Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Echtzeit Videoübertragung für das iOS Multipeer Connectivity Framework

Bachelor-Thesis

Zur Erlangung des akademischen Grades B.Sc.

Jens Woltering

2117431

Erstprüfer: Prof. Dr. Andreas Plaß
Zweitprüfer: Sven Janzen

Hamburg 31.05.2016

Jens Woltering

Thema der Arbeit

Echtzeit Videoübertragung für das iOS Multipeer Connectivity Framework

Stichworte

Echtzeit, Video, Streaming, App, iOS, Multipeer Connectivity Framework

Kurzzusammenfassung

Im Jahr 2014 veröffentlichte Apple das proprietäre Netzwerkframework *Multipeer Connectivity*. Es bietet Entwicklern von mobilen Anwendungen die Möglichkeit, eine lokale und direkte Peer to Peer Vernetzung aufzubauen. Das so entstehende homogene Netzwerk erreicht, in Abhängigkeit der Funkreichweite, einen hohen Datendurchsatz. Bis dato existiert für diese Art der Kommunikation noch keine wiederverwendbare Lösung zur Videoübertragung. Eine solche Funktionalität würde aber einen großen Mehrwert für zukünftige Anwendungen schaffen.

Ein Toolkit, dass diese Aufgabe übernehmen soll und sich ebenso einfach wie das *Multipeer Connectivity* Framework in neue und bestehende Projekte einbinden lässt, wird in dieser Arbeit konzipiert und umgesetzt. Die Übertragung soll in Echtzeit erfolgen und zur Sicherung der Dienstgüte wird, in Abhängigkeit der verfügbaren Bandbreite, ein adaptives Streaming ermöglicht.

Title oft the paper

Realtime video transfer for the iOS Multipeer Connectivity Framework

Keywords

Realtime, Video, Streaming, App, iOS, Multipeer Connectivity Framwork

Abstract

In 2014 Apple introduced the proprietary network framework *Multipeer Connectivity*. It allows develops to easily create nearby peer to peer networks. These networks with a homogeneous architecture achieve high bandwidths, in dependence of the radio range. But so far there is no reusable solution to offer a video signal to connected peers, for that kind of communication.

Because it would be a remarkable feature for many applications; the topic of the paper is to implement a toolkit based on the Multipeer Connectivity framework, that allows video transfer in realtime. In terms of quality-of-service and to achieve smooth video playback, the toolkit will make use of an adaptive streaming technique. All in all the integration of the toolkit should be as easy as the integration of the *Multipeer Connectivity* Framework itself.

Inhaltsverzeichnis

1 Einleitung

1.1 Thematik

Dienste zur Videoübertragung sind heutzutage nicht mehr aus dem Internet wegzudenken. So existieren unzählige Anbieter die audiovisuelle Inhalte über das Internet zur direkten Wiedergabe bereitstellen. Den wenigsten Nutzern solcher Angebote ist bewusst, welche Komplexität sich hinter Diensten dieser Art befindet. Nutzern von Smartphones mit mobilem Internetzugang haben sicherlich schon öfters registriert, mit welch hohem Datenaufkommen eine Videoübertragung verbunden ist, wenn die Bandbreitenbegrenzung des Mobilfunkvertrags in Kraft tritt. Die Auswirkungen die eine gedrosselte Bandbreite mit sich bringen, erinnern stark an eine Zeit, in denen Haushalte lediglich über eine ISDN-Internetverbindung verfügten. Die Wiedergabe von Videos bei geringer Bandbreite erfolgt unter kaum zumutbarer Videoqualität und deutlich herabgesetzter Auflösung – doch sie ist möglich. Eben jener Mechanismus des Anpassens des Videos ist ein bedeutender Ansatz, um auf die verfügbare Bandbreite automatisch zu reagieren. Andernfalls würde die Wiedergabe immerzu unterbrochen werden müssen, um genügend Videoinformationen nachzuladen.

Letztendlich obliegt es aber der subjektiven Bewertung des Nutzers, welche Art der Reaktion auf die verfügbare Bandbreite angemessen ist. In Form von Untersuchungen zur Quality-of-Experience gibt es komplexe Metriken (Balachandran, et al., 2012, S. 4) bei denen folgende Faktoren eine Rolle spielen. Verzögerungszeit zwischen Anforderung des Videos und Beginn der Wiedergabe, Zwischenladen während der Wiedergabe, Videoqualität (Bitrate) und Häufigkeit der Qualitätswechsel während des Ausspielens. Der Umstand der die Bewertung subjektiv macht ist aber die jeweilige Erwartungshaltung des Nutzers, der das Video bezieht. Bei bezahlten Inhalten erwarten Konsumenten die Ausgabe in hoher Qualität genießen zu können. Dafür wird auch eine entsprechend längere Vorladezeit toleriert. Deutlich weniger Spielraum in puncto Erwartungshaltung gibt es in Bezug auf Live-Streams. Dort wird ein unmittelbares und verzögerungsfreies Abspielen des Videos verlangt. Dabei ist zu beachten, dass sich im Gegensatz zu Video on Demand Diensten kein bereits vorproduziertes und für den Transport optimiertes Videomaterial vorliegt. Um den Echtzeitanforderungen dennoch gerecht zu werden, muss seitens des Anbieters eine performante Videosignalverarbeitung erfolgen.

Eine derartige Rechenleistung war bis vor wenigen Jahren nur in kostspieliger Hardware, die nicht in kompakter Bauform existierte, vorzufinden. Doch durch den stetigen technologischen Fortschritt verfügen mittlerweile selbst Smartphones über ausreichend dimensionierte Rechenleistungen.

1.2 Motivation

Im Rahmen eines Semesterprojekts, das den Namen „Perspective Playground“ trug, wurde eine interaktive Rauminstallation unter Zuhilfenahme mobiler Endgeräte und einer *Virtual Reality Brille* entwickelt. Unter der Annahme, dass der beschriebene technologische

Abbildung 1 Perspective Playground – eine interaktive Virtual Reality Installation

Entwicklungsstand von Smartphones so weit vorangeschritten war, dass eine Videoverarbeitung und Übertragung zwischen zwei Mobilgeräten in Echtzeit möglich ist, wurde unter naiver Vorgehensweise einer App geschaffen die eben dies ermöglichte – wenngleich die bereitgestellte Videoqualität deutlich herabgesetzt wurde und die Übertragung sehr fehleranfällig war. Für den Anwendungszweck, zwei iPhones durch eine kabellose Peer to Peer Verbindung zu koppeln und das Videosignal des einen auf das andere Endgerät zu übertragen, lieferte die Lösung aber unter Idealbedingungen ein akzeptables Ergebnis. Nichtsdestotrotz bestand ab diesem Zeitpunkt die Motivation, eine ausgereiftere Lösung zu entwickeln die eine höhere Bildqualität übertragen kann und diese variabel an die verfügbare Bandbreite anpasst.

Als besondere Herausforderung sind die Echtzeitanforderungen hervorzuheben. In Verbindung mit dem Multipeer Connectivity Framework existiert derweil keine Lösung. Durch eine direkte Peer to Peer Kopplung der Geräte, ohne den weiten Weg über das Internet, sind deutlich geringere Latenzzeiten möglich. Als weitere Herausforderung, die aber

viel mehr als weitere Motivation anzusehen ist, ist die Verwendung des Multipeer Connectivity Frameworks. Es bietet eine derart simple Implementierung zur Kommunikation zwischen zwei oder mehr Endgeräten, die in Verbindung mit einer Videoübertragung und den weiteren im Smartphone integrierten Komponenten wie Sensoren eine Fülle von Anwendungsmöglichkeiten bietet.

1.3 Zielsetzung

Gegenstand der Arbeit ist die Konzeption und Implementierung einer Videoübertragung auf Basis der durch das Multipeer Connectivity Framework bereitgestellten Kommunikationswege. Grundgedanke ist die Implementierung so zu gestalten, dass sich die zu entwickelnde Lösung als Toolkit nutzen lässt. Also die Wiederverwendbarkeit der Vernetzung und der Videoübertragung auch für zukünftige Projekte zur Verfügung zu stellen. Darum soll Wert daraufgelegt werden, die Benutzbarkeit und Übertragbarkeit möglichst einfach zu gestalten.

Aus funktionaler Sicht soll das Toolkit die bestmögliche Videoqualität bereitstellen, die die Bandbreite zwischen den Mobilgeräten ermöglicht und dabei stets den Echtzeitaspekt berücksichtigen. Als Ziel sind weiche Echtzeitanforderungen gesetzt, bei der die Verzögerung zwischen Aufnahme und Ausspielung maximal eine halbe Sekunde betragen sollte. Um dies zu ermöglichen, kann die Videoqualität in Bezug auf die Auflösung, Bitrate und Framerate angepasst werden.

Neben der Implementierung des Toolkits erfolgt, als weiteres Ziel dieser Arbeit, auch die Anfertigung einer Beispielanwendung. Sie soll neben der Funktionsweise auch die Benutz- und Erweiterbarkeit demonstrieren. Wenn das Toolkit sowohl die qualitativen als auch funktionalen Ziele erfüllt, steht ein weitaus höheres Ziel an. Denn die Lösung als solche stellt lediglich einen Mehrwert dar, wenn sie in den richtigen Kontext gebracht wird. Zusammen mit der in den Mobilgeräten zur Verfügung stehenden Peripherie und Sensorik, entsteht ein breites Spektrum an interessante Projekten. Die angestrebte Erweiterbarkeit ließe die Integration von Bilderkennungsalgorithmen oder einer Indoor-Navigation zu. In Kombination mit mehreren Kamerabildern wären komplexe Virtual Reality Anwendungen, in kompakter Bauform, denkbar.

1.4 Gliederung der Arbeit

Nach der thematischen Einordnung im ersten Kapitel, erfolgt die Vermittlung von theoretischen Grundlagen in Kapitel zwei. Dabei werden die maßgeblich bestimmenden Faktoren erörtert, die ein Videosignal charakterisiert und für die Übertragung von Bedeutung sind. Zudem werden unterschiedliche Arten von Streaming und deren Methodik thematisiert, die auf die weitere Entscheidungsfindung und Konzeption im dritten Kapitel Einfluss nehmen. Des Weiteren fließen in diesem Kapitel die Analyse der Frameworks zur Kommunikation, sowie Funktionsweisen bereits existierender Echtzeitprotokolle mit ein. Im viertem Abschnitt erfolgt, Bezug nehmend auf das Konzept und die theoretischen Grundlagen, eine Darstellung der Implementierung. Beschrieben werden dabei alle essenziellen Stellen im Toolkit, die für das Verständnis und die Funktionsweise ausschlaggebend sind. Zur Demonstration der Verwendungsmöglichkeiten des Toolkits, wird im fünften Kapitel die Umsetzung der Beispielanwendung beschrieben. Schlussendlich erfolgt im sechsten Kapitel eine Gegenüberstellung von Zielsetzung und tatsächlich umgesetzter Funktionalitäten.

2 Theoretische Grundlagen

2.1 Kontinuierliche Medien

Kontinuierliche Medien, oder auch zeitbezogenen Medien genannt, sind in ihrer Präsentation an ein Zeitschema gebunden. Dies trifft zum Beispiel auf Video- und Audiosignale zu. Ein Video stellt eine Sequenz von aufeinander folgenden Einzelbildern dar. Das menschliche Gehirn nimmt ab einer Bildfrequenz von 16 Bildern pro Sekunde die Sequenz nicht mehr als Einzelbilder wahr, sondern lässt die Bilder miteinander verschmelzen und es entsteht der Eindruck eines bewegten Bildes. Dabei stellen die genannten 16 Bilder pro Sekunde aber lediglich eine Untergrenze dar, die tatsächliche Anzahl kann durch die individuelle Wahrnehmung variieren. Heutzutage werden Videos in der Regel deshalb mindestens mit einer Bildfrequenz von 24 Bildern die Sekunde ausgespielt, um dem entgegen zu wirken und auch Flimmereffekte zu reduzieren.

Wenn kontinuierliche Medien wie Videos über ein Computernetzwerk übertragen und auf der Empfängerseite ausgespielt werden sollen, muss das dafür vorgesehene Zeitschema eingehalten werden. Da Computernetze vornehmlich paketbasiert aufgebaut sind, stellt dies eine besondere Herausforderung dar. Nicht jedes Datenpaket das durch ein Netzwerk versendet wird, nimmt den gleichen Weg durch das Netzwerk, um sein Ziel zu erreichen – geschweige denn es erreicht das Ziel überhaupt. Um das in Pakete unterteilte Video in seine Ursprungsform zu bringen, hat der Empfänger die zeitlich richtige Reihenfolge wiederherzustellen. Dabei ist obendrein noch mit deutlichen Verzögerungen durch verirrte oder verloren gegangen Pakete zu rechnen, die auf die technisch bedingte Ende-zu-Ende-Verzögerung noch hinzuaddiert werden muss. (Kappes, 2013, S. 298)

Liegen zusätzlich noch Echtzeitanforderungen vor, so muss das Videosignal unverzüglich wiedergegeben werden. Unverzüglich ist in diesem Sinne als sehr kleine Zeitspanne zu verstehen, die toleriert wird, um gegebenenfalls die Reihenfolge der eintreffenden Pakete wiederherzustellen, eher die bis dato eingetroffenen Einzelbilder ausgespielt werden. Für die zu spät eingehenden Pakete bedeutet dies, dass sie verworfen werden müssen. Andernfalls würden Zeitsprünge in der Präsentation der Sequenz erfolgen, die den Eindruck eines bewegten Bildes signifikant stören. Weniger einflussreiche Auswirkungen hingegen hat das Verwerfen der zu spät eintreffenden Informationen, was in Form vereinzelter Bildfehler oder einer geringfügig abweichenden Bildfrequenz kaum auffallen würde.

2.2 Das Multipeer Connectivity Framework

Multipeer Connectivity ist ein von Apple Inc. entwickeltes Netzwerk Framework, das die Vernetzung und Kommunikation unter räumlich naheliegenden iOS Endgeräten ermöglicht. Es unterstützt die Funktechnologien Bluetooth und WLAN. Der Vernetzungsradius ist dabei an die technologiespezifische Funkreichweite gebunden. Die Kommunikation untereinander erfolgt Peer to Peer und bedarf keiner Internetverbindung. Eine weitere Eigenschaft des Multipeer Connectivity Frameworks ist die Multicast Kommunikation zwischen verbundenen Geräten. Darüber hinaus baut das Framework ein vermaschtes Netz auf. Dabei kann ein Endgerät mit einem anderen Endgerät kommunizieren, ohne dass diese direkt

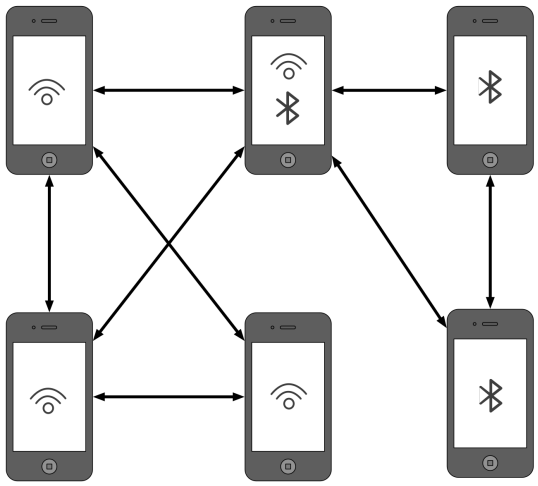

Abbildung 2 Vermaschtes Netz mit unterschiedlichen Schnittstellen

miteinander verbunden sind. Dies ist unter Zuhilfenahme eines dritten Endgeräts möglich, das an dieser Stelle als Vermittler fungiert, und sowohl mit dem Sender als auch Empfänger verbunden ist. Dadurch lässt sich zum Beispiel der Kommunikationsradius erheblich ausweiten. Diese Funktionsweise des Frameworks nutzt zum Beispiel die Instant-Messaging Anwendungen *FireChat* der *OpenGarden Inc.*

Die Einbindung des Frameworks in neue, als auch existierende Projekte ist unkompliziert und lässt sich ohne Drittanbietersoftware nativ umsetzen. In der von Apple zur Verfügung gestellten Developer Library, werden Klassen und Protokolle des Netzwerkframeworks beschrieben. Die explizite Implementierung der bereitgestellten Funktionen zum Verbindungsaufbau und zur Kommunikation ist der Dokumentation allerdings nicht zu entnehmen. Alban Diquet hat im Jahr 2014 auf der *Hack In The Box Security Conference* in Malaysia das Multipeer Connectivity Framework angesichts der angebotenen Verschlüsselung untersucht (Diquet, 2014). Mittels reverse engineering wurden die verwendeten Technologien und beteiligten Protokolle ermittelt, die am Verbindungsaufbau und

dem Transport von Informationen mitwirken. Neben der wichtigsten Erkenntnis, der Offenlegungen einer Sicherheitslücke[1] die das Aushebeln der Verschlüsselung durch eine Man-In-The-Middle Attacke ermöglicht, liefert die Konferenz weitere interessante Aspekte.

Das proprietäre Multipeer Connectivity Framework verwendet intern lediglich eine Fülle standarisierter Netzwerkprotokolle. So findet eine paketbasierende Übermittlung auf Grundlage des Internetprotokolls statt. Für den Verbindungsaufbau erfolgt eine Bekanntmachung über *Bonjour*, ein weiterer Parameteraustausch auf Basis von *TCP* und eine Aushandlung eines gemeinsamen Kommunikationskanals über die Protokolle *STUN* und *ICE*. Für den weiteren Verlauf dieser Arbeit ist aber die Kommunikation nach erfolgreicher Initialisierung der Verbindung von Bedeutung. Der Informationsaustausch zwischen den Endgeräten erfolgt auf Grundlage des *UDP* Protokolls – einem verbindungslosen Transportverfahren. Dies hat den Vorteil, dass der Versand ohne großen Overhead erfolgt und eine schnelle Übermittlung ermöglicht. Im Umkehrschluss bedeutet dies, dass es keinerlei Kontrollmechanismen gibt, die die Verbindung steuern. Insbesondere die Einhaltung der Paketreihenfolge ist nicht zwingend gewährleistet. Im Zuge einer Videoübertragung sind diese Fälle zu berücksichtigen und auszugleichen. Auf Kosten der Transportgeschwindigkeit bietet das Multipeer Connectivity Framework zusätzlich die Versandmethode „Reliable“ an. Laut Dokumentation garantiert diese Option den Versand eines jeden Datenpakets, eine erneute Übermittlung im Fehlerfall sowie die Einhaltung der Sendereihenfolge.

