RST264-PL

Gegenüber Abschnitt 3.5.3 haben sich hier nur die Zeilen 17-24 verändert. In den Zeilen 17-19 werden unter Berücksichtigung des momentanen Pufferinhaltes mittels der Methode *simulate_packet_loss_norm_dist(...)* Paketverluste simuliert und diese Simulation in einer csv-Datei gespeichert. Beim Aufruf der Methode wird die Verteilung durch die Angabe entsprechender Parameter spezifiziert. Aus der gespeicherten csv-Datei wird in Zeile 21 eine Liste mit Wahrheitswerten ausgelesen. Für jedes zu versendende Paket im Puffer des Senders enthält diese *True* oder *False*. Das Senden der RTP-Pakete erfolgt wieder in der Methode *stream()*. In dieser wird wiederum die Methode *send_all_packets_and_simulate_loss(...)* aufgerufen und hierbei die Liste mit den Wahrheitswerten übergeben. Ist der entsprechende Wert für ein RTP-Paket in der Liste *True*, so wird das dazugehörige RTP-Paket beim Senden nicht berücksichtigt.

```
1  import threading
2  import ...
3
4  default_pcap_filepath = 'default_stream.pcap'
5  parser = Pcap_nalu_parser(default_pcap_filepath)
6  nalu_list = parser.parse_nalus()
7
8  au_detector = Access_unit_detector(nalu_list)
9  au_list = au_detector.get_access_units()
10
11 rtp_timestamp_list = parser.rtp_timestamp_list
12 rtp_helper = RTP_helper(rtp_timestamp_list)
13 sender = Socket_sender("127.0.0.1", 3010, "127.0.0.1", 4010,
       rtp_helper)
14 for au in au_list:
15     sender.access_unit_to_buffer(au, 1400, [500, 600, 700])
16
17 random_seed = 42
18 csv_filename = "packet_loss_main_simulation"
19 simulate_packet_loss_norm_dist(sender.buffer,random_seed,
       csv_filename,fps=30,mean=0.125,std=0.025,max_delay=0.18)
20
21 loss_array = read_loss_array_from_csv(project_path + "output/" +
       csv_filename + ".csv")
22
23 def stream():
24     sender.send_all_packets_and_simulate_loss(loss_array)
25
26 def receive():
27     dir_path_for_video_save = project_path_for_cmd + 'output'
28     sdp_filepath = default_sdp_file_path
29     filename = 'streaming_main'
30     ffmpeg_receive_cmd ='cd %s && ffmpeg -protocol_whitelist file,
           crypto,rtp,udp -i %s -vcodec libx264 -f mp4 -c copy %s.mp4'
31     ffmpeg_cmd = ffmpeg_receive_cmd %(dir_path_for_video_save,
           sdp_filepath,filename)
32     subprocess.call(ffmpeg_cmd, shell=True)
33
34 t1 = threading.Thread(target=receive)
35 t2 = threading.Thread(target=stream)
36 t1.start()
37 t2.start()
```

4 Fallstudien

In diesem Kapitel werden mehrere Fallstudien durchgeführt. Hierbei wird zunächst die Laufzeit der wichtigsten Komponenten des RST264-Werkzeugs untersucht. Anschließend erfolgt eine Fallstudie zur Untersuchung der Anzahl der benötigten RTP-Pakete und des dabei entstehenden Overheads. Abschließend wird untersucht, wie sich die Verwendung verschiedener Paketgrößen auf die Videoqualität auswirkt.

Alle im Rahmen dieser Fallstudien verwendeten oder erzeugten Dateien können [2] entnommen werden.

4.1 Fallstudie zur Untersuchung der Laufzeit der wichtigsten Funktionalitäten des RST264-Werkzeugs

In diesem Abschnitt werden die wichtigsten Funktionalitäten des Tools RST264 hinsichtlich ihrer Laufzeit untersucht. Zuerst wird analysiert wieviel Zeit benötigt wird, um alle NALUs aus einem gegebenen Videostrom zu reassemblieren, danach erfolgt die Ermittlung des Zeitbedarfs für die Einteilung der NALUs in AUs und abschließend wird darauf eingegangen wie groß die benötigte Laufzeit zum Verpacken von NALUs in RTP-Pakete ist.

Alle folgenden Messungen wurden auf einem MacBook Pro mit 2,8 GHz Intel Core i7-Prozessor mit 16 GB Arbeitsspeicher unter Einsatz des Betriebssystems macOS Sierra (Version 10.12.6) sowie Python 3.6.1 durchgeführt.

4.1.1 Reassemblieren von Videoströmen

Zur Untersuchung der benötigten Zeit für die Reassemblierung von NALUs aus Videoströmen werden acht Videos mit verschiedener Länge (5s, 10s, 20s, 30s, 40s, 50s, 60s, 120s) auf gleiche Weise gestreamt und der Videostrom aufgezeichnet. Die so erstellten pcap-Dateien sind die Grundlage dieser Untersuchung. Ein Video mit kürzerer Spieldauer ist hierbei immer eine Teilsequenz aller längeren Videos. Die Videos enthalten somit, soweit im Rahmen der unterschiedlichen Längen möglich, den gleichen Inhalt. Pro Video wurden 50 Durchgänge durchgeführt und jeweils der Durchschnitt gebildet.

Abbildung 22 stellt dar, wie die benötigte Laufzeit für die Reassemblierung von der Anzahl der im Video enthaltenen NALUs abhängt. In Abbildung 23 wird die benötigte Zeit unter Berücksichtigung der einzulesenden RTP-Pakete gezeigt.

Für beide Messreihen wurde eine Regression durchgeführt, die ebenfalls in den Abbildungen enthalten ist. Es ist zu erkennen, dass die benötigte Zeit in Sekunden in guter Näherung jeweils linear von der Anzahl der NALUs bzw. von der Anzahl der einzulesenden RTP-Pakete abhängt. Für die Reassemblierung von je 500 NALUs wird eine Zeit von ca. 3 Sekunden benötigt. Die Reassemblierung von 2500 RTP-Paketen nimmt ca. 1,7 Sekunden in Anspruch.

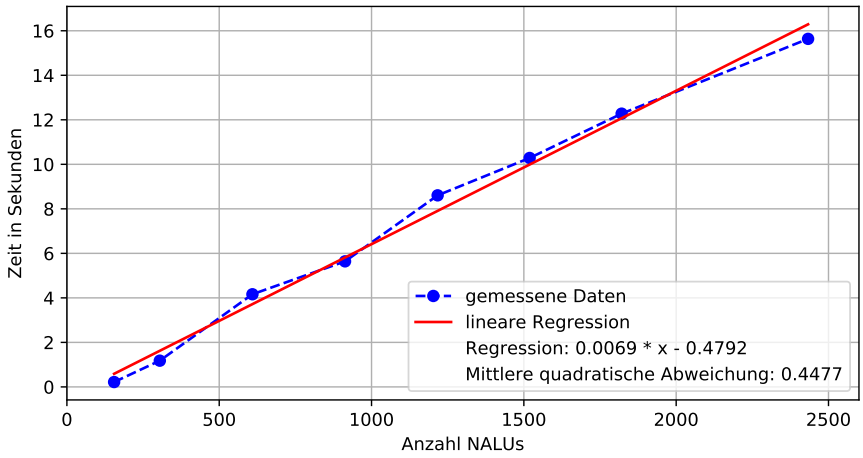

Abbildung 22: Zeitbedarf zum Reassemblieren der Videodaten in Abhängigkeit von der Anzahl der NALUs

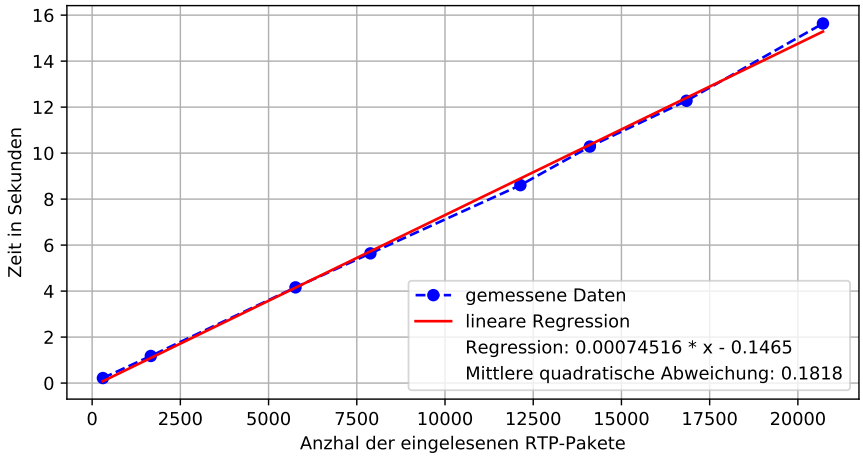

Abbildung 23: Zeitbedarf zum Reassemblieren der Videodaten in Abhängigkeit von der Anzahl der einzulesenden RTP-Pakete

4.1.2 Zusammenfassung von NALUs nach Framegrenzen

Zur Messung der benötigten Zeit, um die NALUs durch die Komponente RST264-FD frameweise zusammenzufassen, wurden die gleichen Videos und auch die gleichen Videoströme wie im vorherigen Abschnitt verwendet. Auch hier wurden 50 Zeitmessungen vorgenommen und die reine Zeit für das Erkennen der Framegrenzen gemessen. Die Zeit zum Reassemblieren der NALUs wurde nicht mit einberechnet. Abbildung 24 zeigt die benötigte Zeit in Sekunden in Abhängigkeit von der Anzahl der im Strom enthaltenen NALUs.

Auch für diese Messungen wurde eine Regression durchgeführt und auch hier ist zu erkennen, dass die benötigte Laufzeit, in diesem Fall zum Erkennen von Framegrenzen und Einteilen der NALUs in AUs, linear von der Anzahl der vorliegenden NALUs abhängt. Für die Zuordnung von je 500 NALUs wird eine Zeit von ca. 3,6 Sekunden benötigt.

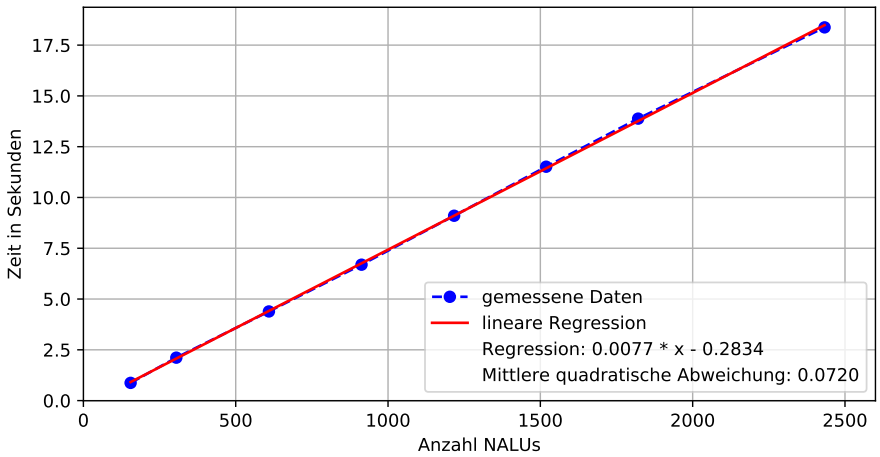

Abbildung 24: Zeitbedarf zum Erkennen von Framegrenzen in Abhängigkeit von der Anzahl der NALUs

4.1.3 Verpacken der NALUs in RTP-Pakete

Zur Untersuchung der benötigten Zeit zum Verpacken der extrahierten NALUs in RTP-Pakete wurden als Referenz wieder die Videos aus Abschnitt 4.1.1 herangezogen (außer das Video mit 120s Länge). Zudem wurde die Auswirkung von verschiedenen Maximalgrößen für die RTP-Nutzlast untersucht. Als Größen wurden 128 Byte, 256 Byte, 512 Byte, 769 Byte, 1024 Byte und 1280 Byte genutzt.

Auch hier bezieht sich die gemessene Zeit ausschließlich auf die Zeit für das Verpacken. Es fließt weder die benötigte Zeit für die Reassemblierung der NALUs, noch die Dauer für die Einteilung nach AUs mit ein. Abbildung 25 zeigt die benötigte Zeit in Abhängigkeit von verwendeten maximalen RTP-Nutzlastgrößen und der Anzahl der enthaltenen NALUs. Abbildung 26 zeigt die durchschnittliche Zeit, die zum Verpacken einer NALU benötigt wird.

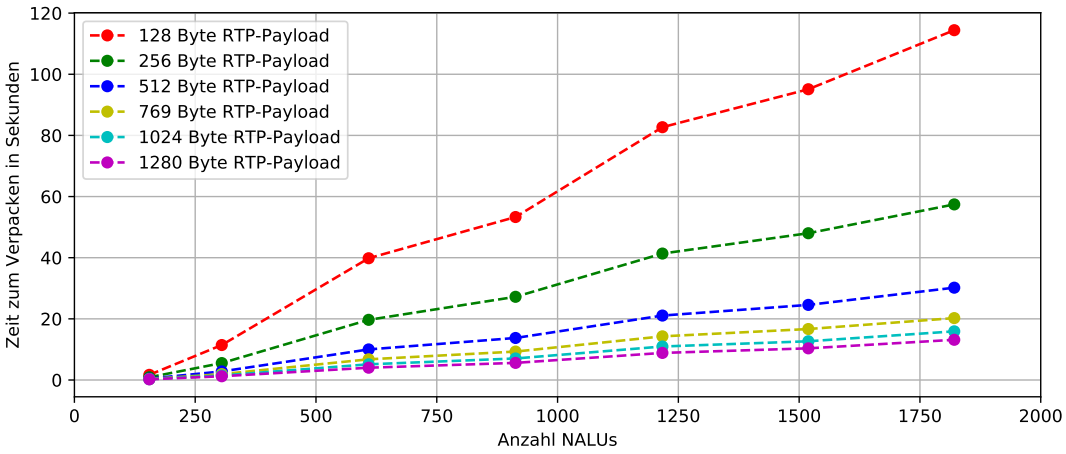

Abbildung 25: Zeitbedarf zum Verpacken eines Videos in RTP-Pakete in Abhängigkeit von der Anzahl der NALUs und der verwendeten Paketgröße

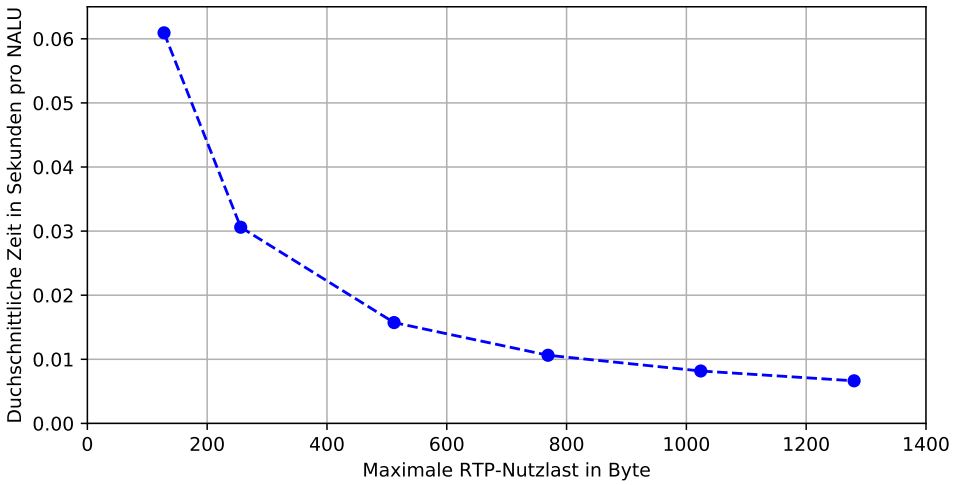

Abbildung 26: Durchschnittlicher Zeitbedarf für das Verpacken einer NALU

Den beiden Abbildungen ist zu entnehmen, dass auch die Zeit zum Verpacken von NALUs in etwa linear von der Anzahl der vorliegenden NALUs abhängt. Es ist jedoch auch zu sehen, dass die benötigte Zeit annäherungsweise exponentiell von der verwendeten Maximalgröße für die RTP-Nutzlast abhängig ist. Während für das Verpacken einer NALU in RTP-Pakete mit maximal 128 Byte Nutzlast ca. 60 ms benötigt werden, sinkt diese Zeit für die Nutzlastgröße von 256 Byte auf ca. 30 ms und für eine Nutzlastgröße von 1024 sogar auf unter 10 ms. Dies ist vor allem darauf zurückzuführen, dass bei kleineren maximalen RTP-Nutzlastgrößen auch entsprechend mehr RTP-Pakete für je eine NALU benötigt werden.

4.1.4 Zwischenfazit

Aus dieser Fallstudie geht hervor, dass die Laufzeit der untersuchten Komponenten linear von der Anzahl der im eingelesenen Videostrom enthaltenen NALUs abhängig ist. So ist bei einem sehr langen Videostrom auch eine verhältnismäßig akzeptable Laufzeit des Werkzeugs RST264 zu erwarten. Die Fallstudie hat jedoch auch gezeigt, dass bei einer relativ kleinen maximalen Nutzlastgröße für die RTP-Pakete mit einer deutlich erhöhten Laufzeit, zumindest für das Verpacken der NALUs in RTP-Pakete, zu rechnen ist.

4.2 Fallstudie zur Untersuchung der benötigten RTP-Pakete und des daraus resultierenden Overheads

In dieser Fallstudie wird zum einen betrachtet, wie sich die maximale RTP-Nutzlastgröße auf die Anzahl der benötigten RTP-Pakete auswirkt. Zum anderen wird der beim Verpacken der Videodaten durch die Protokolle UDP und RTP entstehende Overhead untersucht, auch dies erfolgt unter Berücksichtigung unterschiedlicher maximaler RTP-Nutzlastgrößen.

Als Ausgangsvideo wird ein 20s langes Video herangezogen (ebenfalls in [2] zu finden) und als maximale RTP-Nutzlastgrößen werden wie in der vorherigen Fallstudie die Größen 128 Byte, 256 Byte, 512 Byte, 769 Byte, 1024 Byte und 1280 Byte betrachtet.

4.2.1 Erzielte Resultate

Dass bei kleineren maximalen RTP-Nutzlastgrößen auch entsprechend mehr RTP-Pakete für je eine NALU benötigt werden, wurde schon im vorherigen Abschnitt festgestellt. Dies lässt sich auch in Abbildung 27 erkennen. Die exponentielle Abnahme der benötigten RTP-Pakete erklärt sich dadurch, dass ein RTP-Paket mit doppelter maximaler Nutzlastgröße auch doppelt so viel Nutzlast aufnehmen kann. Die unterschiedlichen Anzahlen an benötigten RTP-Paketen je maximaler Nutzlastgröße führt auch zu unterschiedlich viel Overhead. In Abbildung 28 ist dieser prozentual in Abhängigkeit von der verwendeten maximalen Nutzlastgröße dargestellt. Da der Gesamtoverhead maßgeblich von der Anzahl der benötigten RTP-Pakete abhängig ist, ist auch hier eine exponentielle Abnahme zu erkennen.