2.3 Videokompression

Die Datenrate, die ein unkomprimiertes Videosignal erzeugt, ist enorm. Zieht man für ein vereinfachtes Rechenbeispiel[2] eine VGA Auflösung von 640 x 480 Pixel, also 307.200 Pixel pro Frame, heran, so ergibt sich mit einer Farbtiefe von 24 Bit und 25 Bilder die Sekunde eine Datenrate von rund 184 Mbit/s. Bei höheren Auflösungen wie zum Beispiel HD 720 oder HD 1080 steigt die Datenrate auf 557 Mbit/s bis 1,2 Gbit/s an. Trotz technologischen Entwicklungen im Consumer Segment, ist eine Übertragung und Vorhaltung von unkomprimierten Videosignalen äußerst ineffizient. Die Datenrate eines Videosignals lässt sich jedoch deutlich reduzieren, wenn man Informationen aus dem Signal entfernt, die redundant oder irrelevant sind. Nahezu jedes Komprimierungsverfahren verfolgt im Kern

[1] Die Sicherheitslücke wurde von Apple daraufhin mit dem Update auf iOS 9 geschlossen
[2] Pixel pro Frame x Farbtiefe x Bilder pro Sekunde

den Ansatz, Irrelevanz- und Redundanzreduktionen, basierend auf unterschiedlichen Algorithmen, durchzuführen. Die so stattfindende Codierung und Decodierung erfolgt mittels eines sogenannten Codecs[3].

2.3.1 Irrelevanzreduktion

Das menschliche Auge nimmt aufgrund seines anatomischen Aufbaus Informationen unterschiedlich ausgeprägt wahr. So ist es in der Lage Helligkeitsunterschiede besser zu differenzieren als Farbunterschiede (Fischer, 2016, S. 134 ff.). Gestützt durch diese und weiteren Erkenntnissen der menschlichen Wahrnehmung, lassen sich Informationen aus dem Videosignal entfernen, die für das Auge keinen erkennbaren Informationsgehalt haben. Mittels solcher Irrelevanzreduktionsverfahren reduziert sich die Datenrate schon spürbar. Jedoch führt das Entfernen von vermeintlich irrelevanten Informationen zu Informationsverlusten und ist ein unumkehrbarer Prozess.

2.3.2 Redundanzreduktion

Eine Redundanz im Zuge der Videokomprimierung liegt vor, wenn ein Videosignal Anteile aufweist, die keine relevanten Informationen beinhalten, da sie zum Beispiel doppelt vorliegen. Aus diesem Grund können diese Segmente gelöscht werden, ohne dass sich der Informationsgehalt und die damit verbundene Qualität mindert (Sack & Meinel, 2009, S. 166). Dies ist problemlos möglich, da sich die eliminierten Informationen mittels Deduktion ableiten und rekonstruieren lassen. Grundsätzlichen lassen sich bei einer Folge von Bildern zwei Arten der Redundanz unterscheiden

2.3.2.1 Räumliche Redundanz

Räumliche Redundanz liegt vor, wenn beispielsweise innerhalb eines Frames benachbarte Pixel gleiche Farbwerte aufweisen. Insbesondere nach der Irrelevanzreduktion, wenn die Farbtiefe auf einen für das menschliche Auge angepassten Farbraum reduziert wurde, findet man viele dieser Bereiche im Frame. Anstatt anschließend die einzelnen Farbinformationen pro Pixel abzuspeichern, ermitteln Algorithmen etwaige Redundanzen und bilden kürzere Beschreibungen für diese Blöcke des Einzelbildes. Bei der H.264 Codierung zum Beispiel, übernimmt die arithmetische Codierung einen Teil der Redundanzreduktion. (Sack & Meinel, 2009, S. 296) Darüber hinaus gibt es eine Vielzahl weiterer Verfahren und Algorithmen zum Entfernen von räumlichen Redundanzen. Diese unterscheiden sich sowohl in ihrer Laufzeit als auch Effektivität. Nicht selten werden mehrerer solcher Verfahren miteinander kombiniert, um das bestmögliche Ergebnis zu erzielen.

[3] Silbenwort aus coder und decoder.

2.3.2.2 Zeitliche Redundanz

Da Videos eine Abfolge von zeitlich versetzten Einzelbildern darstellen, ähneln sich die aufeinanderfolgenden Bilder meist sehr stark. Besonders wenn sich der abgebildete Bildinhalt nicht sehr schnell bewegt, sind die Übereinstimmungen enorm. Darum werden bei modernen Komprimierungsverfahren die jeweiligen Einzelbilder in sogenannte Makroblöcke unterteilt, die zum Beispiel Blockgrößen von 16x16 Pixeln aufweisen. Ausgehend von einem Startbild, dem sogenannten Intraframe *(kurz. I-Frame)*, in dem irrelevante und räumliche Redundanzen entfernt wurden, werden in den darauffolgenden Frames nur die jeweiligen Veränderungen ermittelt. Außerdem handelt es sich bei den Veränderungen oftmals nur um Verschiebungen der Makroblöcke. Die eigentlichen Farbinformationen bleiben erhalten. Diese Verschiebungen lassen sich in Form von Vektoren ausdrücken und erfordern weniger Speicherplatz. Der Anteil der Bewegungsvektoren kann dabei bis zu 40% des komprimierten Videosignals ausmachen. (Schmitz, Kiefer, Maucher, Schulze, & Suchy, 2006, S. 20)

Ein komprimiertes Video besteht am Ende also aus einer Art Bauanleitung um das Ursprungsbild zu rekonstruieren. Zur korrekten Darstellung bedarf es aber folglich, dass auch immer vorangegangene Frames hinzugezogen werden müssen. Um die Abhängigkeiten nicht allzu groß und Folgefehler gering zu halten, werden I-Frames in festen Intervallen erstellt. Die sich darauf beziehenden Einzelbilder inklusive des I-Frames werden in diesem Zusammenhang auch als Group of Pictures bezeichnet. Durch den deutlich höheren Informationsgehalt der Intraframes, muss bei einer Übertragung auch die jeweilige Verzögerung berücksichtigt werden (Kang, Burza, & Stok, 2006).

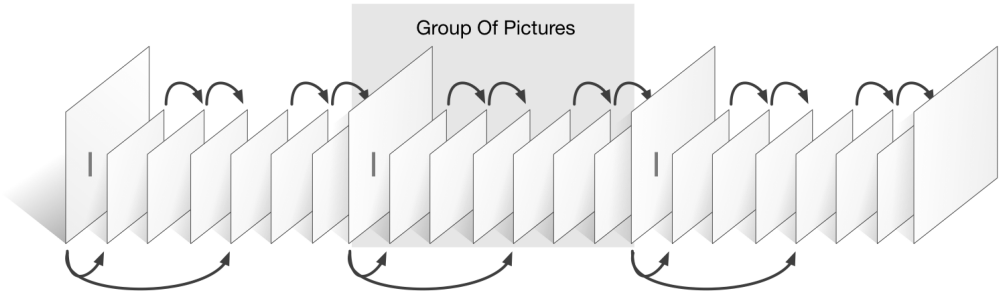

Abbildung 3 Schematische Darstellung einer Gruppe von Bildern

Je nach Codec lässt sich die Genauigkeit und Stärke der Redundanzreduktion einstellen und damit verbunden der Grad der Kompression beeinflussen. Dadurch kann die Datenrate des Videos bis um den Faktor 100:1 verringert werden, ohne signifikante Einbußen hinnehmen zu müssen (DivX, LLC.). Ab einem gewissen Grad ist aber ein deutlicher Qualitätsverlust spürbar, da dadurch nicht mehr nur Redundanzen entfernt, sondern auch bildgebende Informationen zusammengefasst oder entfernt werden.

2.4 Video-Streaming

Das gleichzeitige Übertragen und Abspielen von Medieninhalten von entfernen Quellen wird als Streaming bezeichnet. Diese Technik stellt besondere Anforderungen an das Videosignal als auch an die Übertragung als solche.

2.4.1 Progressiv Download

Die einfachste Variante, die nur bedingt als Streaming bezeichnet werden kann, ist der *progressive download* (Plag & Riempp, 2007, S. 52 ff). Dabei wird eine Videodatei über das *Hypertext Transfer Protocol (HTTP)* zur Verfügung gestellt. Wird mit dem Herunterladen der Datei begonnen, kann in Abhängigkeit der Abspielsoftware, das Video bereits abgespielt werden, bevor der Download abgeschlossen ist. Voraussetzung für diese Art des Abspielens ist, dass die Videodatei in einem entsprechenden Container vorliegt, der dem Abspielprogramm bereits zu Beginn des Downloads *Metadaten* der Datei zur Verfügung stellt. Ein weiterer Aspekt ist die verfügbare Bandbreite, da der Download linear verläuft. Das Video kann nur bis zu der Stelle betrachtet werden, wie es der Fortschritt des Downloads erlaubt. Die Möglichkeit des Spulens, also den Abspielpunkt innerhalb der Videodatei zu verändern, ist ebenfalls vom Fortschritt des Downloads abhängig. Liegt der gewünschte Punkt außerhalb des bereits heruntergeladenen Bereichs, so muss das Video erst bis zu diesem Punkt nachgeladen werden. Ist der Download der gesamten Datei aber erst einmal abgeschlossen, kann das Video in der vorliegenden Qualität auch offline abgespielt werden.

2.4.2 Streamingprotokolle

Ein echtes Streaming erfordert die Kommunikation zwischen Sender und Empfänger, denn im Idealfall sollte die Wiedergabe des Videosignals direkt nach dem Aufruf, egal an welcher Position, beginnen. Dazu soll die Videodatei nicht bis zum jeweiligen Punkt vorgeladen werden müssen. Noch bedeutsamer und zugleich komplexer ist die kontinuierliche Wiedergabe ohne bandbreitenbedingte Unterbrechungen. Infolgedessen wurden zu diesem Zweck Übertragungsprotokolle entwickelt, die die Steuerung des Streams übernehmen. Für den Transport von Streaming-Inhalten, also zum Beispiel eines Videos, wird als Grundlage häufig das auch vom Multipeer Connectivity verwendete User Datagram Protocol (UDP) eingesetzt. Es bietet besondere Vorteile, wenn der Datendurchsatz einen höheren Stellenwert als die Zuverlässigkeit hat (Tanenbaum & Wetherall, 2012).

2.4.2.1 Realtime Transport Protocol

Das Realtime Transport Protocol (RTP) wurde entworfen, um Daten in Echtzeit zu übertragen. Die Übertragung selbst obliegt dabei einem untergeordneten Protokoll. Geeignet dafür und in den meisten Fällen auch verwendet, ist das zuvor erwähnte *User Datagram Protocol*. RTP ergänz das *UDP* Paket um weitere wichtige Informationen, damit Paketverluste festgestellt und auch die Sendereihenfolge wiederhergestellt werden können. Ermöglicht wird dies durch das Anfügen einer fortlaufenden Sequenznummer. Geht ein Paket verloren, kann dies auf der Empfängerseite durch eine lückenhafte Sequenznummerierung festgestellt werden. Das betroffene Paket wird jedoch nicht erneut gesendet. Es besteht aber die Möglichkeit, sich anhäufende Paketverluste zu registrieren, auf die eine Anwendung reagieren kann.

Eine weitere Information, die das Realtime Transmission Protocol dem Datenpaket anfügt, ist ein Zeitstempel zum Zeitpunkt des Versands. Er kann dazu dienen *Jitter*[4] zu ermitteln, doch verfolgt in erster Linie dem Echtzeitgedanken. Eine übermittelte Information, die nach einer gewissen Zeitspanne an Relevanz verliert, kann dadurch ermittelt und bei Bedarf verworfen werden.

2.4.2.2 Realtime Transport Control Protocol

Das *Realtime Transport Control Protocol* (RTCP) dient primär der Kontrolle der Echtzeitübertragung, kann zudem aber zur Steuerung des Streams herangezogen werden. Es übermittelt, anders als das *Realtime Transport Protocol*, keine Medieninhalte. Gleichwohl ist *RTCP* eng mit *RTP* verzahnt und übermittelt, um die Bandbreite zu schonen, nur in periodischen Abständen Verbindungsinformationen der Kommunikationsteilnehmer. Diese Informationen gehen sowohl vom Sender als auch vom Empfänger aus und werden in Form von Reports versandt. Dadurch wird aus einer zuvor unidirektionalen Verbindung des *Realtime Transport Protocols* eine bidirektionale Kommunikation. Die Reports ermöglichen allen beteiligten Kommunikationspartnern Rückschlüsse über die Verbindungsqualität zu ziehen. Die Informationen für die Reports werden aus der Summe der empfangenen und gesendeten Pakete ermittelt. Daraus lassen sich Paketverlustraten und unter Einbeziehung der Zeitstempel, auch geschätzte Übermittlungszeiten berechnen. Wertet man mehrere dieser Reports aus, so lässt sich eine tendenzielle Entwicklung der Verbindung und der zur Verfügung stehenden Bandbreite ermitteln.

[4] Jitter bezeichnet eine zeitliche Schwankung von fixierten Zeitpunkten

Darüber hinaus erlaubt *RTCP* die Synchronisation mehrerer Kommunikationsteilnehmer. Aus der Anzahl der empfangenen Pakete lässt sich eine zeitliche Einordnung in der Abspielposition eines Streams ermitteln. Diese Informationen können dazu herangezogen werden, den Stream zwischen mehreren Empfängern anzugleichen.

2.4.2.3 Realtime Streaming Protocol

Das *Realtime Streaming Protocol* (RTSP) wird zur Steuerung von Streaming Inhalten eingesetzt. Dabei findet ein zuverlässiger Nachrichtenaustausch auf Basis von *TCP* statt, in dem der Client dem Server mitteilt, welche Inhalte angefordert werden. Aus diesem Grund wird das Protokoll auch häufig als Netzwerkfernbedienung bezeichnet. (Schulzrinne, Rao, & Lanphier, 1998, S. 4)

Vom Aufbau und der Syntax ähneln sich *RTSP* und das *Hypertext Transfer Protocol (HTTP)* sehr. Es verfügt über essenzielle Methoden wie *SETUP, DESCRIBE, PLAY* und *PAUSE*, die zusammen mit einer URL an den Server geschickt werden. Doch im Gegensatz zum *Hypertext Transfer Protocol* wird durch das *RTSP* eine zustandsgebundene Sitzung initialisiert. Es besteht während des Abrufens eines Streams also eine bidirektionale Kommunikation zwischen Server und Client. Der Transfer der Medieninhalte, nachdem der Client die *PLAY*-Operation aufgerufen hat, erfolgt dann über das zustandslose *Realtime Protocol (RTP)* mittels *UDP*.

2.4.2.4 Weitere proprietäre Protokolle

Zur Wiedergabe von Streams wurden eine Vielzahl weiterer proprietärer Protokolle entwickelt. Viele von ihnen sind auf besondere Anwendungsfälle zugeschnitten oder bieten erweiternde Kontrollstrukturen an. Vor allem bei kostenpflichtig vertriebenen Inhalten kann ein besonderer Schutz notwendig sein, der eine Authentifizierung und Sicherstellung des Kopierschutzes bedarf. Doch auch wenn es besondere Echtzeitanforderungen gibt, wie beispielsweise einer Liveübertragung, werden besonders angepasste Streamingprotokolle verwendet.

Ein im Internet sehr verbreitetes Protokoll ist das *Realtime Transfer Messaging Protocol (RTMP)*. Es ist ein von Adobe Systems Inc. entwickeltes Protokoll, das nach Aussagen des Herstellers für eine hoch performante Übertragung von Streaming Inhalten im Flashplayer konzipiert wurde (Adobe Systems Inc., 2012). Als Übertragungsgrundlage dient dabei *TCP*, was es in seinem Grundaufbau von RTP unterscheidet. So ist es immer vom jeweiligen Anwendungsfall abhängig, für welches Protokoll man sich entscheidet. Annährend alle existierenden Varianten weisen aber auch Gemeinsamkeiten auf. Neben der Übertragung der Streaming-Inhalte erfolgt stets eine weitere Kommunikation, um die Dienstgüte auf hohem Niveau halten zu können.

3 Konzeption

3.1 Native App Entwicklung

Die zu erstellende Echtzeitvideoübertragung für das Multipeer Connectivity Framework soll eine exklusiv für iOS entwickelte Lösung darstellen. Das Netzwerkframework wurde im Jahr 2014 mit dem Mac OS Update 10.10 ebenfalls auf weitere Plattformen portiert, doch soll die angestrebte Lösung sich zunächst auf den Mobilgerätebereich konzentrieren. Die Umsetzung soll rein mittels der von Apple entwickelten Programmiersprache *Swift* erfolgen, die als Nachfolger der zuvor dominierenden Sprache *Objective-C* anzusehen ist. Bei der Implementierung ist vorgesehen, auf Drittanbieter Software und die Verwendung zusätzlicher Hardware, wie einem dedizierten Server, zu verzichten. Das von Apple bereitgestellte *SDK* beinhaltet im Kern alle benötigten Funktionen und Schnittstellen. Die so zum Teil hardwarenahen Zugriffe, schaffen eine performante Grundlage für die Entwicklung. Zudem arbeiten durch die native Implementierung alle Komponenten bestmöglich zusammen und sind aufeinander abgestimmt. Des Weiteren bringt dies bei der Verarbeitung von großen Datenmengen, was bei Videosignalen zutrifft, große Vorteile.

3.2 Soll-Zustand

Das Toolkit soll auf jedem Endgerät, im folgendem Peer genannt, in der Lage sein, sowohl die Aufgaben des Senders, als auch des Empfängers zu erfüllen. Dadurch würde erreicht werden, die Funktionalitäten innerhalb einer App zu nutzen und zur Laufzeit zwischen Sender und Empfänger wechseln zu können. Das gleichzeitige Senden und Empfangen ist nicht vorgesehen. Ein Empfänger soll jedoch in der Lage sein, zeitgleich die Videosignale mehrerer Sender zu empfangen.

Beim Senden ist vorgesehen stets die besten Videoparameter zu ermitteln, damit eine Übertragung in Echtzeit möglich ist. Zudem soll es möglich sein, selbst Einfluss auf die Videoqualität zu nehmen. Dazu sollen Schranken gesetzt werden können, die die Mindestqualität eingrenzen. In diesem Fall wäre die Echtzeit aber nicht mehr sichergestellt.
Es würde dennoch immer versucht werden, in Abhängigkeit der nicht eingegrenzten Parameter, die bestmöglichen Übertragungseigenschaften zu wählen. Dazu ist es notwendig,

dass die Peers untereinander kommunizieren und ihre Parameter untereinander austauschen.

Die folgende Abbildung stellt mögliche Vernetzungsszenarien dar und verdeutlicht die geplanten Kommunikationskanäle zwischen den Peers. Dabei sollen, wie auch von anderen Streamingprotokollen bekannt, Bild und Zustandsdaten getrennt voneinander übertragen werden.