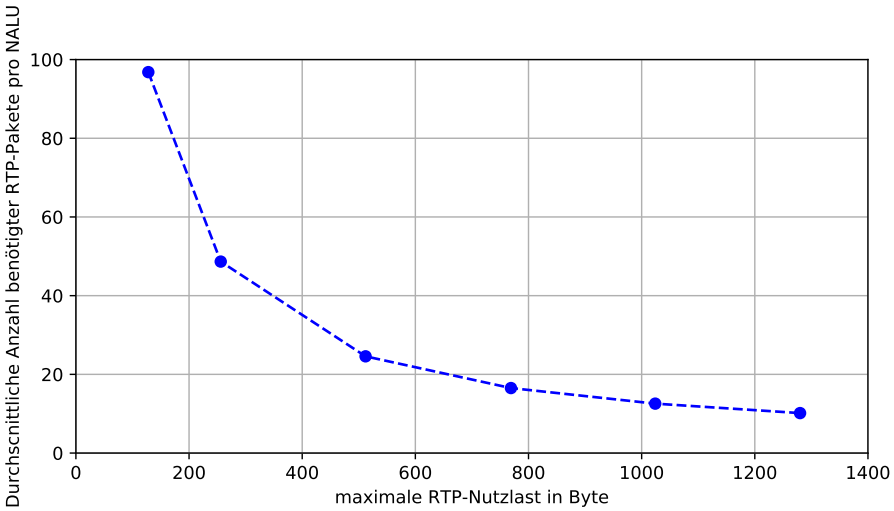

Abbildung 27: Durchschnittliche Anzahl benötigter RTP-Pakete zum Versenden einer NALU in Abhängigkeit von der maximalen RTP-Nutzlastgröße

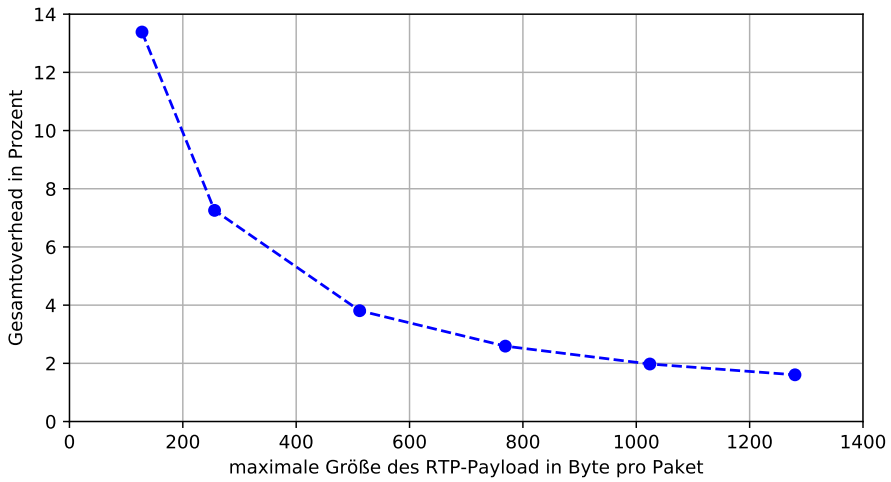

Abbildung 28: Overhead in Abhängigkeit von der maximalen RTP-Nutzlastgröße

4.2.2 Zwischenfazit

Aus dieser Fallstudie geht hervor, dass bei der Wahl von kleinen maximalen RTP-Nutzlastgrößen mit einer deutlich erhöhten Anzahl an benötigten RTP-Paketen und einem entsprechend hohen Overhead zu rechnen ist. Aufgrund dessen ist eine Nutzlastgröße von 128 Byte, mit der mehr als 10% Overhead generiert werden, zu vermeiden und wird in den folgenden Fallstudien nicht mehr berücksichtigt. Auch eine Nutzlastgröße von 256 Byte sollte eher nicht verwendet werden, sie wird der Anschaulichkeit wegen aber in weiteren Fallstudien herangezogen.

4.3 Fallstudie zur Untersuchung der Auswirkung von verschiedenen maximalen RTP-Nutzlastgrößen auf die Qualität des übertragenen Videostroms

In dieser Fallstudie wird untersucht, wie sich die Verwendung von verschiedenen maximalen RTP-Nutzlastgrößen auf die Videoqualität eines Videostroms auswirkt. Als Ausgangslage dient ein Video mit einer Länge von 20 Sekunden und einer Frameanzahl von 600. Beim Streamen dieses Videos wird neben den maximalen RTP-Nutzlastgrößen (256 Byte, 512 Byte, 769 Byte, 1024 Byte und 1280 Byte) auch der Abstand zwischen den IDR-Frames variiert. Hierbei werden 3 Variationen herangezogen: Mindestens jedes 5. Frame ein IDR-Frame, mindestens jedes 20. Frame ein IDR-Frame und die Standardkonfiguration von *ffmpeg*, bei der im vorliegenden Video in etwa jedes 100. Frame ein IDR-Frame ist.

Um die Analyse der Bildqualität zu ermöglichen, werden die Videoströme mittels *ffmpeg* empfangen und gespeichert. Ein vorab durchgeführter Testdurchlauf hat ergeben, dass ein empfangenes Video sich auch ohne Paketverluste leicht gegenüber seinem Original verändert. Dies ist für den Betrachter so gut wie nicht zu erkennen, es schlägt sich jedoch in den Messungen geringfügig nieder. Dass diese Veränderungen nicht durch die Verarbeitung der Daten durch das RST264-Tool verursacht werden, wurde wie folgt sichergestellt: Ein Video wurde mit *ffmpeg* gesendet und auch direkt wieder mittels *ffmpeg* empfangen, ohne dass das Werkzeug RST264 zum Einsatz kam. Auch hierbei traten die leichten Veränderungen auf und werden folglich nicht durch die Verarbeitung der Daten durch das RST264-Tool verursacht. Da in dieser Fallstudie ausschließlich die Auswirkungen verschiedener maximaler RTP-Nutzlastgrößen bzw. verschiedener Anzahlen an enthaltenen IDR-Frames auf die Bildqualität eines übertragenen Videostroms bei Paketverlusten untersucht werden sollen, muss von den oben beschriebenen kleinen Veränderungen im Video abstrahiert werden. Aufgrund dessen wird als Referenz für die vorgenommene Analyse der Videoqualität nicht das Originalvideo herangezogen, sondern jeweils ein Video, welches ohne Paketverluste übertragen wurde. Dies unterscheidet sich nicht gegenüber einem Video, welches ohne Einsatz des RST264-Werkzeugs empfangen wurde, was wiederum die korrekte Verarbeitung der Daten unterstreicht.

4.3.1 Möglichkeiten zur Analyse der Videoqualität

Im Rahmen dieser Arbeit werden aufgrund ihrer einfachen Anwendbarkeit zur Analyse der Videoqualität die oft genutzten Metriken Mean Squared Error (MSE), Peak-Signal-to-Noise-Ratio (PSNR) und Structural Similarity (SSIM) verwendet. Im Folgenden sollen diese kurz erläutert werden, wobei die Formeln hierfür [8] entnommen sind.

Es sei darauf hingewiesen, dass diese Metriken in der Regel nur eine grobe Einschätzung über die Videoqualität geben und nicht immer mit dem tatsächlichen Empfinden eines menschlichen Betrachters übereinstimmen. Diese Problematik wird unter anderem in dem, von dem Streamingportal Netflix veröffentlichten, Artikel [15] oder in [23] thematisiert. Metriken oder Verfahren, mit denen die Bewertung der Videoqualität zuverlässiger vorgenommen werden kann, erfordern einen wesentlich höheren Aufwand und kommen deshalb im Rahmen dieser Arbeit nicht zum Einsatz.

4.3.1.1 Mean Squared Error (MSE)

Bei der Berechnung des MSE zwischen zwei Bildern wird die Summe über die quadrierte Differenz aller Pixel gebildet und diese durch die Anzahl der enthaltenen Pixel geteilt. Es wird also entsprechend der Bezeichnung der Metrik der mittlere quadratische Fehler zwischen zwei Bildern berechnet. Dies lässt sich durch folgende Formel ausdrücken, wobei N die Anzahl der Pixel ist und X_i bzw. Y_i jeweils den i-ten Pixel darstellen:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (X_i - Y_i)^2$$

Da mit MSE der mittlere quadratische Fehler berechnet wird, ist bei einem erhöhten Wert von einer schlechten Bildqualität auszugehen.

4.3.1.2 Peak-Signal-to-Noise-Ratio (PSNR)

PSNR kann mit Spitzen-Signal-Rausch-Verhältnis ins deutsche übersetzt werden. Berechnet wird dieses nach folgender Formel, wobei 255 die maximale Intensität des Bildes darstellt.

$$PSNR = 10 \cdot log_{10}(\frac{255^2}{MSE})$$

Ein Nachteil, der sich bei dieser Berechnung ergibt, ist: Werden zwei identische Bilder verglichen, so ist der MSE null und somit ist der Wert für PSNR in diesem Fall nicht definiert. Damit diese weit verbreitete Metrik dennoch im Rahmen dieser Arbeit verwendet werden kann, wird der Wert von PSNR hier in diesem Fall auf 50 festgelegt.

Da durch PSNR das Verhältnis aus Signal und einem Rauschen berechnet wird, weist ein hoher Wert dieser Metrik auf eine tendenziell gute Bildqualität hin.

4.3.1.3 Structural Similarity (SSIM)

Die SSIM ist eine Variante zur Berechnung der Bildqualität gegenüber einem Referenzbild, bei der im wesentlichen drei Faktoren mit einfließen: Luminanz, Kontrast und Struktur. Die Berechnung dieser Metrik ist etwas aufwendiger, daher wird hier keine Formel angegeben, sie kann z.B. [8] entnommen werden.

Da mit dieser Metrik die Ähnlichkeit zweier Videos bestimmt wird, weist ein hoher Wert auf eine gute Bildqualität hin.

4.3.2 Beschreibung und Analyse der Messergebnisse

Die oben dargestellten Metriken wurden jeweils für die bereits erwähnten Variationen der Videos und Paketgrößen angewendet. Es wurden pro Fall jeweils zehn Durchgänge vorgenommen und der Mittelwert gebildet.

Bei den Betrachtungen der Ergebnisse im Bezug auf den MSE (dargestellt in Abbildung 41 bis Abbildung 43) ist folgendes zu erkennen: Der maximale Abstand der IDR-Frames hat erhebliche Auswirkungen auf die Videoqualität. Während der höchste zu verzeichnende MSE-Wert mit 4000 unter Verwendung des Standard-IDR-Frameabstandes zu verzeichnen ist, liegt der höchste MSE-Wert bei einem IDR-Frameabstand von maximal 20 nur bei knapp über 3000 und bei einem IDR-Frameabstand von 5 sogar nur bei knapp über 2500. Ein kleinerer Abstand der IDR-Frames führt also zu einem kleineren MSE und wirkt sich somit positiv auf die Videoqualität aus.
Eine verringerte relative Anzahl an Paketverlusten führt zu verringerten Werten des MSE und somit zu einer besseren Videoqualität. Am deutlichsten zeigt sich dies bei Verwendung des Standard-IDR-Frameabstandes. Hier sorgt der Unterschied zwischen 0,2% und 1% Paketverlust für eine Differenz des MSE von mindestens 1000 und maximal 2000. Bei einem verringerten Abstand zwischen den IDR-Frames verringert sich auch der Unterschied der MSE-Werte, vor allem für größere Paketgrößen.
Auch die Variation der Paketgrößen beeinflusst den MSE. Dies zeigt sich vor allem bei Paketverlusten von 1% bzw. 0,5%. Hier ist der MSE beim Vergleich der kleinsten Paketgröße von 256 Byte und der größten Paketgröße von 1280 Byte bei der größten Paketgröße um bis zu 2000 geringer. Auch bei einer kleineren Anzahl an Paketverlusten zeigt sich für alle IDR-Frameabstände eine Verringerung des MSE bei steigender Paketgröße und somit eine Verbesserung der Qualität.

Bei den Betrachtungen der Ergebnisse im Bezug auf PSNR (dargestellt in Abbildung 44 bis Abbildung 46) ist folgendes zu erkennen: Auch für das PSNR wirkt sich der maximale IDR-Frameabstand aus. Während sich die Werte für einen IDR-Frameabstand von 5 zwischen knapp unter 20 und etwas unter 40 bewegen, liegen sie für den Standardabstand zwischen 15 und 30. Ein geringerer IDR-Frameabstand führt also auch zu höheren PSNR-Werten.

Ebenso wie ein verringerter IDR-Frameabstand führt auch ein geringerer relativer Paketverlust zu einer höheren PSNR und somit zu einer besseren Videoqualität. Bei den IDR-Frameabständen von 5 bzw. von 20 verursacht der Unterschied zwischen 0,2% und 1% Paketverlust eine Differenz des PSNR von etwa 10. Für große Paketgrößen trifft dies auch für den Standard-IDR-Frameabstand zu.
In allen betrachteten Fällen führt ein erhöhter Wert der verwendeten Paketgröße ebenfalls zu einem erhöhten Wert des PSNR und somit zu einer besseren Bildqualität. Die Unterschiede sind am deutlichsten bei einem IDR-Frameabstand von 5 bzw. 20 zu erkennen. Hier beträgt die Differenz zwischen dem PSNR bei 256 Byte Paketgröße und 1280 Byte Paketgröße für alle betrachteten Paketverlustanzahlen ca. 10.

Die Beobachtungen bezüglich des PSNR lassen sich weitestgehend auf die Beobachtungen für die SSIM übertragen, welche in Abbildung 47 bis Abbildung 49 dargestellt sind. Auch hier führt ein kleiner maximaler IDR-Frameabstand zu einem erhöhten Wert der Metrik, was auf einen positiven Einfluss auf die Bildqualität schließen lässt. Während die SSIM für fast alle Messungen mit IDR-Frameabstand 5 bzw. 20 zwischen 0,7 und 0,9 liegt, bewegen sich fast alle Werte bei Anwendung des Standardabstandes zwischen 0,65 und 0,8.
Ebenso wie bei den vorherigen Metriken führt auch bei der Betrachtung der SSIM ein geringerer prozentualer Paketverlust zu einer besseren Bildqualität und somit zu höheren SSIM. Bei einem kleineren IDR-Frameabstand ist dies tendenziell deutlicher zu sehen.
Auch die verwendete Paketgröße hat Einfluss auf die SSIM. In fast allen betrachteten Fällen führt eine Erhöhung der Paketgröße von 256 Byte auf 1280 Byte zu einem Zuwachs des Wertes um 0,1 und somit zu einer tendenziell besseren Bildqualität.

Insgesamt lässt sich also feststellen, dass folgende Faktoren zu einer Verbesserung der Bildqualität beitragen: Geringer Paketverlust, erhöhte Anzahl der IDR-Frames und erhöhte maximale RTP-Nutzlastgöße.

Um die Ursachen hierfür zu erläutern, werden Abbildung 38 bis Abbildung 39 herangezogen. In diesen drei Grafiken wird jeweils in rot die Anzahl der Paketverluste pro Frame dargestellt und in blau die gemessene Qualität eines einzelnen Frames anhand der Metrik SSIM. Die grünen Balken kennzeichnen, welche der Frames IDR-Frames sind.

Für alle drei Abbildungen ist die relative Anzahl der Paketverluste gleich, die absolute Anzahl hingegen nicht. Grund hierfür ist, dass sowohl bei kleineren maximalen RTP-Nutzlastgrößen, als auch bei bei einer erhöhten Anzahl an IDR-Frames aufgrund der erhöhten Größe dieser Frames insgesamt mehr Pakete zum Versenden benötigt werden. Dies ist auch der Grund dafür, dass die Peaks bezüglich der Anzahl der Paketverluste oft bei einem IDR-Frame zu finden sind.

- **Geringer Paketverlust:**
  Eine geringere Anzahl von Paketverlusten führt zu einer besseren Qualität, dies ist selbsterklärend. Umso mehr Pakete korrekt übertragen werden, umso mehr der versendeten Daten stehen dem Empfänger zur Decodierung des Videos zur Verfügung.

- **Erhöhte Anzahl der IDR-Frames:**
  Bei einer erhöhten Anzahl an IDR-Frames ist zwar die absolute Anzahl an Paketverlusten deutlich höher, dennoch führt dies zu einer Verbesserung der Videoqualität. Warum dies der Fall ist, lässt sich gut durch den Vergleich von Abbildung 38 und Abbildung 39 erkennen. Bei einem Paketverlust in Abbildung 38, bei dem die Abstände zwischen IDR-Frames recht groß sind, führt jeder neue Paketverlust, mit wenigen Ausnahmen, zu einer kontinuierlichen Verschlechterung der Qualität der einzelnen Frames. Diese wird erst wieder besser, sobald das nächste IDR-Frame empfangen wird. Gleiches ist auch in Abbildung 39 zu erkennen. Da hier die IDR-Frames aber deutlich dichter beieinander liegen, verbessert sich die Videoqualität erheblich schneller wieder und so auch die Qualität des gesamten Videos.
- **Erhöhte maximale RTP-Nutzlastgöße:**
  Warum eine erhöhte maximale Nutzlastgröße zu einer verbesserten Bildqualität führt, lasst sich anhand der Abbildungen nicht eindeutig erkennen. Bei dem Vergleich von Abbildung 39 und Abbildung 40 zeigt sich jedoch der Trend, dass ein einzelner Paketverlust bei einer großen Paketgröße nur zu einer minimal stärkeren Verschlechterung der Videoqualität führt, obwohl die Anzahl von verloren gegangenen Daten um den Faktor fünf größer ist. In Anbetracht dieser Tendenz ist die Videoqualität maßgeblich von der Anzahl der Paketverluste abhängig. Diese ist bei kleinen maximalen RTP-Nutzlastgrößen wesentlich höher und somit ist in diesen Fällen auch mit einer schlechteren Bildqualität zu rechnen.

4.3.3 Zwischenfazit

Wie bereits erwähnt, führen die Faktoren geringer Paketverlust, erhöhte Anzahl der IDR-Frames und erhöhte maximale RTP-Nutzlastgöße zumindest für die hier betrachteten Fälle zu einer Verbesserung der Bildqualität. Doch wie gut lassen sich diese Faktoren beeinflussen, um die Qualität eines Videostroms zu verbessern?

Die Anzahl der Paketverluste lässt sich seitens eines Senders nur sehr schwer beeinflussen. Zwar gibt es Möglichkeiten, Ressourcen für das Versenden von Daten zu reservieren (*IntServ*) oder Paketen mit Echtzeitanforderungen eine erhöhte Priorität zuzuweisen (*DiffServ*) und so Paketverluste zu minimieren. Diese führen in der Praxis aufgrund diverser Schwierigkeiten bei der Umsetzung aber nur selten zum gewünschten Ergebnis.
Der Abstand der IDR-Frames im Videostrom lässt sich im Gegensatz zum Paketverlust leicht beeinflussen. Die Erhöhung der Anzahl der IDR-Frames ist aber auch immer mit einem erhöhten Aufkommen an zu versendenden Daten verbunden und so muss abgewogen werden, ob dies im konkreten Fall eine angebrachte Alternative ist.
Die Größe der Pakete lässt sich ebenso wie die Anzahl der IDR-Frames mit wenig Aufwand variieren, zudem führt sie nur bei einer extrem kleinen Wahl der Paketgröße zu einer deutlich größeren Datenmenge. Dieser Faktor ist von den drei vorgestellten also der, welcher am ehesten beeinflusst werden sollte. Dies zeigt auch, dass das entwickelte RST264-Werkzeug sehr sinnvoll zur Verbesserung der Videoqualität eingesetzt werden kann, da es eine einfache Möglichkeit bietet, die Paketgrößen individuell anzupassen.

Es sei an dieser Stelle angemerkt, dass die in diesem Abschnitt herausgearbeiteten Erkenntnisse mit dem Wissen zu betrachten sind, dass pro Ausgangslage nur jeweils zehn Messungen durchgeführt wurden. Daher sind die in den betrachteten Grafiken dargestellten Mittelwerte nur in gewissem Maße repräsentativ. Einen Eindruck bezüglich der Verlässlichkeit der dargestellten Mittelwerte geben die Abbildungen im Anhang, welche denen aus diesem Abschnitt gleichen, aber zusätzlich 95%-Konfidenzintervalle enthalten. Bei der Betrachtung dieser Grafiken fällt auf, dass die Konfidenzintervalle für die IDR-Frameabstände von 5 bzw. 20 relativ klein sind und sich nur seltenen überlappen. Demnach sind in diesen Fällen die berechneten Mittelwerte recht aussagekräftig. Weniger aussagekräftig sind die erhobenen Mittelwerte für den Standard-IDR-Frameabstand. Hier sind die bestimmten Konfidenzintervalle vergleichsweise groß und überlappen sich häufig. Gerade hier wären weitere Messungen nötig, um die Messergebnisse und die daraus geschlossenen Resultate zu stützen.