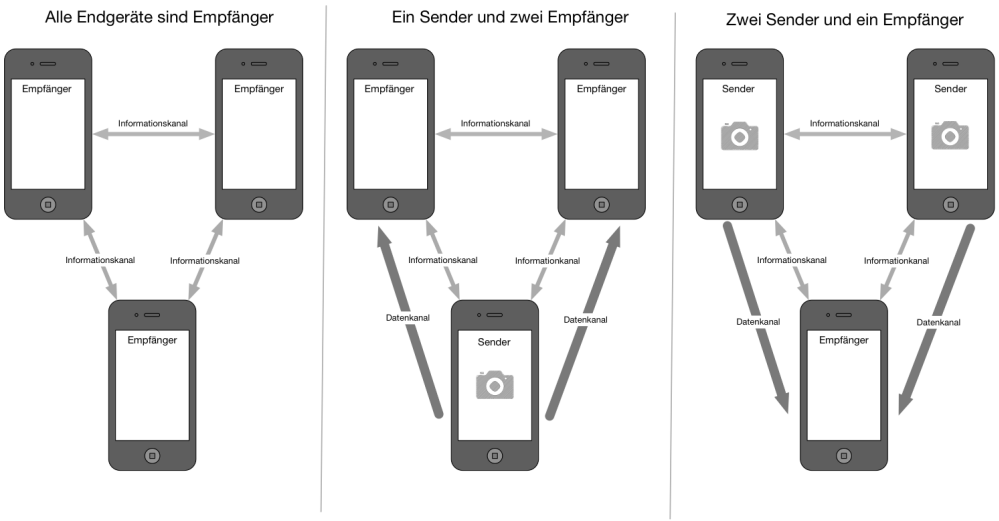

Abbildung 4 Multi- und Unicast Verbindung bei der Videoübertragung

Das Senden des Videosignals erfolgt mittels Multicast an alle vorhandenen Empfänger. Die Verteilung der Zustandsdaten eines jeden Peers, soll über einen weniger priorisierten Informationskanal mittels Broadcast erfolgen. Diese Informationen sollen dabei nur einen Bruchteil der verfügbaren Bandbreite einnehmen.

Folgende Informationen würden über den Informationskanal ausgetauscht werden:

- Peer-Name
- Zustand: Sender oder Empfänger
- gesetzte Schranken: Auflösung, Bitrate und Framerate
- aktuelle Übertragungsparameter: Auflösung, Bitrate und Framerate

Jeder Peer verfügt damit über alle Informationen benachbarter Peers. Eine Aktualisierung dieser Daten erfolgt mit jeder neuen Bekanntmachung, die über den Informationskanal eingeht.

3.3 Schnittstellenanalyse

Um den definierten Soll-Zustand zu erreichen, insbesondere den noch nicht weiter spezifizierten Echtzeitansatz, bedarf es zunächst der Ermittlung der seitens des *SDK* bereitgestellten Funktionen. Herauszufinden gilt, auf welche Weise kommuniziert werden kann und welche Informationen man gewinnen kann, um die Übertragungskapazitäten festzustellen. Des Weitern gilt zu klären, in welcher Form die zu übermittelnden Daten vorliegen müssen und wie diese gegebenenfalls für den Transport aufbereitet werden können.

3.3.1 Multipeer Connectivity Framework

Die Einbindung des Netzwerkframeworks in bestehende als auch neue Projekte gestaltet sich einfach. Für die Nutzung sind keine Netzwerkkenntnisse vorausgesetzt und eine Verbindung ist in wenigen Augenblicken umgesetzt. Das Multipeer Connectivity Framework stellt nach einer etablierten Verbindung drei Funktionen zur Übertragung von Informationen zu Verfügung. Im Folgenden werden die angebotenen Funktionen hinsichtlich ihrer Nutzbarkeit für das Toolkit beschrieben

Funktionsname	**Typ der zu übermittelnden Daten**	**Adressaten**
sendResourceAtURL	NSURL – Verweis auf das Dateisystem oder eine öffentliche Ressource	Einer
startStreamWithName	NSOutputStream – ein Bytestream wird initiiert	Einer
sendData	NSData – eine Byte-Array mit definierter Größe	Mehrere

Tabelle 1 Übersicht der Funktionen zur Kommunikation im Multipeer Connectivity Framework

Das Senden von Ressourcen kommt als Option zur Übertragung nicht infrage, da dafür der Sender das Videosignal zumindest temporär auf den eher langsamen Gerätespeicher ablegen müsste. Diese Art der Übertragung ähnelt dem *progressive download* sehr und entspricht nicht dem Grundgedanken der Echtzeitübertragung. Selbst wenn man die temporären Dateien sehr klein hält oder gar Frame für Frame ablegt, entsteht ein gewaltiger Overhead durch die Speicherung und der Verwaltung auf dem Dateisystem.

Die Verwendung eines Byte-Streams stellt zunächst eine solide Grundlage für die Übertragung der Videosignale dar. Es ermöglicht die kontinuierliche Übertragung von Daten mit unbestimmter Größe. Dabei werden die Daten Byte für Byte in den geöffneten Stream gegeben und sind auf Empfängerseite wieder zusammenzuführen. Dies hat zur Folge, dass die Steuerung und Verwaltung des Streams einen ausschlaggebenden Einfluss auf die Performance und damit auch auf die Echtzeit des Systems haben.

Die Übertragungsfunktion sendData bietet als einzige der drei angebotenen Funktionen die Möglichkeit Daten an einen oder mehrere Empfänger zu senden. Die zu übermittelnden Daten müssen als *NSData* Objekt vorliegen, das einen objektorientierten *Wrapper* für Byte

Buffer darstellt. *Swift* verfügt über ein breites Spektrum an Konstruktoren um *NSData* Objekte zu erzeugen. Eine Vielzahl elementarer Datentypen lassen sich zudem als ein solches Objekt darstellen. Diese Art der Übertragung scheint ideal für den Informationskanal zu sein. Doch auch für die Übertragung des Videosignals besteht großes Potenzial, vorausgesetzt es lässt sich als NSData Objekt darstellen.

Ein hohes Maß an Flexibilität erreicht man durch die Verwendung von Dictionaries als Information. Sie sind eine Form von Collections, die eine Zuordnung von Key- und Valuepaaren vornimmt und eine NSData konforme Struktur bildet. Dadurch lassen sich selbst definierte Pakete schnüren, die sowohl Header- als auch Payloadbereiche aufweisen können. Als beispielhafte Nachricht die mittels der sendData Funktion gesendet werden kann, dient folgender Codeauszug.

```
var message : [String:AnyObject] =
      [
      //defined package-types
      "type":"hello",
      //some data convertible to NSData
      "payload" : "some Data"
    ]
```

Listing 1 Beispielhafte Definition einer Nachricht mittels eines Dictionarys

Da das Multipeer Connectivity Framework an sich als geschlossenes Framework auftritt, ist die Integration des *Realtime protocols (RTP)* nicht möglich. Die dargestellte nachrichtenbasierte Kommunikation bietet aber die Möglichkeit, die Funktionalitäten des Echtzeitprotokolls nachzubilden.

3.3.2 AVFoundation

Das AVFoundation Framework stellt eine Vielzahl von Klassen und Funktionen bereit, um unter iOS audiovisuelle Medieninhalte zu verwalten und abzuspielen. Mithilfe des Frameworks ist ein Zugriff auf die Kameras des iPhone möglich. Es gilt zu überprüfen, ob das Framework eine Ausgabe erzeugen kann, die über das Multipeer Connectivity Framework versandt werden kann und welche Möglichkeiten existieren, die Ausgabe zur Laufzeit an die verfügbare Bandbreite anzupassen.
Letzteres stellt aber keine allzu große Herausforderung dar – zumindest was die Auflösung und die Framerate betrifft. Diese lassen sich während der Nutzung der Kamera ändern. Dazu stellt das Framework vordefinierte Presets bereit.

Presetname
AVCaptureSessionPreset352x288
AVCaptureSessionPreset640x480
AVCaptureSessionPreset1280x720
AVCaptureSessionPreset1920x1080
AVCaptureSessionPresetiFrame960x540
AVCaptureSessionPresetiFrame1280x720

Tabelle 2 Verfügbare Ausgabepresets die das AVFoundation Framework liefert

Was die so definierte Ausgabe der Kamera betrifft, gilt es einen Weg zu finden das Videosignal zu komprimieren, um es anschließend direkt versenden zu können. Dazu gibt es zwei unterschiedliche Ansätze. Es besteht die Option den Output der Kamera, wie bei einer gewöhnlichen Videoaufnahme, in Form einer abspielbaren Datei auf den Gerätespeicher abzulegen. Dabei wird das Videosignal intern mittels H.264 codiert und im MOV-Container abgespeichert. Dies erfolgt aber mit vergleichsweise langsamen Zugriff auf das Dateisystem und das Video müsste in sehr kleine Fragmente zerlegt werden. Diese Variante der Eingangssignalverarbeitung ist für die Echtzeitübertragung nicht die beste Option.

Das AVFoundation Framework bietet aber noch einen erweiterten Zugriff auf das Videosignal. Dabei ist es möglich jeden Frame der Kamera individuell im Rohformat zu verarbeiten. Die so vorliegenden Frames sind aufgrund ihrer unkomprimierten Gestalt aber ebenfalls nicht dazu geeignet sie in Rohform zu übermitteln. Darum muss eine geeignete Kompression angewandt werden, die performant genug ist bis zu 30 Bilder die Sekunde zu komprimieren und eine Ausgabe liefert, die sich über das Multipeer Connectivity Framework übertragen lässt.

3.3.3 VideoToolbox

Im bereits erwähnten ersten Ansatz für ein Semesterprojekt wurden die Rohframes in JPEG-Bilder umgewandelt. Dies erwies sich zum damaligen Zeitpunkt als optimale Lösung, da die verfügbare Bandbreite ausreichte, um JPEG-Frames mit hoher Kompression und einer Auflösung von 480x640 Pixeln zu übertragen. Selbiges Verfahren kommt für die geplante Echtzeitübertragung aber nicht infrage, da für höhere Auflösungen die Kompression deutlich erhöht werden müsste. Der damit verbundene Verlust an Bildinformationen würde spürbar zunehmen. Außerdem ist die Umwandlung der Einzelbilder in das JPEG-Format mit hoher Rechenleistung verbunden. Selbst bei geringer Auflösung nimmt die Umwandlung schon einen Großteil der verfügbaren Rechenkapazität ein, was durch Steigerung der Kompressionsrate und der Auflösung dazu führen würde, dass das Gesamtsystem vollkommen auslastet oder gar überlastet würde. Da die Verwendung von JPEG-Sequenzen zudem eine enorm hohe Datenrate erzeugt und moderne Videocodecs deutlich effizienter arbeiten, wird der Ansatz verfolgt das Videosignal mittels eines dafür geeigneten Codecs zu komprimieren.

Mit der Einführung von iOS 8 stellt Apple ein geeignetes Werkzeug in Form eines weiteren Frameworks bereit, das diese Aufgabe übernimmt. Das low-level Framework *VideoToolbox* erlaubt das hardwarenahe Codieren und Decodieren von Videosignalen und eine individuelle Einflussnahme durch unterschiedliche Parameter. Zu tragen kommt dabei der H.264 Codec. Die hardwarenahe Ausführung macht sich besonders in der Performance bemerkbar. Sie erlaubt es Full HD Material mit einer Framerate von 25 Bilder die Sekunde in Echtzeit zu codieren, ohne dabei die Rechenkapazität des Gesamtsystems zu blocken. Man darf jedoch nicht außer Acht lassen, dass auch bei einer Codierung in Echtzeit eine gewisse Zeit vergeht, eher das Einzelbild tatsächlich komprimiert vorliegt. Als weiterer Punkt kommt hinzu, dass die Datenrate des Videos nun deutlich kleiner geworden ist, die Frames als solche aber noch nicht in der richtigen Form vorliegen, damit sie über das Multipeer Connectivity Framework versendet werden können.

Nach der H.264 Codierung muss jeder Frame noch für die Übertragung aufbereitet und kompatibel gemacht werden, damit er als NSData Objekt versendet werden kann. Dies ist im Übrigen nicht nur bei der Verwendung des Multipeer Connectivity Frameworks der Fall. Auch bei der Paketisierung für andere Protokolle erfolgt eine Fragmentierung des Frames nach einem von der IETF[5] beschriebenem Prinzip (Wenger, Hannuksela, Stockhammer, & Westerlud, 2005). Dabei wird das H.264 codierte Videosignal in sogenannte Network Abstraction Layer Units (kurz NALU) umgewandelt. Dadurch wird das Einzelbild, das ursprünglich nach der Codierung in einem von Apple definierten Objekt vorliegt, in eine universelle Form auf Byte-Ebene gebracht. Diese ist für den plattformunabhängigen Netzwerktransfer geeignet.

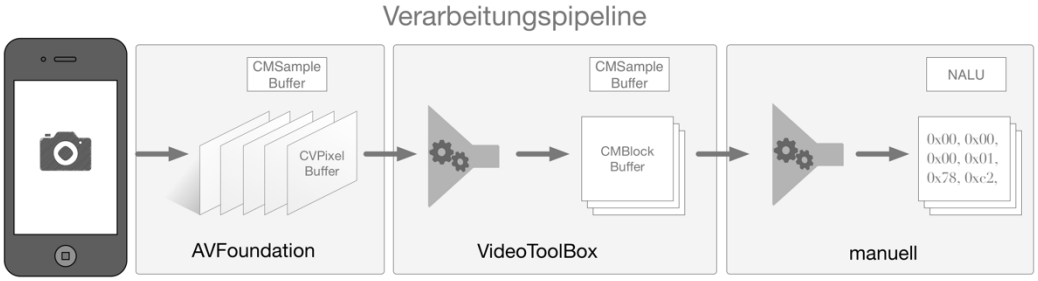

Abbildung 5 Schematische Darstellung der Videosignalverarbeitung

3.4 Echtzeitdatenübertragung

3.4.1 Zeitabhängige Übertragung

Eine direkte Einbindung des Realtime Transfer Protokolls (RTP) kommt nach der Analyse des Multipeer Connectivity Interfaces nicht infrage, da das in sich geschlossene Framework keine Eingriffe in den Paketversand ermöglicht. Die erprobte Funktionalität die das Protokoll jedoch bietet, gilt es auf Basis der vorliegenden Gegebenheiten zu adaptieren. Zentraler Grundgedanke ist, die zu übertragenen Frames die als Nachrichten versendet werden, mit zusätzlichen Informationen zu versehen. Anhand dieser Informationen soll der Empfänger schon beim Eingang der Nachricht in der Lage sein zu erkennen, ob die ihm übermittelten Daten noch relevant sind. Treten vermehrt verspätete Nachrichten auf, die der Empfänger verwerfen muss, besteht die Möglichkeit dem Sender dies durch eine von ihm initialisierte Nachricht zu signalisieren. Auf diesem Weg besteht eine bidirektionale Kommunikation, die im Gegensatz zu TCP nicht jeden Eingang einer Nachricht quittiert, aber im Zweifelsfall Probleme bei der Übertragung anzeigt.

[5] Abkürzung für Internet Engineering Task Force; eine offene internationale Arbeitsgruppe die sich mit der technischen Weiterentwicklung des Internets befasst.

Zeitstempel

Ermöglicht wird dies durch das simple Anfügen eines Zeitstempels an jede Nachricht, die einen Frame enthält. Dazu eignet sich die im Vorfeld erläuterte Form der Nachrichtenstruktur. In einem Dictionary wird neben dem Einzelbild als NALU, auch ein Zeitstempel eingetragen.

```
var message : [String:AnyObject] =
   [
   //defined package-types
   "type":"frame",
   //extracted representation timestamp from the frame
   "timestamp": 1234567890,
   //frame as NALU
   "payload" : NALU
 ]
```

Listing 2 Definition einer Nachricht zur Übertragung von Frames

Hinterlegt wird nicht der Zeitpunkt des Versands, sondern der im Frame eingebettete Zeitstempel, der den Aufnahmezeitpunkt repräsentiert. Obwohl diese Information dann doppelt vorliegt, da sie in der Payload enthalten ist, ist es sinnvoll sie erneut in die Nachricht mit aufzunehmen. Um an den eingebetteten Zeitstempel zu gelangen, müsste zunächst verhältnismäßig rechenintensiv die Frameverarbeitungspipeline in umgekehrter Reihenfolge durchlaufen werden. Diese Schritte sollten, um keine unnötige Performance zu verbrauchen, nur dann durchgeführt werden, wenn der Frame auch tatsächlich ausgespielt werden soll.

Sequenznummern

Eine Verwendung von Sequenznummern für jede Nachricht, wie es im RTP angewandt wird, ist nicht vorgesehen. Dies reduziert, wenngleich auch nur zu einem sehr geringen Anteil, die Bandbreite und ist für den vorgesehenen Anwendungszweck obsolet. Im Realtime Transfer Protocol findet die Sequenznummer unter anderem Anwendung, um die Reihenfolge der Pakete wiederherzustellen. Diese Aufgabe kann aber auch auf Basis des Zeitstempels erfolgen, da dieser in Millisekunden angegeben wird und bei einer Framerate von 25 Bilder die Sekunde keine identischen Zeitstempel pro Peer auftreten. Es gibt ohnehin keinen großen Puffer vor dem Ausspielen der einzelnen Frames und damit auch kaum die Möglichkeit die Reihenfolge der Frames wiederherzustellen. Im Zweifel werden vorzugsweise Frames aufgrund ihres verspäteten Eingangs abgewiesen, anstatt eine künstliche Verzögerung durch Puffer zu erzeugen. Bei sich häufenden Verspätungen erfolgt eine Anpassung der Datenrate, um den Missstand zu beheben.

3.4.2 Kontrollkanal

Durch das Anfügen eines Zeitstempels allein, lassen sich aber noch keine hinzureichenden Informationen über die Dienstgüte ermitteln. Es lässt sich nur, zu einem Zeitpunkt, an dem es eigentlich schon zu spät ist und Störungen und Artefakte bereits aufgetreten sind, feststellen, dass Übertragungsprobleme vorherrschen. Für eine frühzeitige Erkennung von Übertragungsproblemen ist eine kontinuierliche, oder zumindest periodisch stattfindende, Überprüfung notwendig. RTCP ermittelt zu diesem Zweck unter anderem einen Annäherungswert für die Übertragungszeit pro Paket. Eine solche Kennzahl ist ein idealer Ausgangspunkt zu Bewertung der Kommunikation. Ein einzelner Wert ist in diesem Zusammenhang jedoch wenig aufschlussreich, doch die stetige Ermittlung und Gegenüberstellung vorheriger Werte, erlaubt es Tendenzen zu bilden.