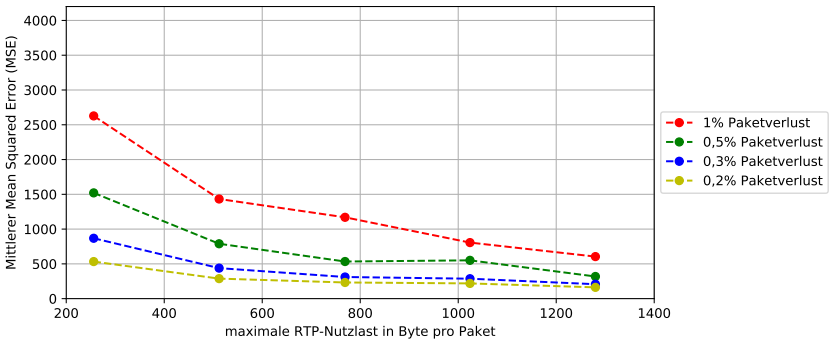

Abbildung 29: MSE unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 5

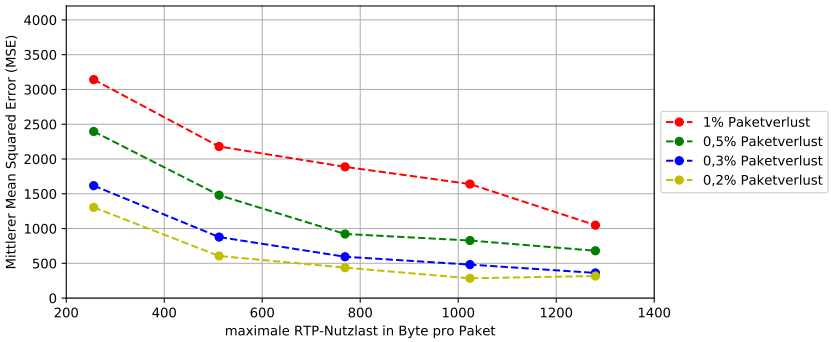

Abbildung 30: MSE unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 20

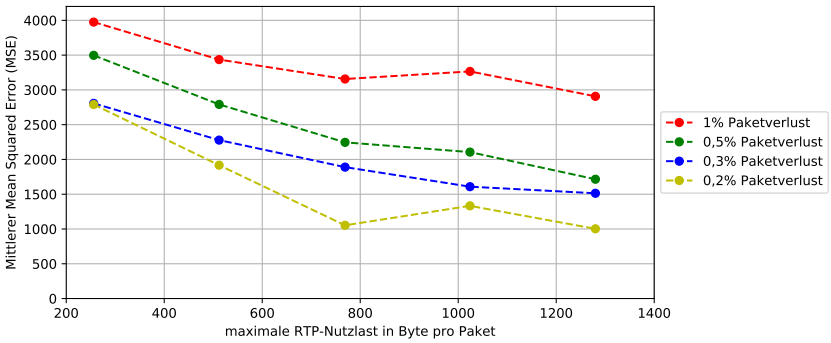

Abbildung 31: MSE unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit dem Standard-IDR-Frameabstand

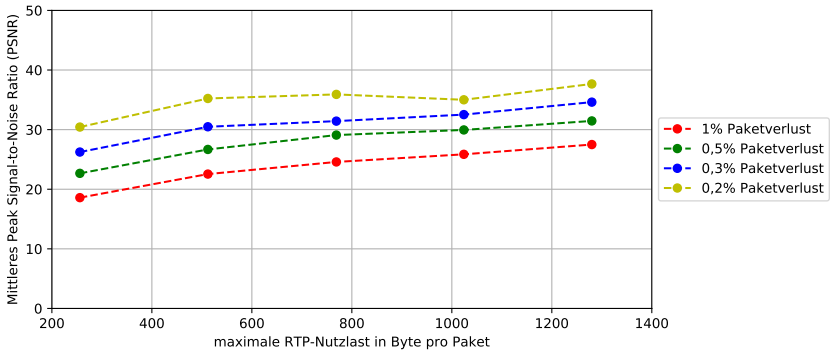

Abbildung 32: PSNR unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 5

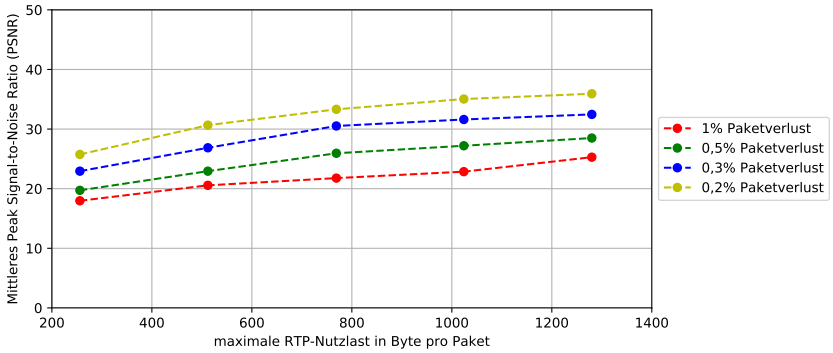

Abbildung 33: PSNR unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 20

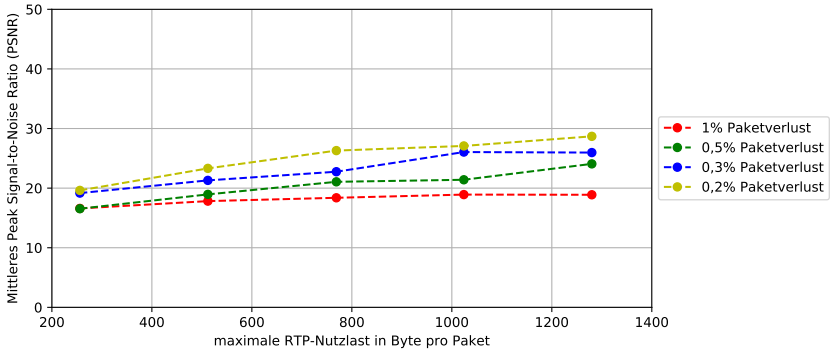

Abbildung 34: PSNR unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit dem Standard-IDR-Frameabstand

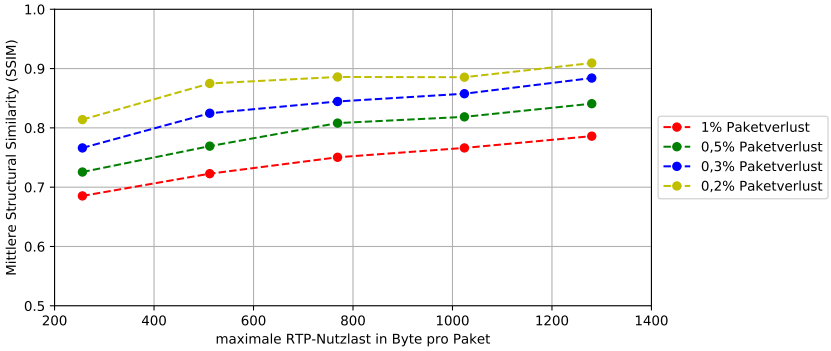

Abbildung 35: SSIM unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 5

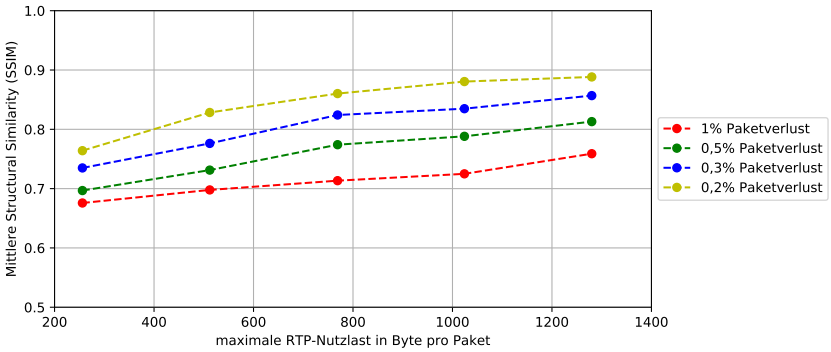

Abbildung 36: SSIM unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 20

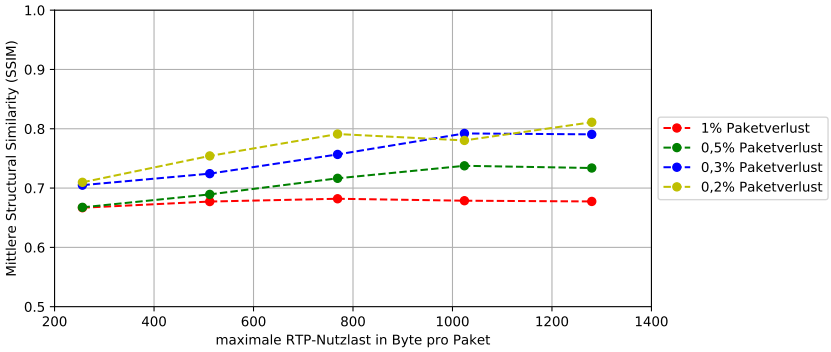

Abbildung 37: SSIM unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit dem Standard-IDR-Frameabstand

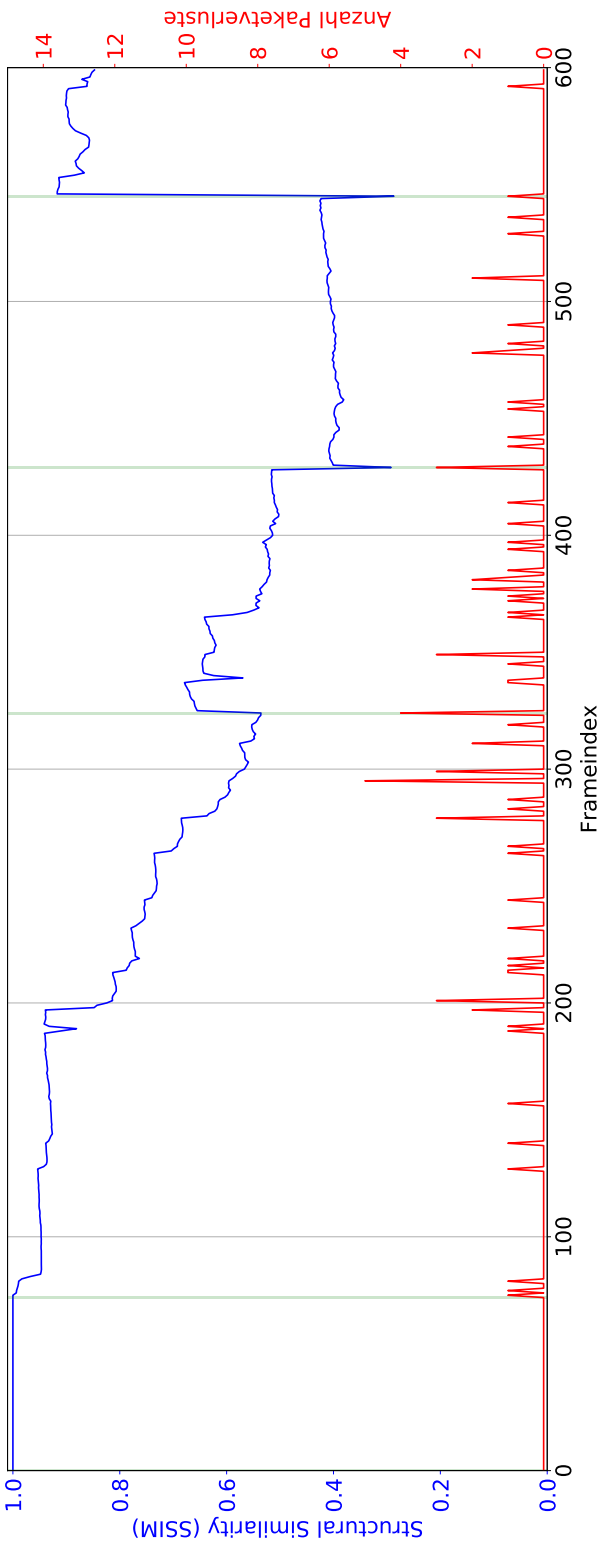

Abbildung 38: Bewertung der Videoqualität anhand von SSIM unter Berücksichtigung der Anzahl an Paketverlusten je Frame (Standard-IDR-Frameabstand, 256 Byte RTP-Nutzlastgröße)