Eine genaue Messung der Übertragungszeiten ist jedoch nur schwer zu realisieren. Dazu wäre eine exakte Zeitsynchronisation zwischen Sender und Empfänger notwendig. Nur auf diese Weise könnte unter Zuhilfenahme des Absendezeitstempels die Übertragungszeit berechnet werden. Eine derartige Form der Synchronisierung ist aber sehr umständlich und fehleranfällig. Auch RTCP verzichtet auf diese Vorgehensweise und ermittelt nur einen Schätzwert der Übertragungszeit auf Basis der Summe aus angekommenen Paketen seit dem letzten Report und den eingebetteten Zeitstempeln der jeweiligen Pakete. Eine identische Vorgehensweise im Zusammenspiel mit dem Multipeer Connectivity Framework würde aber unzureichende Ergebnisse liefern, da jedes Einzelbild, beziehungsweise Differenzbild, nur durch ein einziges Paket repräsentiert wird. Die Größe der jeweiligen Pakete kann wegen der angewandten H.264 Codierung deutlich variieren und auch die angestrebte dynamische Anpassung der Framerate kann dazu führen, dass in einem Auswertungsintervall zu wenig Informationen vorliegen um verlässliche Informationen zu gewinnen.

Aus diesem Grund müssen spezielle Nachrichten eigens für diesen Zweck in die Übertragung eingeschleust werden. Diese würden zweifelsohne die Bandbreite belasten, doch durch ihre verhältnismäßig geringe Größe machen sie nur einen marginalen Anteil der Gesamtbandbreite aus. In ihrer Struktur unterscheiden sich diese Pakte von den Frame-Paketen. Sie weisen keine eigentliche Nutzlast auf und haben dadurch eine konstante Größe. Es bleibt aber noch die Problemstellung zur Bestimmung der Zeit, die ein Paket benötigt um vom Sender zum Empfänger zu gelangen. Dieses Problem lässt sich in Bezug auf das Multipeer Connectivity Frameworks aber auf die Bestimmung der Paketumlaufzeit (englisch Round Trip Time; kurz RTT) reduzieren. Also der Zeit die ein Paket benötigt, um vom Sender zum Empfänger und zurück an den Sender zu gelangen.

Diese Bestimmung bietet hinzureichende Ergebnisse, da es sich um eine direkte Peer to Peer Verbindung handelt, in der keine weiteren Netzwerkknoten die Übermittlung beeinflussen oder verzögern. Für den Hin- und Rückweg nimmt die Nachricht also den gleichen Weg.

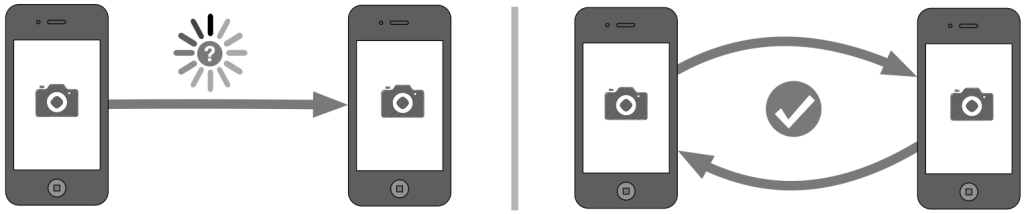

Abbildung 6 Problem der Übertragungszeitermittlung

Eine vom Sender initiierte Nachricht zur Ermittlung der Umlaufzeit muss zu Beginn lediglich mit einem Zeitstempel versehen werden, der die Ausgangzeit darstellt. Wenn das Paket den Empfänger erreicht, muss dieses so schnell wie möglich zurück an den Absender geschickt werden. Der Eingang beim Absender wird dann durch einen weiteren Zeitstempel festgehalten. Die Differenz aus den beiden Zeitstempeln gibt dann die Paketumlaufzeit preis. Da die Verweildauer aufseiten des Empfängers jedoch unbekannt ist und beeinflusst wird durch die Systemauslastung, lässt sich das Verfahren noch optimieren. Der Empfänger quittiert den Eingang der eingetroffenen Nachricht ebenfalls mit einem Zeitstempel und fügt einen weiteren Zeitstempel hinzu, bevor die Nachricht ihn wieder verlässt. Anhand dieser Informationen kann der Initiator der Paketumlaufzeitermittlung, also der ursprüngliche Sender, die Dauer berechnen die das Paket auf der Gegenseite verweilte, ehe es zurückgeschickt wurde. Subtrahiert man diese Zeit von der Gesamtdauer, erhält man eine verhältnismäßig gute Annäherung an die eigentliche Übertragungsdauer aus Hin- und Rückweg.

Nur ein einzelnes Paket zur Bestimmung der Paketumlaufzeit in die Übertragung einfließen zu lassen schont die Bandbreite, kann aber zu fehlerhaften Rückschlüssen führen. Da es sich um eine Funkverbindung zwischen den Endgeräten handelt, treten mit hoher Wahrscheinlichkeit störende Faktoren auf, die zu Paketverlusten und stark variierenden Übertragungszeiten führen. Darum sollte die Umlaufzeit aus einem Mittelwert einer ganzen Reihe von Zeitermittlungen erfolgen. Die so gewonnene Kennzahl stellt dann einen zuverlässigen Status quo der Verbindung dar. Unter Einbeziehung vorheriger Messungen ist dann

auch eine Tendenzbestimmung im Multipeer Connectivity Framework möglich. Abbildung 7 zeigt noch einmal die erforderlichen Kommunikationsschritte, die benötigt werden, um einen Zeitwert für die Übertragung zu ermitteln.

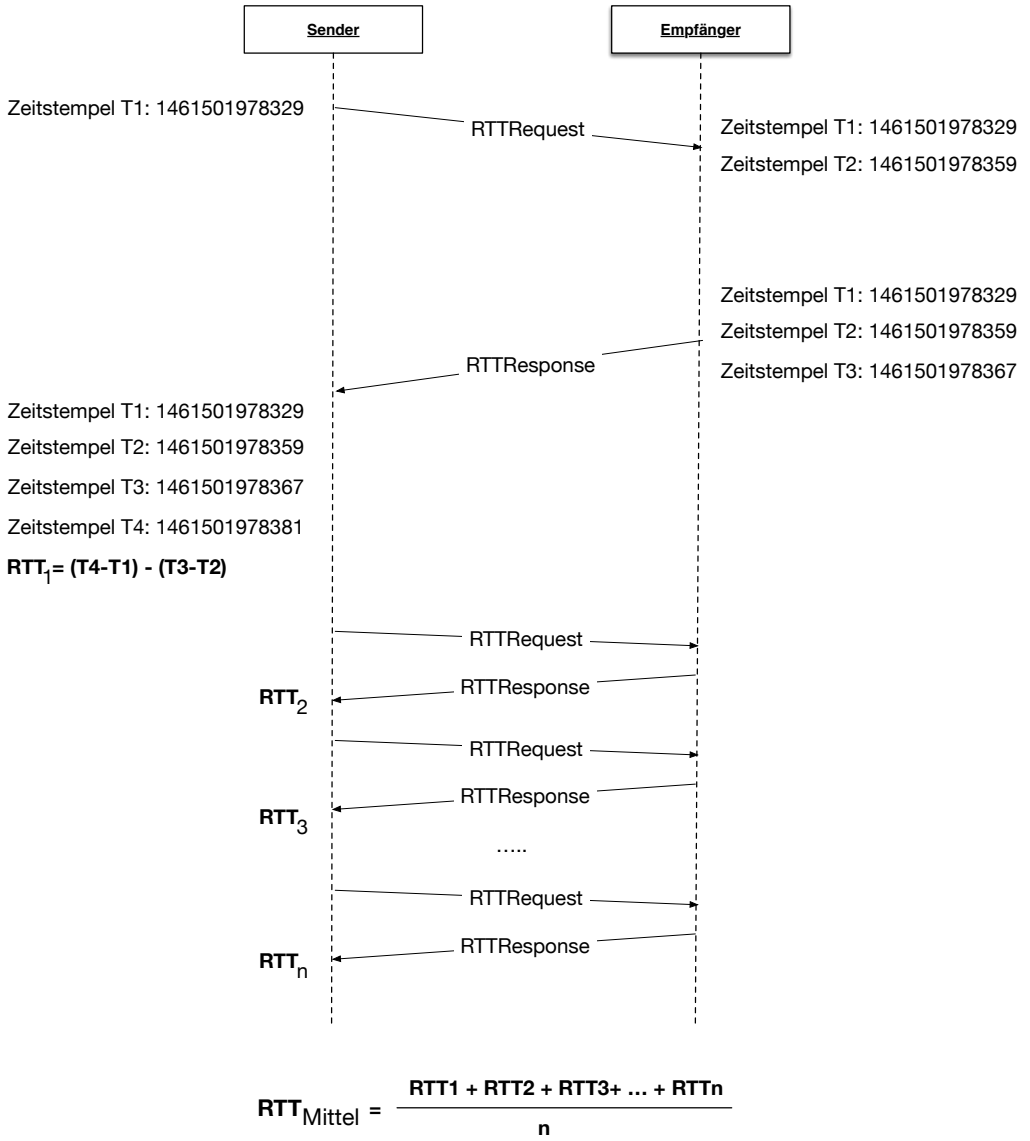

Abbildung 7 Exemplarischer Paketverlauf zur Bestimmung der Umlaufzeit

3.4.3 Steuerungsbefehle

Neben der Übermittlung von Frames und der Überwachung der Verbindungsqualität ist ein weiterer Informationsaustausch erforderlich. Er dient der Steuerung des Streams und der Übermittlung von Zuständen der verbundenen Peers. Die Nachrichten, die dafür versendet werden, sollten auf einer zuverlässigen Übertragungsmethode beruhen, damit sie nicht in der massiven Übertragung der Frames verloren gehen. Außerdem erfolgt dieser Informationsaustausch dabei immer als Broadcast, damit alle verbundenen Peers über den gleichen Kenntnisstand des Netzwerks verfügen.

Die gesamte Übertragung innerhalb des Dienstes lässt sich in drei Phasen unterteilen. Noch bevor die eigentliche Übertragung des Videosignals beginnt, erfolgt in Phase Eins eine Art Handshake, der zur Bekanntmachung der Peers dient. Den Zeitpunkt zur Initialisierung dieses Schrittes wird durch das Multipeer Connectivity Framework bestimmt. Erst wenn die Verknüpfung der Endgeräte erfolgreich durchgeführt wurde und diese untereinander kommunizieren können, kann Phase Eins beginnen. Dabei wird ein Dictionary mit allen Zustandsinformationen und der Kennzeichnung „hello“ an alle verbundenen Peers geschickt. Die Empfänger können anhand dieser Nachricht ermitteln, ob der Peer schon bekannt ist oder intern eine neue Instanz, mit samt übermittelter Daten, zur Datenhaltung angelegt werden muss.

Ist die Bekanntmachung abgeschlossen, befinden sich die Peers in Bereitschaft zur Videoübertragung – der zweiten Phase. In diesem Zustand werden bereits in periodischen Abständen Qualitätsmessungen durchgeführt und die Verbindung aufrechterhalten. Ebenfalls zu diesem Zeitpunkt möglich, ist die Benachrichtigung aller Peers über geänderte Grundeinstellungen wie Peername oder den konfigurierbaren Mindestanforderungen des Videos. Dazu sendet der Peer eine identisch aufgebaute Nachricht wie in Phase Eins. Als Kennzeichnung dient in diesem Fall aber die Bezeichnung „update“. Wechselt der Zustand eines Peers vom Empfänger zum Sender, wird dies in Form einer weiteren Nachricht an alle Peers signalisiert. Sie enthält lediglich die Kennzeichnung „start“ und einen Zeitstempel. Im direkten Anschluss beginnt der Peer mit der Übertragung des Videosignals, an die ihm bekannten Empfänger. Damit ist Phase Drei eingeleitet.

Bei der aktiven Videoübertragung herrscht reger Nachrichtenverkehr zwischen den Peers. Neben der eigentlichen Übertragung der Frames und den Paketen zur Messung der Verbindungsqualität, werden in dieser Phase auch häufig update-Nachrichten versendet. Sie enthalten die aktualisierten Parameter, des sich an die Bandbreite anpassenden Videosignals. Es werden jedoch noch weitere Steuerungsnachrichten benötigt, die der Überlastbehandlung dienen. Durch die eingebetteten Zeitstempel in den Frame-Nachrichten kann der

Empfänger eine Verzögerung in der Übertragung feststellen. Um dies dem Sender mitzuteilen, bedarf es einer Nachricht. Inhalt dieser Nachricht mit der Kennzeichnung „warning“, ist neben einem Zeitstempel ein Wert, der den Grad der Verzögerung angibt. So kann in Abhängigkeit des übermittelten Wertes angemessen auf die Verzögerung reagiert werden.

Letztendlich steht nur noch ein Befehl zum Beenden der Übertragung aus. Diese Nachricht gleicht dem Aufbau der Start-Nachricht, verfügt aber über die Kennzeichnung „stop“.

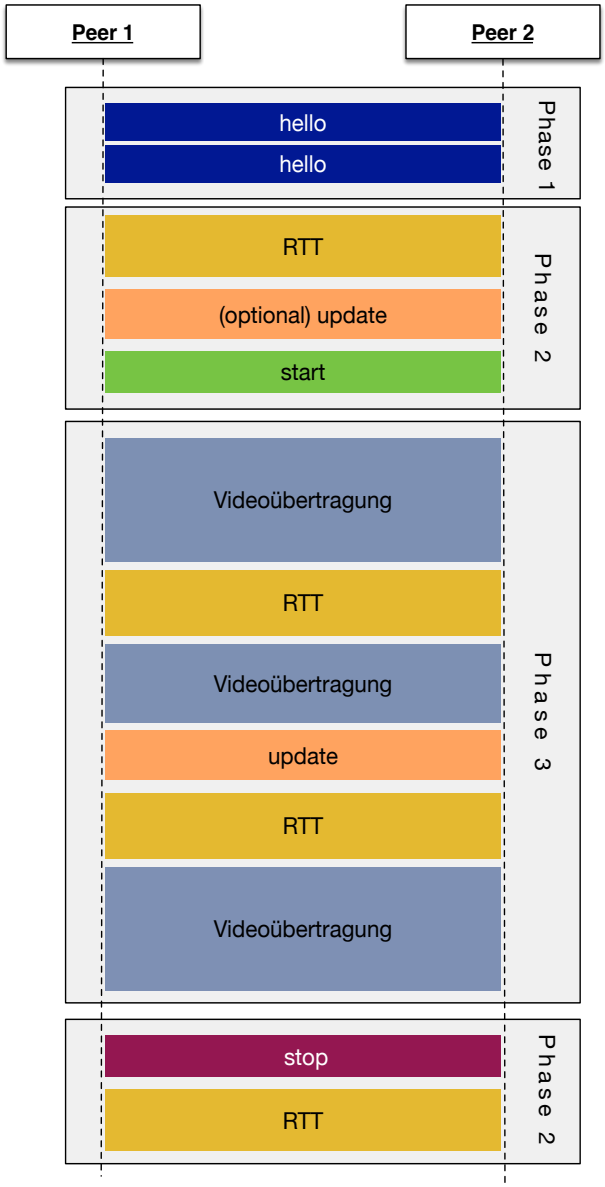

Abbildung 8 Schematischer Ablauf der drei Phasen der Videoübertragung

3.5 Stauvermeidung

Im Verlauf der bisherigen Konzeption und in Teilen der Einleitung wurde die Anpassung des Videosignals an die Bandbreite bereits erwähnt. Zu welchem Zeitpunkt und unter welchen Bedingungen, die noch nicht weiter spezifizierten Anpassungen erfolgen, gilt es festzulegen. Die größte Herausforderung besteht darin, die stetig variierende Bandbreite der Funkverbindung zu berücksichtigen. Dabei ist es nicht notwendig die tatsächlich verfügbare Nettobandbreite mit der benötigten Bandbreite gegenüberzustellen, was ohnehin nicht ohne Weiteres möglich wäre. Viel mehr genügt es die Anzeichen eines drohenden Staus rechtzeitig zu deuten oder im Falle eines Staus angemessen zu reagieren.

Ein eindeutiges Anzeichen für einen bereits vorherrschenden Stau stellt die Tatsache dar, dass Nachrichten den Empfänger wiederkehrend deutlich verspätet erreichen. Diesen Umstand gilt es tunlichst zu vermeiden, denn auf die Echtzeitvideoübertragung bezogen verstößt dies gegen die gesetzten Anforderungen. Auf konzeptioneller Seite ist die Erkennung des verspäteten Eingangs durch die Zeitstempel und die Benachrichtigung mittels Steuerungsnachricht bereits abgedeckt. Da sich ein bereits existierender Stau bei hohen Datenraten mit jedem weiteren Frame vergrößern würde, müssen, sofern sich die verfügbare Bandbreite nicht verbessert, gravierende Maßnahmen getroffen werden um das Problem in den Griff zu bekommen. Aus diesem Grund empfiehlt sich als Reaktion die Verwendung einer Slow-Start[6] Strategie. Sobald der Sender über das Problem informiert wurde, setzt er das Videosignal herab auf das Minimum. Dadurch wird eine sofortige Reduzierung der Datenrate erreicht, die der Kommunikation die Möglichkeit bietet, sich neu einzupendeln und den Echtzeitanforderungen wieder gerecht zu werden.

Da die Erkennung zu spät eingehender Frames nur dann erfolgen kann, wenn es im Grunde genommen schon zu spät ist, erfolgt durch den Kontrollkanal eine frühzeitige Erkennung. Die dabei vorgesehene kontinuierliche Überprüfung der Paketumlaufzeiten erlaubt es Rückschlüsse zu drohenden Staus zu ziehen, noch bevor eine Überlastung auftritt. Die Betrachtung vorangegangener Messungen der Umlaufzeiten gibt preis, ob eine tendenzielle Verbesserung oder Verschlechterung der Verbindung vorliegt. Auf solche Tendenzen lassen sich präventiv die Parameter des Streams gemäßigt heruntersetzen. Infrage kommt beispielweise die sukzessive Reduzierung der Bitrate oder der Auflösung. Weitere Aufschlüsse bietet der direkte Vergleich der aktuellen Messung mit dem Durchschnitt vergangener Umlaufzeiten.

[6] In Anlehnung an die Strategie, die bei der TCP Überlastkontrolle angewandt wird

Schlussendlich besteht noch eine weitere, bisher konzeptionell nicht erfasste Variante der Stauvermeidung. Durch die Einführung einer in ihrer Größe beschränkten Warteschlange aufseiten des Senders, werden zu sendende Nachrichten nach dem FIFO[7]-Prinzip versandt. Erst wenn eine Nachricht den Sender verlassen hat, wird ein Platz in der Warteschlange frei. Wenn kontinuierlich mehr Nachrichten zum Versand eingereiht als tatsächlich abgesendet werden können, so würde sich die Warteschlange ins unermessliche verlängern, wenn diese nicht begrenzt wäre. Nach dem in der Abbildung gezeigten Leaky-Bucket Prinzip lässt sich ein Überlauf erfassen, der als frühzeitiges Anzeichen eines drohenden Staus gedeutet werden kann.

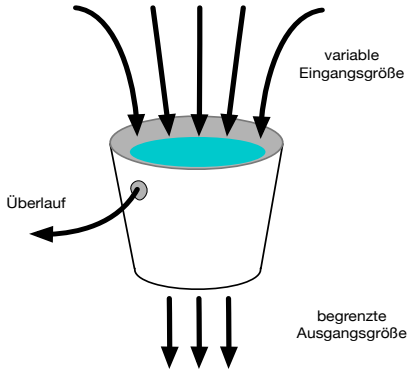

Abbildung 9 Schematische Darstellung des Leaky Bucket Prinzips

Sollte es im Verlauf der Übertragung zu einem Überlauf kommen, dient dies als zuverlässiges Anzeichen dafür, dass das Videosignal in der derzeitigen Form die verfügbare Bandbreite überschreitet. Ein weiterer Vorteil dieser Methode ist, dass bei überfüllter Warteschleife neue Nachrichten verworfen werden. So baut sich keine konstante Verzögerung auf, die die weitere Übertragung negativ beeinflussen würde. Da durch diesen Zustand aber Frames verloren gehen, ist schnelle Reaktion durch eine Anpassung des Videosignals notwendig.

Im Umkehrschluss steht noch die Erkennung positiver Entwicklungen aus. Diese lässt sich aus den negativen Entwicklungen ableiten. Denn mit jedem Prüfintervall, in dem keine qualitativ zu beanstanden Merkmale ausfindig gemacht werden können, steigt die Wahrscheinlichkeit einer positiven Entwicklung. Die Anzahl dieser positiven Überprüfungen wird in Form eines Zählers festgehalten. Zusammen mit der Tendenzbestimmung der Paketumlaufzeit kann so eine gestaffelte Steigerung der Qualität des Videosignals vollzogen werden. Bei jedem erkannten negativen Einfluss wird der Zähler wieder zurückgesetzt. Durch dieses Verfahren soll erreicht werden, dass die Qualität möglichst schnell ansteigt, aber sich beim Erreichen einer kritischen Grenze in der Waage hält.

[7] First in – First out; Eine Abarbeitung die der Reihe nach geschieht.

4 Umsetzung

4.1 Systemstruktur

Im folgenden Abschnitt wird der Aufbau des zu entwickelnden Toolkits zur Echtzeitvideoübertragung, mit allen bisher beschriebenen Funktionalitäten, gesamtheitlich erläutert. Unter Berücksichtigung der im ersten Kapitel beschriebenen Ziele, ein erweiterbares und in der Benutzbarkeit einfach zu implementierendes Toolkit zu schaffen, ist die Strukturierung der Software von besonderer Bedeutung. Betrachtet man die im Verlaufe des Konzepts erläuterten Maßnahmen zur Bereitstellung und Sicherung der Echtzeitübertragung, so lässt sich folgende schematisch vereinfachte Darstellung für das Toolkit aufstellen.

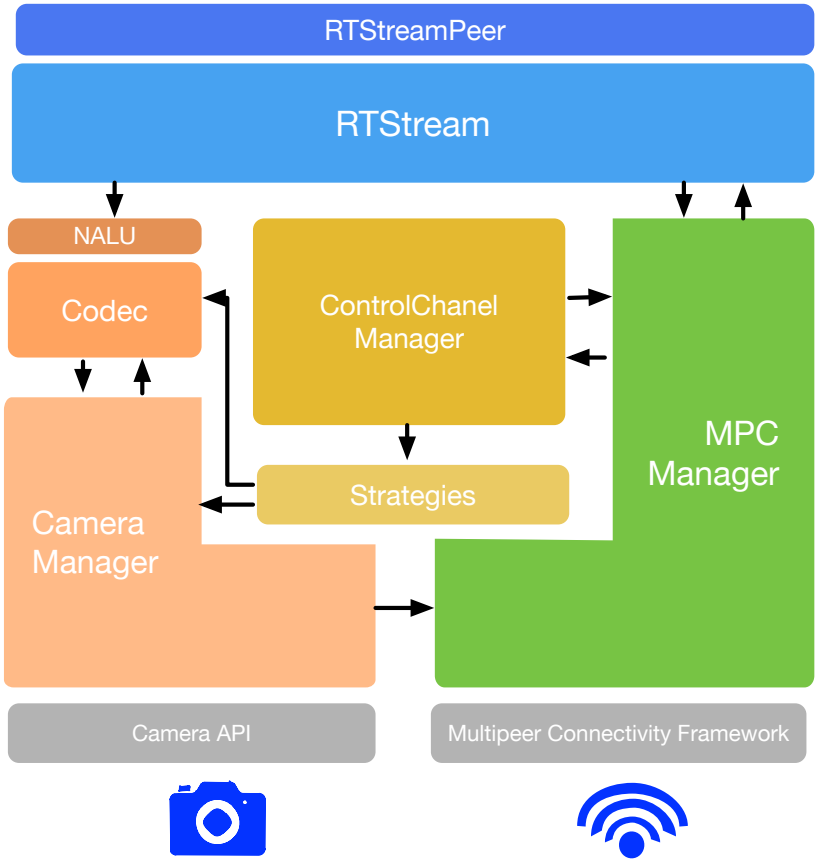

Abbildung 10 Vereinfachte Darstellung der Softwarearchitektur

In Anlehnung an das Single-Responsibility-Prinzip werden für die Bestandteile des Toolkits, die einen gemeinsamen Zweck dienen einzelne Klassen gebildet. Demzufolge werden bei der Umsetzung folgende Klassen mit den beschriebenen Aufgabenbereichen zu implementieren sein:

MPCManger: Die MPCManager Klasse implementiert das Multipeer Connectivity Framework. Sie stellt damit die Grundlage der Kommunikation dar und reagiert auf die Zustände des Netzwerkframeworks. Sämtliche Nachrichten werden über diese Klasse verschickt.

CameraManager: Die CameraManager Klasse repräsentiert die direkte Schnittstelle zur Kamera des iPhones. Sie initialisiert alle erforderlichen Schritte um auf die Hardware zuzugreifen und liefert das Videosignal in Rohformat. Sie ermöglicht ebenfalls die Parameter der Ausgabe zur Laufzeit zu verändern.

Codec: Die Codec-Klasse stellt die Implementierung des Frameworks dar, dass der Komprimierung des Videosignals dient. Auch die Bitrate soll sich zur Laufzeit ändern lassen können.

NALU: Die NALU-Klasse stellt Funktionen bereit, um das komprimierte Videosignal in eine universelle Form zu bringen, damit es über das Netzwerk übertragen werden kann. Auf der anderen Seite ist es in der Lage Daten aus dieser Form zurück, in einen für die Programmiersprache Swift geeigneten Datentyp, zu transferieren.

ControlChanelManager: Als internes Herzstück dient die ControlChanelManager Klasse. Sie setzt die Echtzeitanforderungen um und kontrolliert den Nachrichtenaustausch. Übertragungsproblemen werden an dieser Stelle erfasst und Reaktionen eingeleitet.

Strategies: In der Strategies Klasse werden die Verhaltensmuster gebündelt, auf die der ControlChanelManager zur Anpassung des Videosignals zurückgreifen kann. Die darin definierten Strategien greifen auf den Codec als auch auf den CameraManager zu.

RTStream: Diese Klasse ist die zentrale Komponente und repräsentiert die Echtzeitübertragung nach außen. Sie dient der Datenkapselung und initialisiert alle benötigten Instanzen. Durch einen kontrollierten Zugriff auf die von ihr erzeugten Instanzen und deren Funktion wird erreicht, dass keine unvorhergesehenen Zustände während der Übertragung auftreten.

RTStreamPeer: Alle Informationen und Zustände der vernetzten Peers werden als Attribute in dieser Klasse festgehalten. Sie dient der Datenhaltung und wird unter anderem vom ControlChanelManager zur Ermittlung von Übertragungskennzahlen hinzugezogen.

4.2 Entwurfsmuster

Um die Erweiterbarkeit zu wahren und bekannten Herausforderungen der Softwareentwicklung auf bereits erprobte Modelle zu reduzieren, werden Entwurfsmuster für das Toolkit angewandt. Sie erleichtern nicht nur das Verständnis, sondern dienen maßgeblich der Funktionalität.

4.2.1 Singleton

Das Singleton (dt. Einzelstück) Entwurfsmuster zählt zu den sogenannten Erzeugermustern. Es zielt darauf ab, dass nur eine einzige Instanz einer Klasse gebildet werden soll. Für die zentrale Klasse RTStream bietet sich diese Art der Erzeugung besonders an, denn das Toolkit sollte auf einem Endgerät nur ein einziges Mal initialisiert werden. Würden mehrere Instanzen existieren, so würde der Zugriff auf physikalische Hardware, wie zum Beispiel die Kamera, blockiert werden. Die Realisierung dieses Erzeugermuster in Swift ist simpel. Wie abgebildet, muss eine Klassenvariable als *static* gekennzeichnet werden und ihr eine Instanz der eigenen Klasse zugeordnet werden. In Kombination mit dem Schlüsselwort *let* kann die so definierte Variable nur ein einziges Mal gesetzt werden. Um zudem sicherzustellen, dass an keiner weiteren Stelle eine Instanz der Klasse erzeugt werden kann, sollte der Konstruktor als privat gekennzeichnet werden.

```
static let sharedInstance = RTStream(serviceType: "rt-Video")
private init(serviceType:String){...}
```

Listing 3 Implementierung des Singleton Entwurfsmusters

4.2.2 Delegate

Das Delegate-Entwurfsmuster findet in vielen von Apple entwickelten Frameworks weite Verbreitung. Es ermöglicht die Zuständigkeit von Aufgaben auf andere Objekte zu verteilen (delegieren). Dazu wird ein protocol definiert, das in anderen Programmiersprachen einem Interface gleicht und zur Verteilung der Aufgabe aufgerufen wird. Ein Objekt, das die im *protocol* deklarierte Funktion implementiert und sich zur Ausführung registriert hat, übernimmt anschließend die delegierte Aufgabe.

Eine Einteilung dieses Musters in eine bestimmte Kategorie von Entwurfsmustern nach der Gang of Four[8] ist schwierig. In gewisser Hinsicht ähnelt es dem *Decorator* Strukturmuster. Oftmals wird es aber auch als Verhaltensmuster verwendet, mit dem Ziel ein registriertes Objekt zu benachrichtigen, ohne es explizit zu kennen. Im Gegensatz zum *observer pattern* erlaubt das delegate Entwurfsmuster aber nur eine 1:1 Beziehung und ist damit ungeeignet, eine Gruppe von Empfängern zu benachrichtigen. In puncto Performance schneidet das delegate-Muster bei der Benachrichtigung eines einzelnen Empfängers besser ab, als das observer pattern (Harrison, 2010). Aus diesem Grund ist es auch erste Wahl bei der Verwendung innerhalb der Bildverarbeitungspipeline. Es ermöglicht auch das Zwischenschalten oder Austauschen von weiteren Verarbeitungsschritten, ohne die Gesamtkette zu unterbrechen.

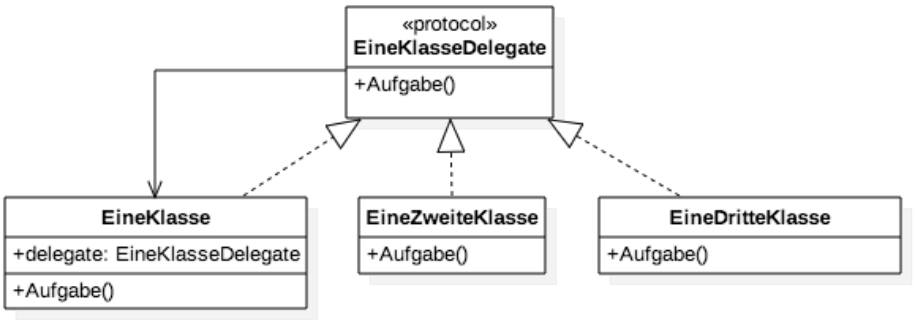

Abbildung 11 Klassendiagramm zur Veranschaulichung des delegate patterns

4.2.3 MVC

Das Model-View-Controller Prinzip ist in der Entwicklung von mobilen Anwendungen eines der am häufigsten verwendeten Architekturmuster. Es beschreibt die fundamentale Trennung zwischen Daten (Model), Darstellung (View) und Logik (Controller). Auch bei der Entwicklung des Toolkits wird auf diese Trennung geachtet, damit es sich bestmöglich in andere Projekte implementieren lässt. Im Grunde genommen ist diese Aufteilung simpel, da es über keine darstellenden Komponenten verfügt. Für die Datenhaltung ist die Klasse *RTStreamPeer* vorgesehen. Der Rest kann als Logik angesehen werden.

4.3 Verbindungsaufbau

Bevor der im Konzept genannte Handshake zur Bekanntmachung der Peers im Netzwerk stattfinden kann, muss die Verbindung auf Ebene des Multipeer Connectivity Frameworks erfolgen. Dazu wird im *MCPManager* das Multipeer Framework implementiert. Um

[8] Gang of Four steht als Synonym für die vier Autoren, die das Buch „Design Patterns. Elements of Reusable Object-Oriented Software“ verfasst haben

den Verbindungsaufbau so einfach wie möglich zu gestalten, wird dieser weitestgehend autonom durchgeführt. In vielen Anwendungen, die das Multipeer Connectivity Framework ebenfalls verwenden, ist eine Interaktion zum Verbindungsaufbau notwendig. Dabei werden Verbindungsteilnehmer explizit aufgefordert in das Netzwerk einzutreten. Dieser Mechanismus wird im Toolkit aber durch eine programmatische Einladungsverwaltung umgangen. Über den erfolgreichen Verbindungsaufbau auf Ebene des Multipeer Connectivity Framework wird man über eine delegierte Funktion informiert. Zustandsbasiert können anschließend eigene Implementierungen erfolgen.

```
func session(session: MCSession, peer peerID: MCPeerID,
             didChangeState state: MCSessionState)
{
    switch state {
    case .NotConnected:
        print("not connected")
    case .Connected:
        if session.connectedPeers.contains(peerID){
            //done here, delegate to next responsible class
            self.delegate?.connectedDevicesChanged(self,
             connectedDevice:peerID, didChangeState: state.rawValue)
        }
    default:
         print("connecting")
    }
}
```

Listing 4 Statusbasierte Reaktion auf die Veränderung der MPC-Sitzung

Um auch hier die Grundsätze des Single-Responsibility-Prinzips zu wahren, endet nach dem Verbindungsaufbau der Aufgabenbereich des MPCManager zunächst. Die Informationen über den Verbindungsstatus und die jeweilige Kennzeichnung des Peers werden anschließend an die nächste Instanz weitergeleitet. Dies ist in diesem Fall der ControlChanelManager, der den Handshake einleiten wird.

Der Handshake erfolgt über den im Konzept erörterten Weg des Nachrichtenversands. Die Adressierung der Nachricht geschieht unter Zuhilfenahme der übermittelten *PeerID*. Inhalt der als *hello* gekennzeichneten Nachricht sind die Attribute der eigenen *RTStreamPeer* Instanz sowie der Status, ob das Gerät als Sender oder Empfänger fungiert. Der Empfänger der *hello*-Nachricht kann aus den übermittelten Daten alle benötigten Informationen beziehen, um für den neuen Verbindungsteilnehmer eine weitere *RTStreamPeer* Instanz zu erzeugen, auf die er jederzeit zugreifen kann. Mit der erfolgreichen Instanziierung und dem Anfügen an ein Array zur Verwaltung der verbundenen Peers ist der Verbindungsaufbau abgeschlossen. Durch die hinterlegte PeerID können auch zukünftig Nachricht an den jeweiligen Peer adressiert werden.

4.4 Datenhaltung

Das Vorhalten aller für die Übertragung relevanter Daten erfolgt wie konzeptionell festgehalten über Instanzen, die die verbundenen Peers repräsentieren. Sie beinhalten aber nicht nur die Daten die aus dem Handshake hervorgehen, sondern auch eine ganze Reihe weiterer Informationen, die der ControlChanelManager ermittelt. Damit ein kontrollierter Zugriff auf die Daten möglich ist, bedarf es einer Struktur. Nachfolgend beispielhaft dargestellt sind die Zugriffsmöglichkeiten auf die in einer RTStreamPeer Instanz hinterlegten Paketumlaufzeiten.

```
var roundTripTimeHistory :Double{
        get {
            //calculation arithmetic mean of stored RTT
            return _roundTripTimeHistory.reduce(0) { $0 + $1 } /
                    Double(_roundTripTimeHistory.count)
        }
        set(aNewvalue){
            _roundTripTimeHistory.append(aNewvalue)
        }
    }

private var _roundTripTimeHistory :[Double] = [0] {
       didSet{
           if self.initialzed == true{
               if _roundTripTimeHistory.count == 10 {
                   _roundTripTimeHistory.removeAtIndex(0)
               }
           }
       }
   }
```

Listing 5 Zugriff auf Instanzvariablen nach dem backing store Prinzip

Es werden für die Datenhaltung pro Merkmal stets zwei Variablen verwendet. Eine ist nach außen öffentlich und ermöglicht das kontrollierte Lesen und Schreiben. Eine weitere ist privat und von außen nicht zugänglich. Dieses, als *backing store* bezeichnete, Prinzip wird für alle Attribute der *RTStreamPeer* Klasse beibehalten. Der Vorteil und die Verwendung dieses Prinzips beinhaltet verschiedene Aspekte. Zum einen kann die Komplexität von berechneten Werten nach außen verborgen bleiben, da die öffentliche Variable eine simple Darstellung verkörpert. In diesem Beispiel gibt die öffentlich Variable die gemittelte Paketumlaufzeit an, obwohl beim Setzen nur eine einzige Zahl anzugeben ist. Des Weiteren lässt sich in Swift das Setzen von Variablen auf diese Art beobachten. Durch die *didSet* Methode lässt sich der Wert einer Variable kontrollieren oder wie in diesem Beispiel angewandt, die Anzahl der für die Mittelwertberechnung herangezogenen Werte begrenzen.