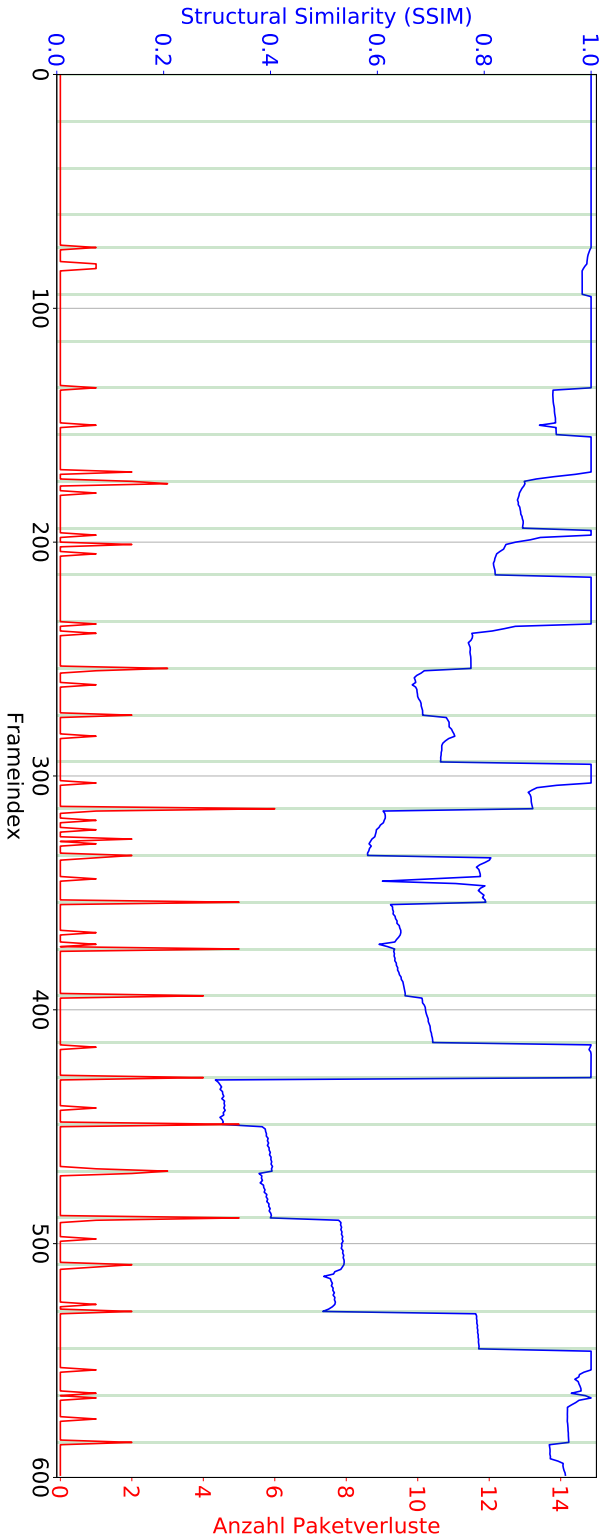

Abbildung 39: Bewertung der Videoqualität anhand von SSIM unter Berücksichtigung der Anzahl an Paketverlusten je Frame (IDR-Frameabstand max. 20, 256 Byte RTP-Nutzlastgröße)

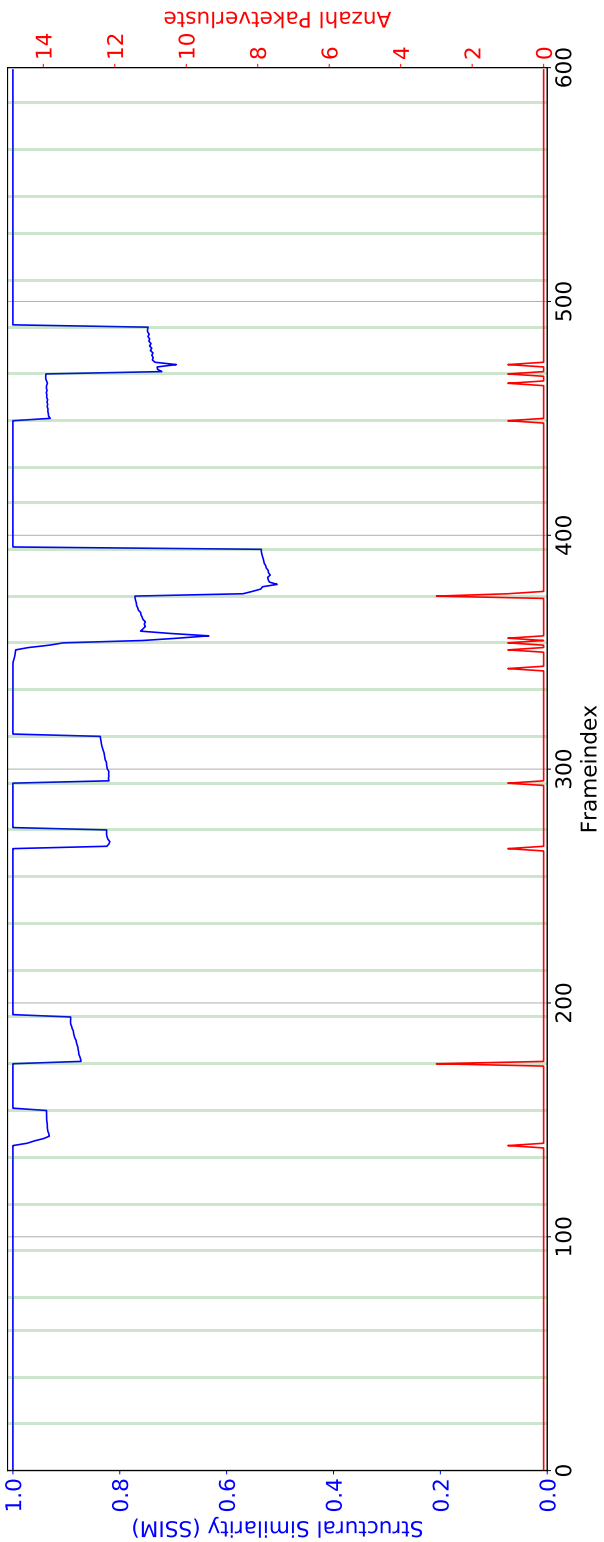

Abbildung 40: Bewertung der Videoqualität anhand von SSIM unter Berücksichtigung der Anzahl an Paketverlusten je Frame (IDR-Frameabstand max. 20, 1280 Byte RTP-Nutzlastgröße)

5 Fazit und Ausblick

5.1 Fazit

Ziel dieser Arbeit war es, ein Werkzeug zu entwickeln mit dem es möglich ist, Videodaten einzulesen, diese zu verarbeiten und anschließend passgenau für einen individuellen Videostrom zu verpacken. Hierbei sollten auch nach Wunsch Paketverluste simuliert werden können. Um dieses Ziel zu erfüllen, wurde in Kapitel 2 zunächst auf Grundlagen bezüglich des H.264-Videocodierungsstandards eingegangen und anschließend ein Überblick über häufig genutzte Kommunikationsprotokolle beim Videostreaming mit Echtzeitanforderungen gegeben.

Unter Einbezug der in Kapitel 2 betrachteten Inhalte beschreibt Kapitel 3 die Entwicklung des Werkzeugs RST264. Hierzu wurde zunächst die gewünscht Funktionalität genauer spezifiziert und eine Strukturierung nach Hauptkomponenten vorgenommen. Zu den Hauptkomponenten des Werkzeugs RST264 wurden jeweils die für die Implementation notwendigen theoretischen Hintergründe erläutert und zudem ein Beispiel gegeben, wie die Hauptkomponenten eingesetzt werden können. Neben einer Möglichkeit zum Einlesen von H.264-codierten Videos aus einem aufgezeichneten Videostrom, einem Vorgehen zum Verpacken der Videodaten in RTP-Pakete und einer Möglichkeit zum Simulieren von Paketverlusten erfolgte vor allem die Entwicklung und Vorstellung eines Verfahrens, mit dem es möglich ist, als NALUs vorliegende H.264-codierte Videodaten frameweise zu segmentieren.

In Kapitel 4 wurden mehrere Fallstudien durchgeführt und hierbei zunächst die Laufzeit des RST264-Tools untersucht. Es stellte sich heraus, dass diese bei allen untersuchten Hauptkomponenten linear von der Anzahl der in dem verwendeten Video enthaltenen NALUs abhängig ist und sich darüber hinaus die verwendete maximale RTP-Nutzlastgröße stark auf die benötigte Zeit zum Verpacken der Videodaten in RTP-Pakete auswirkt. In einer anschließenden Fallstudie wurde der entstehende Overhead und die Anzahl der benötigen RTP-Pakete bei festgelegten maximalen RTP-Nutzlastgrößen untersucht. Es zeigte sich, dass der Overhead stark von der Anzahl der Pakete abhängig ist, welche wiederum exponentiell von der verwendeten maximalen RTP-Nutzlastgröße beeinflusst wird. Die letzte Fallstudie zeigte, wie das RST264-Tool konkret eingesetzt werden kann, um die Auswirkungen verschiedener Faktoren auf die Qualität eines Videostroms zu untersuchen. Variiert wurden hierbei die Anzahl der auftretenden Paketverluste, die Anzahl der enthaltenen IDR-Frames und die Anzahl der verwendeten maximalen RTP-Nutzlastgröße. Den bei dieser Fallstudie erhobenen Daten konnte entnommen werden, dass eine geringere Anzahl an Paketverlusten, eine erhöhte Anzahl an IDR-Frames und eine hohe maximale RTP-Nutzlastgröße zu einer Verbesserung der Videoqualität führt.