Für andere Attribute der *RTStreamPeer* Klasse lässt sich durch dieses Prinzip zum Beispiel auch überprüfen, ob sich ein zu setzender Wert von einem bereits vorhandenem unterscheidet. Wenn dies nicht der Fall ist wird der Wert verworfen, andernfalls wird er neu gesetzt.

Der neue Wert löst dann durch die *didSet* Methode, eine Benachrichtigung aller anderen Peers in Form einer Update-Nachricht aus.

4.5 Videoverarbeitungspipeline

Im folgenden Abschnitt wird die Implementierung der im Konzept als Videoverarbeitungspipeline bezeichneten Schritte beschrieben. Die Pipeline beginnt mit dem Zugriff auf die Kamera, um das unkomprimierte Videosignal zu beziehen und endet nach der Umformung des codierten Einzelbilds in ein für die Übertragung aufbereitetes Paket. Da alle Schritte mitunter dreißigmal die Sekunde ausgeführt werden, ist in allen an der Pipeline beteiligten Prozessen auf die Laufzeit zu achten.

4.5.1 Erweiterter Zugriff auf die Kamera

Der CameraManager implementiert alle für den Kamerazugriff notwendigen Funktionen des *AVFoundation* Frameworks. Dazu wird eine *AVCaptureSession* initialisiert, die maßgeblich die Verwendung der Kamera steuert. Dieser Session wird bei der Einrichtung mitgeteilt, welche der im iPhone verbauten Kameras als Input dienen soll. Ebenfalls wird ein Preset übergeben, dass die Auflösung der Ausgabe bestimmt. Wie im Konzept beschrieben, liegen unterschiedliche Presets vor, die zur Laufzeit geändert werden sollen. Neben der Auflösung soll aber auch die Framerate steuerbar sein. Da diese aber in der Regel fest an das Preset gekoppelt ist, ist ein benutzerspezifisches Eingreifen notwendig. Das AVFoundation Framework stellt dafür aber die Möglichkeit parat, die obere und untere Schranke der Belichtungszeit frei zu wählen. Setzt man beide Begrenzungen auf den gleichen Zeitwert, so lässt sich eine fixe Framerate definieren. Diese Parameter lassen sich, wie das Setzen der Presets, auch während der laufenden Kamera ändern.

```
func setFramerate(fps :Int){
        dispatch_async(sessionQueue, {
        do{
            try self.videoDeviceIn.device.lockForConfiguration()
            self.videoDeviceIn.device.activeVideoMinFrameDuration =
              CMTimeMake(1,Int32(fps))
            self.videoDeviceIn.device.activeVideoMaxFrameDuration =
              CMTimeMake(1,Int32(fps))
            self.videoDeviceIn.device.unlockForConfiguration()
        }catch{
        }
        })
    }
```

Listing 6 Funktion zur individuellen Anpassung der Framerate

Über die Konfiguration des Eingangs hinaus wird ein Output der Session definiert. Dabei kann bestimmt werden, in welchem Pixelformat die Ausgabe des Videosignals erfolgen soll. Zur Auswahl stehen allerlei Konstanten des *CoreVideo* Frameworks, die die Unterscheidung zwischen verschiedenen Farbräumen und Abtastraten ermöglichen. Bereits an dieser Stelle lässt sich durch eine Unterabtastung die Größe der Ausgabe reduzieren. Die so definierte Ausgabe soll nun nicht in Form einer Datei auf das Endgerät abgelegt, sondern zur weiteren Verarbeitung Frame für Frame umgeleitet werden. Dazu ist erforderlich, dass die CameraManager Klasse die Funktion *captureOutput* des protocols *AVCaptureVideoDataOutputSampleBufferDelegate* implementiert und sich dort selbst als ausführende Instanz registriert. Die so nach dem delegate pattern durchgeführte Aufgabenverteilung führt dazu, dass die *captureOutput* Funktion mit jedem neuen Frame der Kamera aufgerufen wird. Jedem Aufruf wird das aktuelle Einzelbild in Form eines *CMSampleBuffer*-Objekts übergeben. Dieses Objekt enthält neben den unkomprimierten Rohdaten des Einzelbildes auch Informationen zum Aufnahmezeitpunkt und Charaktereigenschaften wie Auflösung und Ausgabeparametern. Es lässt sich aber aufgrund seiner Beschaffenheit[9] und vor allem seiner Größe nicht direkt an die verbundenen Peers übertragen. An dieser Stelle beginnen also eine Reihe von Verarbeitungsschritten um das Einzelbild für die Übertragung aufzubereiten. Die Aufgabe des CameraManagers ist an dieser Stelle erst einmal beendet und das CMSampleBuffer-Objekt wird zur Komprimierung an den Codec übergeben.

4.5.2 Direkter Zugriff auf die Video-Codierung

Eine performante und hardwarenahe Lösung ein Videosignal zu codieren und dadurch die Datenrate zu verringern, stellt das im Konzept bereits beschriebene VideoToolbox Framework bereit. Die Implementierung des Frameworks findet in der Klasse Codec statt. Genau wie die Erzeugung der Kernkomponenten *RTStream*, erfolgt die Initialisierung des Codecs nach dem Singleton-Entwurfsmuster. Dadurch kann ein Zugriff auf die statische Instanz ohne zusätzliche Initialisierung erfolgen. Sie steht der Anwendung also zu jedem Zeitpunkt zur Verfügung, um sowohl Einzelbilder zu codieren als auch zu decodieren. Der CameraManager macht davon Gebrauch, um die bereitgestellten Frames zu codieren, ohne zuvor eine Instanz des Codecs initialisieren zu müssen. Ein beispielhafter Aufruf zur Codierung sieht wie folgt aus.

```
Codec.H264.encodeFrame(sampleBuffer)
```

Listing 7 Funktionsaufruf des H.264 Codierers

[9] *Es ist nicht NSData kompatibel.*

Dabei bleibt auf dem ersten Blick verborgen, welche Komplexität sich hinter diesem Aufruf verbirgt. Einer zuvor mit zehn Parametern initialisierten *VTCompressionSession* dient das übergebene Objekt als Ausgangsmaterial zur Codierung. Die Mächtigkeit des Frameworks, die sich aus der Parametrisierung schon erahnen lässt, ist in der Developer Library jedoch gänzlich undokumentiert. Lediglich die auf der *Worldwide Developers Conference (WWDC)* präsentierte Einführung des Frameworks im Jahr 2014, veranschaulicht die Umsetzung (Apple Inc., 2014). Über die eigentliche Funktion *VTCompressionSessionEncodeFrame* wird das gelieferte Einzelbild, nach Vorgaben der bei der Initialisierung übergebenen Parameter, gemäß den Algorithmen des H.264 Codecs komprimiert. Die *VTCompressionSession* verfügt dabei intern über einen Speicher, nach dem die zeitliche Redundanzreduktion erfolgt. Über eine *callback*-Funktion wird das komprimierte Einzelbild zurückgegeben. Dieses liegt anschließend ebenfalls als *CMSampleBuffer*-Objekt vor, doch ist in der internen Struktur stark verändert. Anstatt der Rohdaten einer Rastergrafik beinhaltet es nun die reduzierten Bildinformationen vom Typ *CMBlockBuffer*. Auch die im Objekt eingebetteten Zusatzinformationen haben sich geändert. Sie enthalten nun essenzielle Informationen, die zur Decodierung notwendig sind und das Einzelbild im Kontext der Group of Pictures richtig einordnen. In der Terminologie des H.264 Codecs werden diesen Informationen *Sequence Parameter Sets (SPS)* und *Picture Parameter Sets (PPS)* genannt.

Das Verschicken des nun in der Größe deutlich reduzierten Einzelbilds ist aber aufgrund seiner vorliegenden Form als CMSampleBuffer immer noch nicht möglich. Als nächster Schritt erfolgt die Aufbereitung in eine für die Datenübertragung geeignete Form. Außerdem muss noch sichergestellt werden, dass sich der Codec an die immer wieder variierende Ausgabe des CameraManagers anpasst. Wenn sich durch übertragungsbedingte Anpassungen die Auflösung ändert, muss dies auch vom Codec berücksichtigt werden. Selbiges gilt für Anpassungen der angestrebten Bitrate die vom Codec bestimmt wird. Eine Änderung der Parameter hat immer eine neue Initialisierung der *VTCompressionSession* zur Folge. Sobald signalisiert wird, dass sich die Parameter verändert haben, wird eine Updatefunktion aufgerufen, die die alte Session durch eine neue ersetzt.

4.5.3 Network Abstraction Layer Unit

Als letzter Schritt der Bildverarbeitungspipeline steht die Umformung der *CMSampleBuffer*-Objekte in einen elementaren Stream an. Dazu werden alle bildrelevanten Informationen aus dem Objekt extrahiert und in ein Byte Array importiert, dass sich über das Multipeer Connectivity Framework übertragen lässt. Diese Art der Umformung ist jedoch nicht nur für dieses Framework notwendig, sondern fällt bei nahezu jeder paket- oder streambasierten Übertragung an. Aus diesem Grund ist der H.264 Codec auch so konzipiert, dass sich das Videosignal samt Zusatzinformationen in kleine Pakete, sogenannte Network Abstraction Layer (NAL) Units, zerlegen lässt. Für diese Zerlegung existieren seitens des *AVFoundation* und *VideoToolbox* Framework derzeit keine vorgefertigten Werkzeuge, doch es werden unterstützende Funktionen angeboten.

```
public func CMVideoFormatDescriptionGetH264ParameterSetAtIndex
(
   videoDesc: CMFormatDescription,
 _ parameterSetIndex: Int,
 _ parameterSetPointerOut:UnsafeMutablePointer<UnsafePointer<UInt8>>,
 _ parameterSetSizeOut: UnsafeMutablePointer<Int>,
 _ parameterSetCountOut: UnsafeMutablePointer<Int>,
 _ NALUnitHeaderLengthOut: UnsafeMutablePointer<Int32>

) -> OSStatus
```

Listing 8 Funktion zum Extrahieren von H.264 Parameter Sets

Mit der dargestellten Funktion können aus der Formatbeschreibung des *CMSampleBuffer* die SPS und PPS Parametersets gewonnen werden. In einem Byte-Array werden die Informationen dann nacheinander abgelegt, und zur Separierung mit folgendem Code voneinander getrennt.

0x00, 0x00, 0x00, 0x01

Nach dem gleichen Schema wird auch der Blockbuffer mit den Bildinformationen aus dem *CMSampleBuffer* extrahiert. Damit enthält das Byte-Array nach erfolgreicher Entnahme aller Informationen die folgenden Daten

TRENNCODE – SPS – TRENNCODE – PPS – TRENNCODE - BLOCKBUFFER

Auf die gleiche Art und Weise lässt sich aus den Byte-Daten auf der Empfängerseite auch wieder ein *CMSampleBuffer*-Objekt rekonstruieren. Dafür wurde die Klasse NALU entworfen, dessen Konstruktor bei der Instanziierung die als NSData empfangen Werte benötigt. Darin werden die Trenncodes ermittelt und aus den SPS- und PPS-Sets eine Formatbeschreibung erzeugt. Zusammen mit dem letzten Segment, dem Blockbuffer ent-

steht so erneut ein CMSampleBuffer. Einzig der Zeitstempel des ursprünglichen Aufnahmezeitpunkts geht bei dieser Fragmentierung verloren. Dieser muss bereits vor den Schritten gesondert behandelt und als weitere Information der Frame-Nachricht mitübermittelt werden.

4.6 Nachrichten- und Streamübertragung

Aus dem vorherigen Abschnitt lässt sich die Entscheidung für die bevorzugte Übertragungsart der Einzelbilder ableiten. Vor der eingehenden Recherche über die Versandarten stellte die streambasierte Übertragung einen vielversprechenden Ansatz dar, die Menge an Daten performant zu übertragen. Doch mit der Einarbeitung und nicht zuletzt durch die Analyse der verfügbaren Daten kamen erste Zweifel auf. Da der Stream immerzu gleich große Blöcke aus einem zu sendenden Buffer entnimmt und an den Empfänger übertragt, muss an dieser Stelle überprüft werden, ob sich in den empfangenen Daten bereits ein ganzes Einzelbild befindet. Die Größe der Einzelbilder ist jedoch variable und wird durch die aktuelle Auflösung, der Bitrate und besonders durch die Unterteilung in Intra- und Differenzbildern beeinflusst. Der Empfänger verfügt bei den eingehenden Segmenten nicht über die Kenntnis, welche und wie viele Informationen eingegangen sind. Darum muss man jeden Block an Daten bei der Ankunft auf vorhandene Trennzeichen untersuchen. Zur eindeutigen Unterscheidung einzelner Frames, wurde neben den Trenncodes innerhalb des Frames, noch ein Endcode nach jedem ganzen Einzelbild hinzugefügt. Nach dieser eindeutigen Byte-Folge wird jeder eintreffende Block mit folgender Schleife durchsucht.

```
for var  n = self.inBuffer.count;
         n > inBuffer.count-self.chunkSize;
         n -= 1 {
                    if(
                          inBuffer[n-3] == 0x01 &&
                          inBuffer[n-2] == 0xff &&
                          inBuffer[n-1] == 0x01 &&
                          inBuffer[n] == 0xff)
                          {
                          //Found an endcode at position n
                          return n
                          }
                }
```

Listing 9 Schleife zum Ermitteln von Trennzeichen

Die so umgesetzte Bestimmung der Bildinformationen ist funktional korrekt doch wenig praktikabel. Im Worst Case, wenn die Datenrate des Videosignals sehr hoch ist und damit der Blockbuffer mehrere Segmente füllen würde, würde diese Schleife mehrere Tausend

Male durchlaufen werden müssen, bis ein Endcode gefunden wird. Da dies bis zu dreißigmal die Sekunde geschieht, spiegelt sich das schlechte Laufzeitverhalten in einer merkbaren Verzögerung des Videosignals wieder. Zumal besagtes nicht den gesetzten Echtzeitanforderungen entspricht, wurde die im Konzept beschriebene Übertragung mittels Nachrichten auch für die Frames umgesetzt.

Die Übertragung von Nachrichten mittels der *sendMessage* Funktion des Multipeer Connectivity Frameworks unterbindet diese Problematik gänzlich. Eine Nachricht enthält immer genau ein Frame. Lediglich die korrekte Einordnung der Frames muss gewährleistet werden, da diese Übertragung im Gegensatz zum Stream nicht seriell erfolgt. Doch dem kommt zugute, dass eine übermittelte Nachricht neben dem Einzelbild noch weitere Informationen beinhalten kann, die der konzipierten Echtzeitübertragung dienen. Die zunächst als performancekritisch eingeschätzte Funktion *NSKeyedArchiver.archivedDataWithRootObject(),* die vor dem Versand jeder Nachricht eine Umwandlung eines Dictionaries in *NSData* durchführt, erweist sich als unbedenklich.

4.7 Einhaltung der Echtzeitanforderungen

Damit die selbst gesetzten Anforderungen an die Übertragung eingehalten werden, wird die Übertragung kontinuierlich überwacht und ausgewertet. Wie die in Kapitel drei spezifizierten Verfahren zur Sicherung der Verbindungsqualität implementiert werden, wird im folgenden Abschnitt erläutert.

4.7.1 Framebehandlung

Die wohl einflussreichsten Auswirkungen auf die Echtzeit hat die Framebehandlung. Bereits zum Zeitpunkt des Versands als auch beim Empfang, können wertvolle Informationen gewonnen werden, die zur Einhaltung der Echtzeitanforderungen dienen. Aufseiten des Senders betrifft dies den zuvor erwähnten *Leaky-Bucket Algorithmus*. Nachfolgend dargestellt sind die Implementierung der Warteschlange und des Überlaufs der entscheidet, ob ein Frame versendet oder verworfen wird. Zu tragen kommt diese Logik, nachdem ein Frame vom Codec erfolgreich codiert und in eine übertragbare Form überführt wurde.

```
//send message only if other are connected
if self.connectedPeers.isEmpty == false {
// max queue limit
     if self.outputQueue.count < 4 {
              //add nalu to queue
              self.outputQueue.append(nalUnit)
              dispatch_barrier_sync(self.sendingQueue, {
                     self.sendFrame(timestamp)
              })
     }else{
              //Inform other peers and change parameters
              self.bufferOverflowAlert()
    }
}
```

Listing 10 Implementierung des leaky-bucket Algorithmus

Die Wahl der maximalen Warteschlangengröße von drei Frames ist absichtlich sehr klein gehalten. Dadurch soll verhindert werden, dass eine spürbare Verzögerung durch das Vorhalten der Nachrichten entsteht. Wenn die Warteschlange bereits voll ist, wird der neue Frame nicht ans Ende eingereiht, sondern komplett verworfen. Außerdem erfolgt eine Benachrichtigung zur Überlastbehandlung gemäß Konzept. Ein Platz in der Warteschlange wird frei, wenn ein Frame erfolgreich versandt wurde. Dies geschieht innerhalb der *sendFrame* Funktion. Dabei wird eine Nachricht aus dem übergebenen Zeitstempel und dem jeweiligen Element der Warteschleife konstruiert. Erst nach der abgeschlossenen Übermittlung wird der Frame aus der Wartschleife entfernt.

Nachdem die Nachricht den Empfänger erreicht hat, hat der enthaltene Zeitstempel einen hohen Stellenwert. Über ihn lässt sich ermitteln, ob die Nachricht mit dem Frame zu einem zeitlich versetzen Zeitpunkt eingegangen ist. Um dies zu bestimmen, wäre es unzureichend den aktuellen Zeitstempel mit dem vorherigem zu vergleichen, da nicht gewährleistet ist, dass die Frames in Ausgabereihenfolge eintreffen. Weil eine Synchronisation der Systemzeit zwischen Sender und Empfänger nicht vorliegt, ist auch eine fixe zeitliche Grenze nicht zu bestimmen. Außerdem kann die Framerate variieren, was ebenfalls auf die Toleranz der Zeit Einfluss nimmt. An dieser Stelle wird auf die Datenhaltung zurückgegriffen, die zu jeder Nachricht eines jeweiligen Peers auch die empfangenen Zeitstempel speichert. Auf Basis der letzten zehn Zeitstempel wird ein Mittelwert gebildet, der eine Bewertung des neuen Zeitstempels in Abhängigkeit historischer Daten erlaubt. Mittels der Vergleichsoperation *CMTimeCompare* kann somit entschieden werden, ob die neue Nachricht bereits hätte ankommen müssen. Ist dies der Fall, wird nicht nur die Nachricht verworfen, sondern es werden auch Vorkehrungen getroffen die Datenrate zu reduzieren.