Insgesamt konnten die eingangs festgelegten Anforderungen für die Realisierung eines Werkzeugs durch die Entwicklung des RST264-Tools umgesetzt werden. Somit stellt das RST264-Tool eine praktikable Möglichkeit dar, Videodaten in Form von RTP-Paketen für die Untersuchung verschiedenster Ansätze zur Verbesserung der QoE im Zusammenhang mit Videostreaming bereitzustellen.

5.2 Ausblick

In diesem Abschnitt wird betrachtet, in welcher Form die Weiterentwicklung des RST264-Tools in Zukunft sinnvoll ist und wie die Ergebnisse der Fallstudien bestätigt werden können. Abschließend wird ein allgemeiner Ausblick gegeben.

5.2.1 Erweiterung der Funktionalität des RST264-Werkzeugs

Um einen angemessenen Entwicklungsaufwand zu gewährleisten, erfolgte eine Einschränkung der Anforderungen an das RST264-Tool. Allen voran wurde aufgrund der Aktualität nur der Videocodierungsstandard H.264 herangezogen. Dieser wird zwar nach aktuellem Stand noch sehr häufig angewendet, doch in den nächsten Jahren wird der Folgestandard H.265 immer mehr an Bedeutung gewinnen. In Zukunft wäre es sinnvoll das RST264-Tool so zu erweitern, dass es auch H.265-codierte Videodaten verarbeiten kann. Hauptaufgabe hierbei wäre es, das Verfahren zum Finden von Framegrenzen zu überarbeiten. Auch müsste die interne Verarbeitung der Daten angepasst werden, was sich jedoch vermutlich aufgrund der starken Ähnlichkeit von H.264 und H.265 vergleichsweise einfach realisieren ließe.
Ebenfalls eine mögliche Erweiterung wäre es, dass neben UDP und RTP auch die Anwendung weiterer Protokollkombinationen ermöglicht werden könnte. So wäre das RST264-Tool auch einsetzbar, wenn beim zu untersuchenden Verfahren keine Echtzeitanforderungen vorliegen.
Eine schnell zu realisierende Erweiterung wäre es, die Möglichkeiten zur Simulation von Paketverlusten auszuweiten, da für eine Vielzahl an stochastischen Verteilungen Softwarebibliotheken existieren.

5.2.2 Weitere Fallstudien

Um die Ergebnisse der Fallstudien zu bestätigen, sollten zunächst weitere Fälle betrachtet werden. In der in Abschnitt 4.3 betrachteten Fallstudie wurden pro Ausgangssituation nur jeweils zehn Durchgänge durchgeführt, was teilweise zu etwas größeren Konfidenzintervallen führt. Durch eine deutliche Erhöhung der Anzahl der Durchgänge könnten diese Intervalle verkleinert werde, was zu zuverlässigeren Ergebnissen führen würde. Eine weitere Herangehensweise um die Ergebnisse zu unterfüttern wäre es, die betrachteten Fälle weiter zu variieren. Möglichkeiten hierfür sind z.B. die Untersuchung weiterer IDR-Frameabstände, die Berücksichtigung weiterer Verlustmodelle und das Heranziehen mehrerer verschiedener Videos. Weiterhin könnte in zukünftigen Fallstudien die Analyse der Videoqualität zuverlässiger gestaltet werden. Wie bereits in Abschnitt 4.3.1 erwähnt, sind die Metriken MSE, PSNR und SSIM zwar relativ einfach und schnell zu berechnen, doch existieren viele Beispiele, welche zeigen, dass die Metriken nicht immer mit der tatsächlich wahrgenommenen Videoqualität übereinstimmen. Um eine genauere Analyse der Videoqualität zu ermöglichen, können z.B. Verfahren eingesetzt werden, bei denen die menschliche Wahrnehmung simuliert wird. Ebenfalls wäre eine Befragung einer ausreichend großen

Gruppe an Testpersonen bezüglich der wahrgenommenen Videoqualität möglich, was die zuverlässigste aber auch aufwendigste Variante darstellen würde. Diese und weitere Möglichkeiten zur Beurteilung der QoE lassen sich beispielsweise in [17] nachvollziehen. Die in Abschnitt 4.3 angewendete Modellierung der Paketverluste ist sehr einfach gehalten. So wird z.B. von der verwendeten Datenrate abstrahiert und auch ein möglicher Einfluss der Länge der Pakete auf die Verlustwahrscheinlichkeit wird nicht berücksichtigt. Neben der Betrachtung weiterer und mehrerer Fälle, sowie einer besseren Analyse der Videoqualität, sollten daher bei zukünftigen Untersuchungen auch weitere Verlustmodelle betrachtet werden, die die auftretenden Paketverluste realitätsnäher nachbilden.

5.2.3 Allgemeiner Ausblick

Aufgrund der aktuellen und stetig steigenden Relevanz wird in dem Bereich Videostreaming und QoE die Entwicklung weiterer Verfahren und Standards durch die Wissenschaft aber auch durch Unternehmen wie z.B. Netflix erfolgen. Um auch in Zukunft zur Untersuchung neuer Verfahren zur Verbesserung und Sicherstellung der QoE das RST264-Werkzug einsetzen zu können, ist eine zukünftige Weiterentwicklung des Tools aus Sicht des Autors sinnvoll und notwendig.

6 Anhang

6.1 Inhalte des beigelegten Datenträgers

Der beigelegte Datenträger enthält folgende Daten:

- eine Textdatei mit allgemeinen Informationen
- den Quelltext des entwickelten Werkzeugs RST264
- alle im Rahmen der Fallstudien erhobenen Messdaten in Form von csv-Dateien
- pdf-Datei dieser Bachelorarbeit

Alle auf dem Datenträger gespeicherten Inhalte finden sich auch in [2].

6.2 Inhalte des zur Arbeit zugehörigen Repositorys

Neben den auf dem Datenträger gespeicherten Daten enthält das zur Bachelorarbeit gehörige Repository [2] folgende Inhalte:

- alle genutzten bzw. erstellten Ausgangsvideos, sowie der dazugehörige *ffmpeg*-Befehl
- alle für die Fallstudien aufgezeichneten pcap-Dateien, sowie der dazugehörige *ffmpeg*-Befehl zum Erzeugen des dazugehörigen Videostroms
- alle Videos, die unter Einfluss von simulierten Paketverlusten erstellt und später analysiert wurden

6.3 Weitere Abbildungen zur Fallstudie zur Untersuchung der Auswirkung von verschiedenen maximalen RTP-Nutzlastgrößen auf die Qualität des übertragenen Videostroms

Auf den folgenden Seiten sind die Grafiken der Fallstudie aus Abschnitt 4.3 dargestellt, wobei zu jedem Messpunkt ein 95%-Konfidenzintervall dargestellt ist.

Zur besseren Übersicht sind die Grafiken, welche jeweils die gleiche Metrik zum Gegenstand haben auf einer Seite dargestellt.

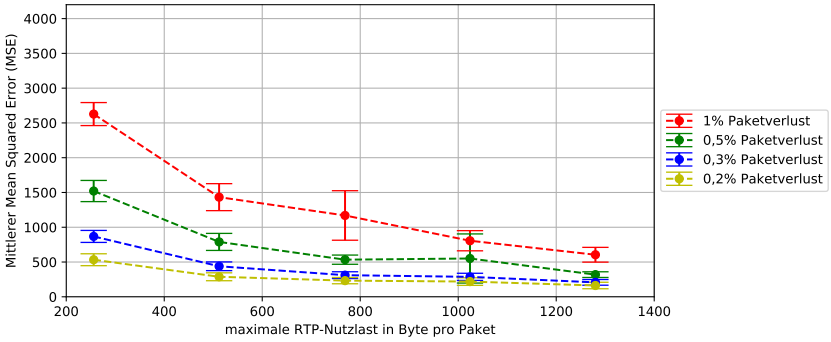

Abbildung 41: MSE unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 5, Werte mit 95%-Konfidenzintervall

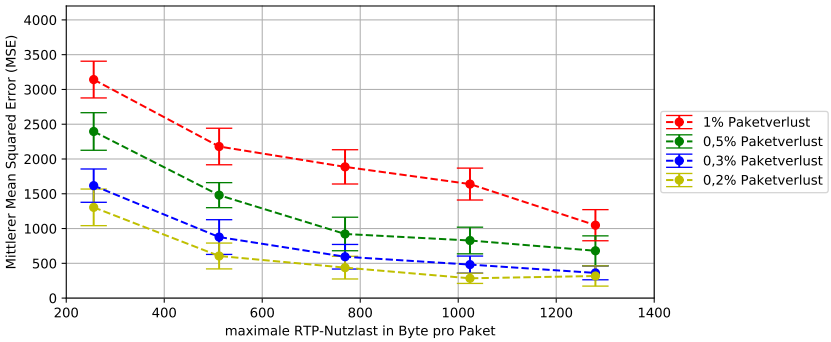

Abbildung 42: MSE unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 20, Werte mit 95%-Konfidenzintervall

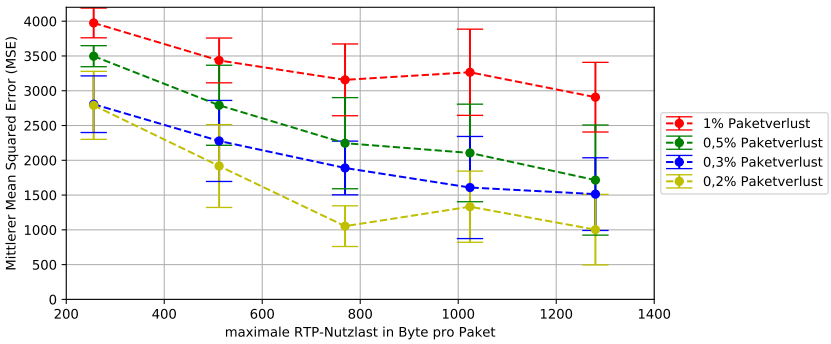

Abbildung 43: MSE unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit dem Standard-IDR-Frameabstand, Werte mit 95%-Konfidenzintervall

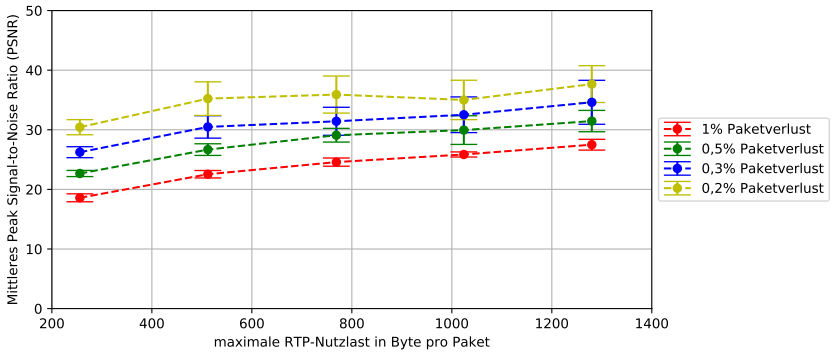

Abbildung 44: PSNR unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 5, Werte mit 95%-Konfidenzintervall

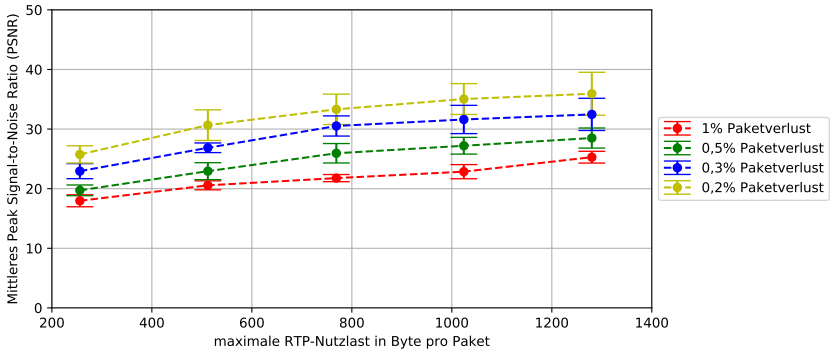

Abbildung 45: PSNR unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 20, Werte mit 95%-Konfidenzintervall

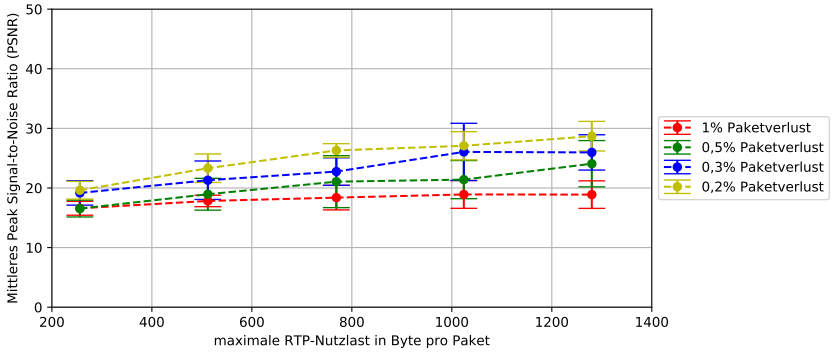

Abbildung 46: PSNR unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit dem Standard-IDR-Frameabstand, Werte mit 95%-Konfidenzintervall

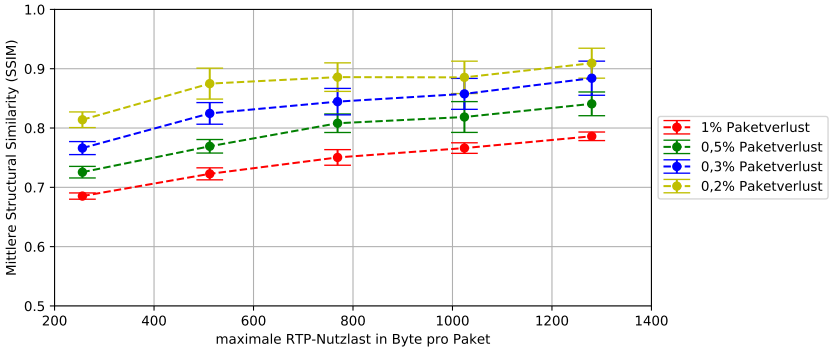

Abbildung 47: SSIM unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 5, Werte mit 95%-Konfidenzintervall

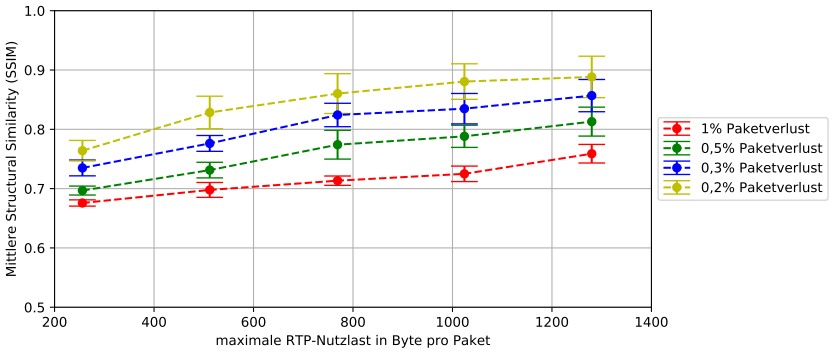

Abbildung 48: SSIM unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit einem maximalen IDR-Frameabstand von 20, Werte mit 95%-Konfidenzintervall

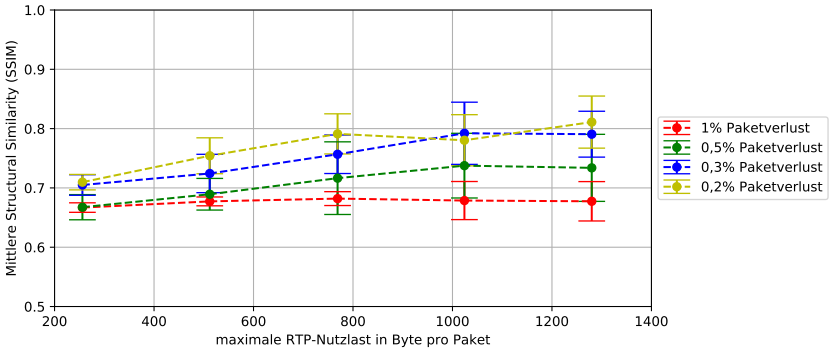

Abbildung 49: SSIM unter Einfluss verschiedener maximaler RTP-Nutzlastgrößen mit dem Standard-IDR-Frameabstand, Werte mit 95%-Konfidenzintervall

Literaturverzeichnis

[1] ffmpeg Documentation. `http://ffmpeg.org/ffmpeg.html`. Abgerufen: 03.09.2017.

[2] Github Repository zur Bachelorarbeit von Nils Straßenburg. `https://github.com/slin96/bachelorthesis_rst264`.

[3] macOS High Sierra. `https://www.apple.com/lae/macos/high-sierra/`. Abgerufen: 11.08.2017.

[4] A. Badach. *Voice over IP: Die Technik*. Hanser, 2010.

[5] M. Burza, J. Kang, and P.D.V van der Stok. Adaptive streaming of mpeg-based audio/video content over wireless networks. *Journal of Multimedia*, 2(2):17–27, 2007.

[6] L. De Cicco, S. Mascolo, and V. Palmisano. Feedback control for adaptive live video streaming. In *Proceedings of the Second Annual ACM Conference on Multimedia systems*, pages 145–156. ACM, 2011.

[7] Cisco. Cisco visual networking index: Forecast and methodology, 2016–2021, Juni 2017. `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf`.

[8] M. C. Q. Farias. Video quality metrics. In *Digital Video*. InTech, 2010.

[9] D. Ferrari. Client requirements for real-time communication services. RFC 1193, RFC Editor, November 1990. `http://www.rfc-editor.org/rfc/rfc1193.txt`.

[10] M. Handley, V. Jacobson, and C. Perkins. Sdp: Session description protocol. RFC 4566, RFC Editor, Juli 2006. `http://www.rfc-editor.org/rfc/rfc4566.txt`.

[11] G. Haßlinger and O. Hohlfeld. The gilbert-elliott model for packet loss in real time services on the internet. In *Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB), 2008 14th GI/ITG Conference-*, pages 1–15. VDE, 2008.

[12] D. Jurca and P. Frossard. Video packet selection and scheduling for multipath streaming. *IEEE Transactions on Multimedia*, 9(3):629–641, 2007.

[13] J. F. Kurose and K. W. Ross. *Computer Networking*. Pearson, 2010.

[14] K. U. R. Laghari and K. Connelly. Toward total quality of experience: A qoe model in a communication ecosystem. *IEEE Communications Magazine*, 50(4), 2012.

[15] Z. Li, A. Aaron, I. Katsavounidis, A. Moorthy, and M. Manohara. Toward A Practical Perceptual Video Quality Metric. `https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652`. Abgerufen: 30.09.2017.

[16] Z.-G. Li and Z.-Y. Zhang. Real-time streaming and robust streaming h. 264/avc video. In *Third International Conference on Image and Graphics (ICIG'04)*, pages 353–356. IEEE, 2004.

[17] T. Rahrer, R. Fiandra, S. Wright, and D. Allan. Triple-play services quality of experience (qoe) requirements. In *DSL Forum TR-126*, volume 2006, 2006.

[18] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. STD 64, RFC Editor, Juli 2003. `http://www.rfc-editor.org/rfc/rfc3550.txt`.

[19] International Telecommunication Union (Telecommunication Standardization Sector). Advanced video coding for generic audiovisual services, Oktober 2016. `https://www.itu.int/rec/recommendation.asp?lang=en&parent=T-REC-H.264-201610-S`.

[20] Y.-K. Wang, R. Even, T. Kristensen, and R. Jesup. Rtp payload format for h.264 video. RFC 6184, RFC Editor, Mai 2011. `http://www.rfc-editor.org/rfc/rfc6184.txt`.

[21] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[22] M. Wien. *High Efficiency Video Coding*. Springer-Verlag, 2015.

[23] S. Winkler and P. Mohandas. The evolution of video quality measurement: From psnr to hybrid metrics. *IEEE Transactions on Broadcasting*, 54(3):660–668, 2008.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Lüneburg, _____ _____

(Nils Straßenburg)