4.7.2 Quality-of-Service Kontrolle

Neben der aktiven Erkennung von Übertragungsproblemen ist noch die passive Kontrolle der Verbindung über Paketumlaufzeiten zu implementieren. Um die Bandbreite nicht im überhöhten Maße zu belasten, werden die Umlaufzeiten periodisch und nicht kontinuierlich ermittelt. Die ControlChanelManager Instanz bestimmt dazu timerbasiert über den Funktionsaufruf *determineRoundTripTime(),* die Kennzahlen für alle verbundenen Peers. Die so nach dem Konzept mit Zeitstempeln versehenen Nachrichten verfügen über die Kennzeichnung *rttReq (Round Trip Time Request).* Auf der Gegenseite wird der Anforderung nachgegangen und das Paket mit weiteren Zeitstempeln gefüllt. Auf dem Rückweg trägt die Nachricht die Kennzeichnung *rttRes (Round Trip Time Response).* Aus der Antwort kann dann die Paketumlaufzeit, wie in Abbildung 7 dargestellt, berechnet werden. Dieses Request-Response-Verfahren wird pro Peer solange wiederholt, bis 15 Antworten vorliegen. Aus allen so ermittelten Zeiten pro Peer lässt sich eine gemittelte Übertragungszeit errechnen, die in die *RTStreamPeer* Datenhaltungsinstanz einfließt. Dass eine einzige Messung der Umlaufzeit nicht aussagekräftig genug ist, verdeutlicht das folgende Diagramm. Es zeigt die im Vorfeld durchgeführten Testmessungen und gibt die Paketumlaufzeiten an, die sich aus einem Umlauf ergeben.

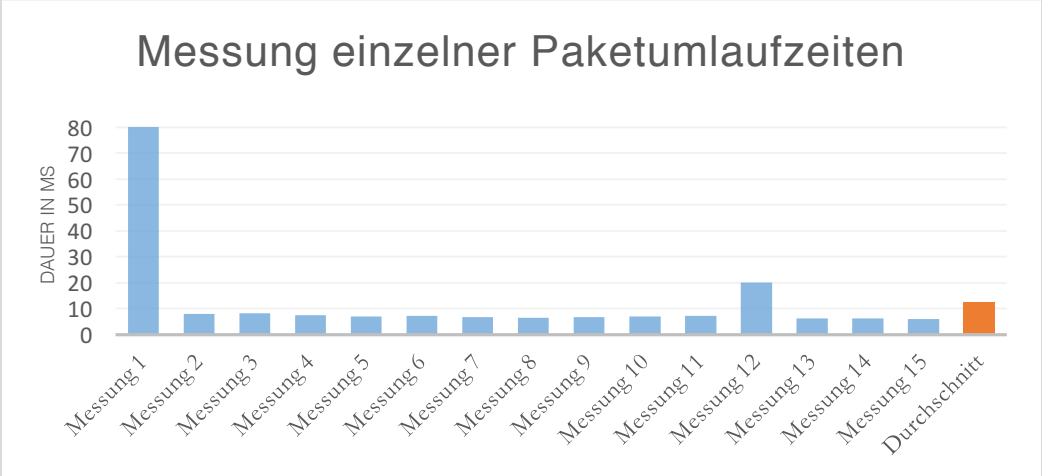

Abbildung 12 Diagramm der Messergebnisse der Paketumlaufzeit

Deutlich erkennbar sind sporadische Abweichungen, die ein Vielfaches der sonst annähernd konstanten Zeit betragen. Einen durch äußerliche Einflüsse hervorgerufenen Grund für diese Differenzen ist nicht auszumachen, da die Endgeräte während der Messung in unmittelbarer Nähe zueinander lagen und auch sonst keine weitere Übertragung erfolgte. Der aus allen Zeiten ermittelte arithmetische Durchschnitt reduziert diese Ungenauigkeiten aber weitestgehend. Eine vorherige Eliminierung der Extremwerte oder die Bestimmung des Median würde indes die Genauigkeit noch steigern, doch auch zusätzliche Rechenkapazitäten erfordern.

Stellt man nun die Durchschnittszeiten verschiedener Messungen gegenüber, so ist erkennbar, dass sich darin nach wie vor Abweichungen befinden. Die Differenzen sind aber um ein Vielfaches geringer und lassen sich für die weitere Auswertung verwenden, wenngleich man Toleranzen einplanen muss.

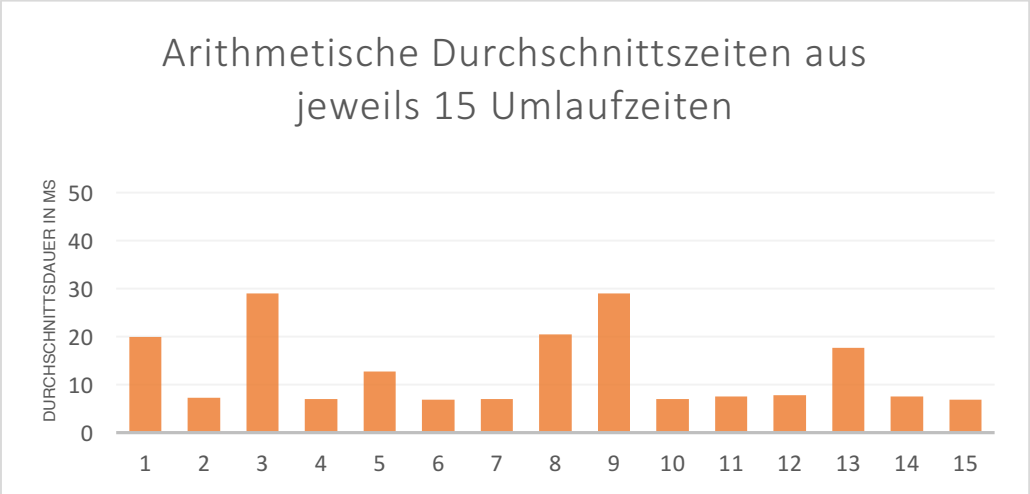

Abbildung 14 Durchschnittliche Paketumlaufzeiten unterschiedlicher Messungen

Die durch die Testmessungen gewonnenen Erkenntnisse helfen auch, die periodisch in die Datenhaltung einfließenden Mittelwerte richtig zu deuten. Einen direkten Rückschluss auf die Verbindungsqualität zwischen aktueller und letzter Messung zu ziehen wäre falsch, denn ein fester Faktor um den sich eine Zeit verändern muss, ist nicht zu bestimmen. Hingegen lässt durch die rückwirkende Betrachtung mehrerer Intervalle eine Tendenz erahnen, welche die sich abzeichnende Entwicklung der Verbindung beschreibt. Die Tendenz gibt die *RTStreamPeer* Instanz mittels folgender Funktion aus.

```
func getRoundTripTimeTendency()->Int{
        var tendency :Int = 0
        if _roundTripTimeHistory.count > 1{
            for i in 1 ..< _roundTripTimeHistory.count {
                if (_roundTripTimeHistory[i-1]
                    < _roundTripTimeHistory[i])
                {
                    tendency = tendency + 1
                }
                if(_roundTripTimeHistory[i-1]
                    > _roundTripTimeHistory[i])
                {
                    tendency = tendency - 1
                }
            }
        }
        return tendency
    }
```

Abbildung 13 Funktion zur tendenziellen Bestimmung der Verbindungsqualität

Der so zurückgegebene Ganzzahlwert, der sowohl positiv als auch negativ sein kann, gibt in Summe die Anzahl steigender als auch fallender Umlaufzeiten der letzten zehn Messungen an. Stellt man nun für die Gesamtbewertung die aktuelle durchschnittliche Umlaufzeit mit der Gesamtdurchschnittszeit gegenüber und zieht zusätzlich die Tendenz hinzu, lässt dies eine gemäßigte aber numerisch fundierte Reaktion auf die Verbindungsqualität zu.

4.7.3 Überlastbehandlung

Die zuvor zur Einhaltung der Echtzeitanforderungen definierten Funktionalitäten enthalten allesamt Bereiche, in denen Bezug auf die Anpassung des Videosignals genommen wird. An einigen Stellen ist die Rede von drastischen Maßnahmen an anderen wiederum von Gemäßigten. Da an unterschiedlichsten Stellen jedoch eine Anpassung nach gleichem Muster erfolgt, werden die vorzunehmenden Maßnahmen an einheitlicher Stelle definiert. Daher wird die Klasse *Strategies* definiert, die ausschließlich über statische Variablen verfügt. Diese Variablen sind dadurch im Programm an beliebiger Stelle verfügbar, ohne zuvor eine Instanz der Klasse *Strategies* initiieren zu müssen. Den Variablen ist jedoch kein Wert zugeordnet, sondern definieren eine Funktion. Die so konstruierte Funktion lässt sich dann wie eine Variable übergeben und ausführen. Die Schlüsselkomponente *RTStream* verfügt über eine Funktion *changeStrategy(),* die die so eingebetteten Funktionen entgegennimmt. An einer beliebigen Stelle im Toolkit kann dann, wie beispielhaft dargestellt, eine Anpassung des Videosignals vorgenommen werden.

```
RTStream.sharedInstance.changeStrategy(Strategies.decreaseResolution)
```

Listing 11 Aufruf zum Ändern der Auflösung

Diese Architektur bietet sich an, da die *RTStream* Instanz Zugriff auf alle an der Veränderung notwendigen Instanzen hat und die Logik zur Anpassung des Videosignals lediglich innerhalb der Strategies Klasse umgesetzt werden muss. Die zu beachtenden Aspekte bei der Anpassung des Videosignals sind dabei nicht ganz trivial. So muss zum Beispiel bei dem gezeigten Aufruf darauf geachtet werden, dass die frei wählbare Mindestauflösung nicht unterschritten wird. Außerdem, da dieser Aufruf primär den Zweck erfüllen soll die Datenrate zu senken, ist dem nicht einzig und allein damit genüge getan die Auflösung herabzusetzen. Zu jeder Auflösung gibt es einen entsprechenden Bitrate-Bereich. Wird dieser beim Wechsel zwischen den Auflösungen nicht berücksichtigt, kann dies dazu führen, dass eine vollkommende Über- oder Unterdimensionierung vorliegt. Die Datenrate würde sich dementsprechend nicht äquivalent verändern.

Neben der sukzessiven Minderung oder Steigerung der Parameter sind aber auch Strategien definiert, die eine rapide Anpassung des Videosignals vornehmen. Beispielsweise ist eine *slowStart* Strategie implementiert. Dabei wird das Videosignal auf die kleinstmögliche Form reduziert. Dies geschieht unter Berücksichtigung der selbst gesetzten Schranken für die Minimalanforderungen. Eine Anwendung dieser Strategie findet statt, wenn ein deutlich verspäteter Frame beim Empfänger eingeht und dieser den Sender darüber informiert.

4.8 Multithreading

Die Echtzeitanforderungen werden nicht allein durch performante Hardware gewährleistet. Von ebenso besonderer Bedeutung ist die richtige Lastverteilung der Aufgaben auf die verfügbaren Prozessoren. Diese Aufteilung verhindert nicht nur die Überstrapazierung einer einzelnen Recheneinheit, sondern ermöglicht auch die umso wichtigere parallele Ausführung von Verarbeitungsschritten.

4.8.1 Grand Central Dispatch

Im Jahr 2009 stellte Apple, mit der zunehmenden Verbreitung von Mehrkernprozessoren, den Entwicklern eine Schnittstelle unter den Namen Grand Central Dispatch (GCD) bereit, die die gleichzeitige Aufführung von Programmteilen ermöglichte. Die darin eingeführte Abstraktion auf Grundlage von Queues, vereinfacht die durch Multithreading entstehenden Herausforderungen und Verwaltungsaufgaben enorm. Über die Möglichkeit hinaus Aufgaben parallel ausführen zu können, übernimmt GCD die gesamte Zeitplanung der in Queues eingeteilten Blöcke. Dies erlaubt eine Priorisierung der zur Ausführung bestimmten Abschnitte, die in unterschiedlichen Quality of Service Queue-Typen definiert werden. Vom System vorgegeben existieren vier nachfolgend dargestellte Prioritätsstufen.

```
QOS_CLASS_USER_INTERACTIVE
QOS_CLASS_USER_INITIATED
QOS_CLASS_UTILITY
QOS_CLASS_BACKGROUND
```

Neben den in absteigender Reihenfolge ihrer Priorität aufgelisteten Queues, existiert noch eine weitere, sogenannte *main queue*. Sie dient vornehmlich dem Aufruf untergeordneter Queues und der Aktualisierung der Benutzeroberfläche. Falls die vorgegebenen Queues zur Kategorisierung nicht ausreichen, besteht zudem noch die Möglichkeit beliebig viele weitere Queues zu definieren. Grundsätzlich gilt bei der Verwendung von priorisierten Queues da-

rauf achtzugeben, dass die jeweilige Ausführdauer an die Priorität angepasst ist. Ein langsamer und lang andauernder Prozess in der *interactive* Queue würde möglicherweise dazu führen, dass das Gesamtsystem blockiert wird.

Neben der Priorisierung besteht noch die Möglichkeit das Verhalten der Abarbeitung der zur Queue hinzugefügten Blöcke zu beeinflussen. Der Aufruf neuer Aufgaben in als *serial* definierten Queues, erfolgt immer erst nach beendeter Ausführung vorangegangener Aufgaben. Innerhalb dieser Queue findet also keine parallele Ausführung statt. In als *concurrent* gekennzeichneten Queues hingegen, erfolgt der Aufruf weiterer Aufgaben bereits nach dem Aufruf vorangegangener Aufgaben, ohne auf die vorherige Beendung zu warten. Weiteren Einfluss auf die Abarbeitung haben die Kennzeichner *sync* und *async*. Sie werden beim Anfügen von Aufgaben in die Queue verwendet. Werden als *async* gekennzeichnete Aufgaben angefügt, wird das weitere Programm unmittelbar nach der Einreihung fortgeführt. Die Abarbeitung der Queue findet dann nebenläufig statt. Bei der synchronen (*sync*) Variante hingegen, wird auf die Rückgabe der Queue gewartet, bevor das Hauptprogramm fortgeführt wird.

Für die Echtzeitübertragung bieten die angebotenen Optionen großes Potenzial, um Nebenläufigkeiten zu definieren und voneinander unabhängige Prozesse zu separieren. Die korrekte Zuweisung der Prioritäten und Abarbeitungsmethoden ist dabei federführend am Erfolg der Einhaltung der Echtzeitanforderungen beteiligt. Eine falsche Konfiguration hingegen kann sich durchaus negativ auswirken und ein Fehlverhalten hervorrufen.

Bei der gesamten Implementierung wurde stark von der *GCD*-Schnittstelle Gebrauch gemacht. Dabei wurde das Toolkit in drei grundsätzliche Verarbeitungsstränge getrennt. Der Kamerazugriff findet in einer selbst definierten, seriellen Queue statt. Bei der Ausgabe eines Frames wird die Funktion zur Codierung asynchron an die priorisierte *utility queue* angehängt. So wird sichergestellt, dass die Kamerasession unabhängig von anderen, lang andauernden Prozessen bleibt. Die Kodierung der Frames findet im Zuge dessen innerhalb einer *concurrent* Queue statt. So ist es möglich, viele aufwendige Schritte parallel im Hintergrund laufen zu lassen. Das Versenden der Nachrichten erfolgt, dem Grundsatz zu trotz lang andauernde Aufgaben keine hohen Prioritäten zuzuordnen, innerhalb der *interactive* Queue. Dies erwies sich als notwendiges Mittel, um der Echtzeit gerecht zu werden.

4.8.2 Kritische Sektionen

Durch die parallele Ausführung von Aufgaben entstehen neue Herausforderungen. An Stellen in Programm an denen gemeinsame Lese- und Schreibezugriffe existieren, kann es zu sogenannten *race conditions* kommen. Darunter sind konkurrierende Zugriffe auf gleiche Ressourcen zu verstehen, welche bei gleichzeitiger Inanspruchnahme zu einem Fehlverhalten führen können. Wenngleich parallel lesende Zugriffe im Allgemeinen kein Problem darstellen, kann dies bei zeitgleichem Schreiben verheerende Auswirkungen haben. Insbesondere in Verbindung mit Arrays kann dies zu Laufzeitfehlern führen, die die Anwendung zum Absturz bringen können.

Solche konkurrierenden Zugriffe erfolgen im Toolkit mehrfach. Durch die vielen Operationen pro Sekunde und die Tiefe der Parallelisierung wäre eine selbstentwickelte Verwaltung zur Kontrolle des Zugriffs nicht praktikabel. Das *GCD* biete diesbezüglich aber ebenfalls eine Lösung. Wenn immer Zugriffe auf Elemente erfolgen, die nicht *thread safe* sind, also durch zeitgleichen Zugriff ein Fehlverhalten auslösen könnten, können diese in *Dispatch Barrier* Blöcken definiert werden. Für alle Aufgaben, die zu einem solchen Block zusammengefasst sind, stellt das GCD sicher, dass zur gleichen Zeit keine anderen Aufgaben erledigt werden. Es wird sozusagen ein künstlicher Flaschenhals erzeugt, durch den nur die definierten Aufgaben in serieller Form abgearbeitet werden. Nach Vollendung des Blocks, nehmen alle anderen Aufgaben ihre Arbeit wieder in paralleler Form auf. Folgendes Beispiel zeigt die Anwendung dieser kritischen Sektionen in der Funktion zum Senden eines Frames.

```
//perform critical operations on outputQueue
dispatch_barrier_sync(criticalQueueAccess, { () -> Void in
     //get actual frame from queue
     let frameToSend = self.outputQueue.first
     message["frame"] = frameToSend
})

//send frame message to all receivers
RTStream.sharedInstance.mcManager.sendMessageToAllReceivers(
messageToSend: NSKeyedArchiver.archivedDataWithRootObject(message))

//perform critical operations on outputQueue
dispatch_barrier_sync(criticalQueueAccess, { () -> Void in
     //remove the frame that was send from queue
     self.outputQueue.removeFirst()
})
```

Listing 12 Verwendung von Dispatch Barrier Blöcken

Die zuvor erwähnte Flaschenhalsmechanik der Barrier-Blöcke ist der Grund, für die in dem Ausschnitt durchgeführte Aufteilung in zwei unterschiedliche Blöcke. Da sich zwischen ihnen die langandauernde Aufgabe des eigentlichen Versands befindet, die keinen kritischen Zugriff enthält, wird diese vom Block ausgeschlossen. Andernfalls würde das gesamte System für die Dauer der Übertragung blockiert werden. Darum wurde an dieser und weiteren Stellen darauf geachtet, die kritischen Blöcke so kurz wie möglich zu halten.

4.9 Bereitstellung von Frames

Die eigentliche Ausgabe des Toolkits endet mit der Bereitstellung der codierten Frames. Die Art des Ausspielens ist der individuellen Implementierung überlassen. Das Toolkit selbst setzt lediglich die Echtzeitanforderungen um und bietet die Möglichkeit, eine verzögerungsfreie Wiedergabe anzubieten. Das *delegate pattern* schafft diesbezüglich eine Schnittstelle zwischen dem Toolkit und der jeweiligen Verwendung. In der *RTStream* Klasse wird ein *protocol* definiert, das bei einer Anwendung individuell implementiert werden kann.

```
protocol RTStreamDelegate {
    //being called everytime a new frame arrives
    func displayFrame(
                      encodedFrame :CMSampleBuffer,
                      fromPeer :MCPeerID
                      )
}
```

Listing 13 Protocol-Definition zur Bereitstellung von Frames

Der so im *protocol* deklarierten Funktion *displayFrame* wird dann, unter Nennung des Senders in Form der PeerID, das im H.264 codierte Einzelbild übergeben. Das Toolkit ruft diese Funktion bei jedem eingehenden Frame auf, egal von welchem Sender es kommt. Dies ermöglicht, in Abhängigkeit der Implementierung, auch die Darstellung von Multicam Views. Es ist jedoch darauf zu achten, innerhalb dieser Funktion nicht allzu komplexe Berechnungen durchzuführen. Denn mit jedem weiteren Peer als Sender verdoppelt sich die Anzahl der Aufrufe. Eine beispielhafte Implementierung zum Ausspielen des Videosignals ist dem nächsten Kapitel zu entnehmen.

5 Verwendung des Toolkits

5.1 Umsetzung einer Beispielanwendung

Zur Darstellung einer möglichen Verwendung des Toolkits wurde im Verlauf der Arbeit auch eine Beispielanwendung umgesetzt. Sie demonstriert die implementierten Konfigurationsmöglichkeiten und diente obendrein während der Umsetzung als Funktionstest. Das nachfolgend abgebildete UML-Diagramm veranschaulicht die für eine Beispielanwendung nach außen relevanten Klassen des Toolkits.

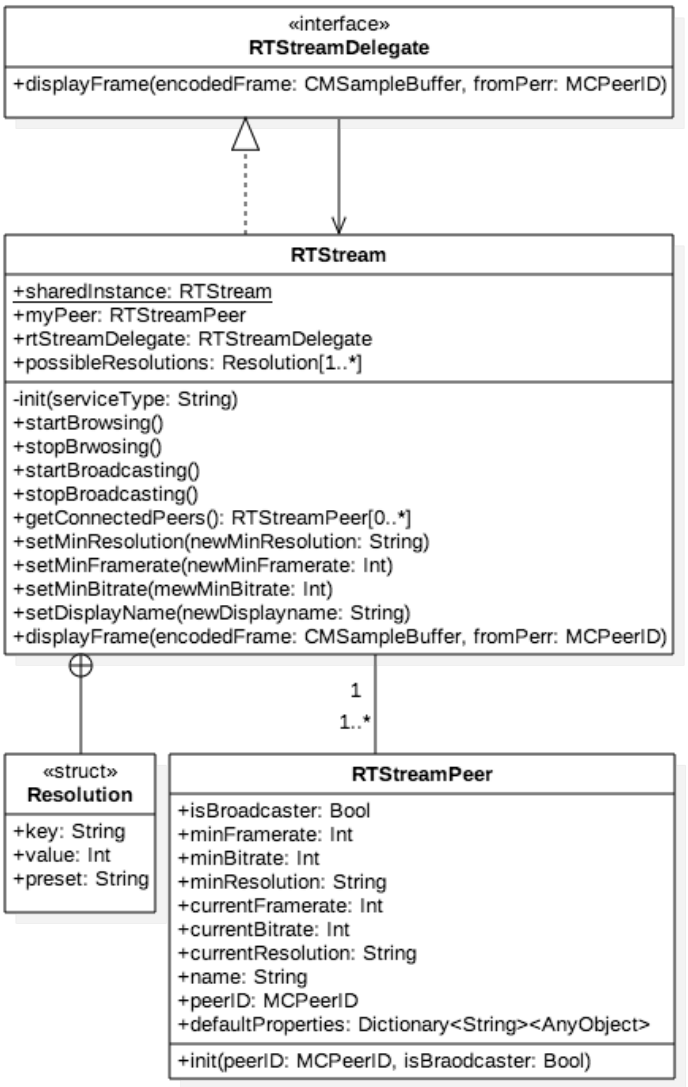

Abbildung 15 Reduziertes UML-Klassendiagram des Toolkits

Dabei handelt es sich lediglich um ein vereinfachtes Diagramm, das um die Komplexität zu reduzieren, lediglich die Kernkomponenten beschreibt. Für die Implementation einer Anwendung sind aber nur die in diesem Diagramm erfassten Klassen von Bedeutung, was nicht heißt, dass die restlichen Klassen belanglos sind. Gegenteiliges ist der Fall – alle zuvor beschriebenen Klassen müssen für die Verwendung des Toolkits zum Projekt der Anwendung hinzugefügt werden.

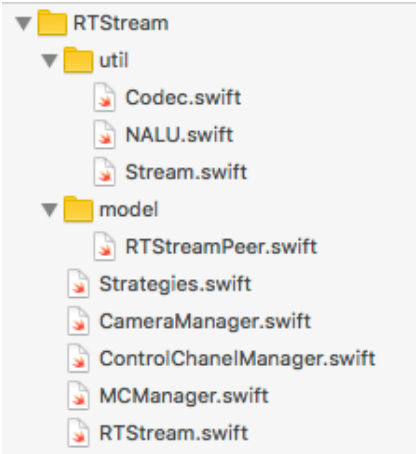

Abbildung 16 Projektdateien des Toolkits

Damit ist die Integration der Echtzeitübertragung im Wesentlichen schon abgeschlossen. Durch das *singleton pattern* und das statische *sharedInstance* Attribut, besteht zum Zeitpunkt der Ausführung der Anwendung bereits Zugriff auf die im Diagramm dargestellten Attribute und Funktionen. Alle benötigten Instanzen für die Übertragung werden mit der Initialisierung der *sharedInstance* ebenfalls initialisiert. Dadurch sollte das im ersten Kapitel erläuterte Ziel, der einfachen Benutzbarkeit, in jedem Fall erreicht worden sein.

Eine aktive Kommunikation zwischen Peers, die selbstverständlich nur stattfinden kann wenn die Anwendung auf zwei Endgeräten läuft, findet zu diesem Zeitpunkt aber noch nicht statt. Die Funktionalitäten des im Toolkit integrierte Multipeer Connectivity Frameworks werden erst durch einen Funktionsaufruf gestartet. Dies ließe sich selbstverständlich mit dem Start der Anwendung automatisieren, oder wie im Fall der Beispielanwendung über eine Benutzeroberfläche steuern. Die Oberfläche beinhaltet alle Optionen, die vom Benutzer selbst bestimmt werden können. Die Funktionen, die dadurch angesprochen werden, fangen nach Möglichkeit falsche Werte, wie zum Beispiel zu hohe Bildfrequenzen, ab. Durch welche User Interface-Elemente die Eingabe erfolgt, obliegt der jeweiligen Implementierung. Für einige Optionen, wie etwa den möglichen Auflösungen, liefert das Toolkit vordefinierte Werte über das Attribut *possibleResolutions*.

Die nachfolgende Abbildung zeigt auf der linken Seite die Benutzeroberfläche zur Konfiguration des Toolkits. Auf der rechten Seite dargestellt sind die jeweiligen Funktions-

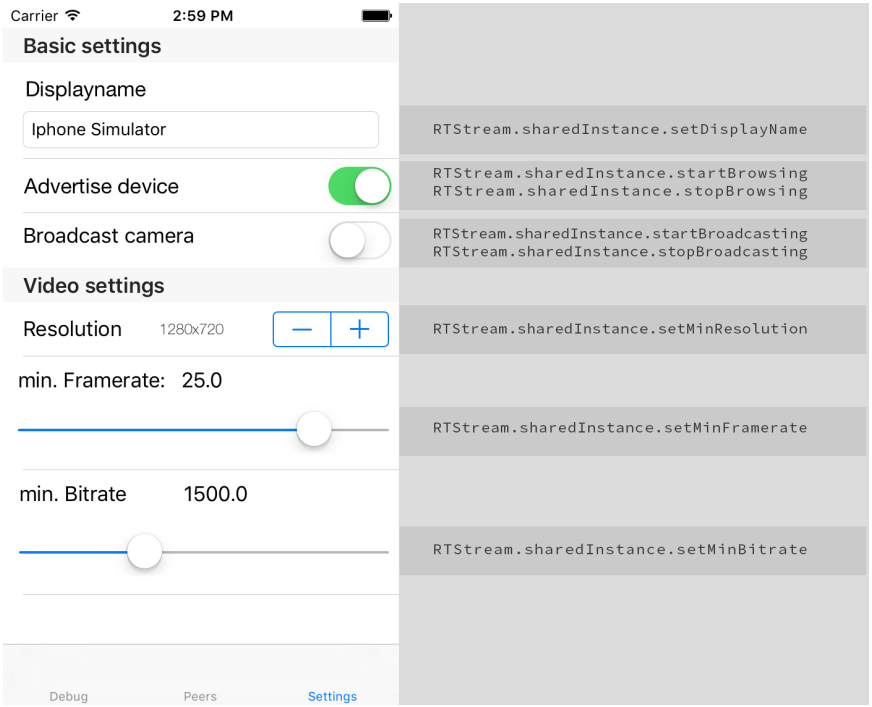

Abbildung 17 Einstellungsmöglichkeiten samt Funktionsaufrufe der Beispielanwendung

aufrufe, wie sie auch dem UML-Diagramm entnommen werden können. Unter Umständen müssen die vom User Interface gelieferten Werte noch in passende Typen umgewandelt werden, damit sie als Parameter entgegen genommen werden können.

Wenn der Schalter *Advertise device* umgelegt ist, und infolgedessen die Funktion *startBrowsing* ausgeführt wird, ist das Gerät für andere Endgeräte, die diese Auswahl ebenfalls getroffen haben, sichtbar. Über den Reiter Peers, am unteren Bildrand, lassen sich in der Beispielanwendung alle im Verbund befindlichen Geräte anzeigen. An dieser Stelle kann über eine Abfrage auf die Datenhaltung aller verbundener Geräte zurückgegriffen werden. Auf diese Weise lässt sich eine Tabelle mit den Namen und dem Status der Geräte zur Darstellung befüllen. Da der Inhalt der Tabelle dynamisch erstellt wird, müssen die Zellen programmatisch gefüllt werden. Die dazu zur Verfügung stehenden Funktionen müssen überschrieben und mit den relevanten Informationen aus der Datenhaltung gefüllt werden.

```
override func tableView(
     tableView: UITableView,
     numberOfRowsInSection section: Int) -> Int
{
            //return the number of connected peers
            return RTStream.sharedInstance.getConnectedPeers().count
}
```

Listing 14 Bestimmung der Anzahl verbundener Peers

Nachdem die Anzahl der benötigten Zellen anhand der Größe des Arrays der verbundenen Geräte bestimmt ist, können die Zellen mit Inhalt versehen werden. Als Inhalt dienen der hinterlegte Displayname sowie der jeweilige Status des Peers. In Abhängigkeit des jeweiligen Status wird die Zelle mit einem *Accessory* versehen, das als Verlinkung zu einer wei-

```
override func tableView(
     tableView: UITableView,
     cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{
     var peer :RTStreamPeer?
     let cell =
              tableView.dequeueReusableCellWithIdentifier("PeerCell",
              forIndexPath: indexPath)
        //get instance of RTStreamPeer for every connected Peer
     peer =
     RTStream.sharedInstance.getConnectedPeers()[indexPath.row]
     cell.textLabel?.text = peer!.name
     //add detail text and cell accessory if peer is broadcaster
     if peer!.isBroadcaster == true {
             cell.detailTextLabel?.text = "broadcasting"
     }else{
             cell.detailTextLabel?.text = "receiver"
             cell.accessoryType=UITableViewCellAccessoryType.None
     }
     //fill cell with peer informations
     return cell
}
```

Listing 15 Befüllung der Tabellenzellen anhand der RTStreamPeer-Instanzen

teren Benutzeroberfläche dient. In anderen Worten – für ein Peer der als Sender fungiert, kann in einer weiteren Oberfläche das Videosignal wiedergegeben werden.

5.2 Implementierung der Videoausgabe

Zum Ausspielen der bereitgestellten Videosignale stehen grundsätzlich zwei Optionen zur Verfügung. Zum einen das decodieren der H.264 Frames über das *VideoToolbox* Framework, mit dem das Videosignal aufseiten des Senders codiert wurde, oder über den vom *AVFoundation* Framework bereitgestellten *AVSampleBufferDisplayLayer*. Für welche Variante auch immer man sich entscheidet, obliegt der Implementierung der im *protocol* definierten Funktion *displayFrame*. In der Beispielanwendung wird die *AVSampleBufferDisplayLayer* Variante verwendet, da die Decodierung über das *VideoToolbox* Framework einen hohen Verwaltungsaufwand erfordert. Zudem liegt nach der Decodierung lediglich eine Rastergrafik vor, dessen Ausspielung ebenfalls noch koordiniert werden müsste. Die im Beispiel implementierte Variante hingegen, spielt codierte Frames direkt aus. Die integrierte Decodierung des *AVSampleBufferDisplayLayer* kommt ohne weitere Frameworks aus und es bedarf keiner komplexen Verwaltung bei sich ändernden Videoparametern oder beim Handling von zwei

Videoquellen. Die im Folgenden dargestellte Realisierung des *protocols* zeigt die einfache Handhabung zum möglichen Ausspielen des Videos. Ein Aufruf dieser Funktion erfolgt aber nur, wenn die View sich als Delegate in der *RTStream* Instanz registriert.

```
//peer will be set by the table view segue
var peerToDisplay :rtStreamPeer?
let displayLayer = AVSampleBufferDisplayLayer()

override func viewDidAppear(animated: Bool) {
     //register view to be the delegate
     RTStream.sharedInstance.rtstreamDelegate = self
}
override func viewDidDisappear(animated: Bool) {
     //unregister the view
     RTStream.sharedInstance.rtstreamDelegate = nil
}

//implemantation of the RTStreamDelegate protocol
func displayFrame(encodedFrame: CMSampleBuffer, fromPeer: MCPeerID)
{
     //ensure the frame belongs to the right source
     if fromPeer == peerToDisplay?.peerID {
         //perform UI changes in the main queue
         dispatch_async(dispatch_get_main_queue(), {
         //pass the encoded frame to the displaylayer
         self.displayLayer.enqueueSampleBuffer(encodedFrame)
         //update displaylayer if needed
         self.displayLayer.setNeedsDisplay()
         //Debug Information
         self.currentBitrateLabel.text =
           self.peerToDisplay?.currentBitrate?.description
         self.currentResolutionLabel.text =
           self.peerToDisplay?.currentResolution
         self.currentFramerateLabel.text =
         self.peerToDisplay?.currentFramerate?.description
         })
     }
}
```

Listing 16 Implementierung der Videoausgabe

Ebenfalls erforderlich, aber in diesem Ausschnitt nicht ersichtlich, ist die Zuweisung der peerToDisplay Variable. Auf Grundlage dieser Variable kann in der *displayFrame* Funktion entschieden werden, ob das Einzelbild in dem *DisplayLayer*-Element dargestellt wird. Mit diesem verhältnismäßig geringen Aufwand ist die gesamte Integration des Toolkits abgeschlossen. Nun kann eine Übertragung und Ausgabe des Videosignals erfolgen.

6 Evaluation

6.1 Realisierung der Zielsetzung

Das Ziel dieser Arbeit, eine Echtzeitvideoübertragung für das Multipeer Connectivity Framework zu schaffen, wurde auf zweierlei Sicht erreicht. Durch die Möglichkeit echtzeitbezogene Daten in Abhängigkeit der verfügbaren Bandbreite zu übertragen wurde die funktionale Seite erfüllt. Die in der Zielsetzung angestrebten qualitätssichernden Maßnahmen, wurden nicht zuletzt durch die konzeptionelle Arbeit gewährleistet. Darunter ist vor allem die mögliche Wiederverwendbarkeit in Form eines Toolkits zu verstehen. Besonders hervorzuheben ist die Effizienz der umgesetzten Lösung. Im Vergleich zu dem im ersten Kapitel erwähnten Semesterprojekt, ist nun das Übertragen von Full HD Videosignalen möglich, ohne wie zuvor bereits bei einer VGA-Auflösung die gesamte Bandbreite und CPU-Leistung der Endgeräte aufzubrauchen.

Doch angesichts des relativ kurzen Entwicklungszeitraums kann es, obwohl vielerlei Punkte berücksichtigt wurden, noch zu Stabilitätsproblemen des Toolkits kommen. Das Ausfindigmachen der Probleme ist dabei nicht immer ganz einfach. Wenn Laufzeitfehler auftreten, geschieht dies meist sporadisch und die Reproduktion ist schwierig. Die genauere Ursache ist dem *Stacktrace* nicht immer zu entnehmen und das Debugging ist mit Hürden verbunden. Sobald zur Fehleranalyse ein Breakpoint erreicht wird, wird die Verbindung in der Regel unterbrochen. Nur selten kommt es vor, dass zu diesem Zeitpunkt der Fehler auftritt, nach dem man gesucht hat. Selbige Problematik besteht bei der Untersuchung der Verbindungsqualität. Alle Werte die man zur Bewertung der Verbindung heranzieht, lassen sich zum Überprüfen lediglich ausgeben, wenn eine Kabelverbindung besteht. Wie sich diese Werte unter Realbedingungen verhalten, ist nur durch eine langwierige Auswertung von Protokolldateien möglich.

6.2 Verhalten der dynamischen Videoanpassung

Die Skalierung des Videosignals ist maßgeblich am Erfolg der Echtzeitübertragung beteiligt. Die dabei zur Verfügung stehenden Parameter, die unterschiedlich große Auswirkungen haben, werden in Abhängigkeit des Handlungsbedarfs verändert und miteinander kombiniert. Eben jene Handhabung führt dazu, dass einige Veränderungen einen deutlich

sichtbaren Einfluss auf die Qualität des Videosignals haben. Was die in der Einleitung erwähnt Quality-of-Experience betrifft, gibt es noch Optimierungsbedarf was die Nutzererwartung betrifft. Eine qualitativ deutlich wahrnehmbare Veränderung sollte demzufolge nicht allzu häufig stattfinden. Durch die derzeitige Stauvermeidungsmethodik wird das Videosignal aber sukzessiv verändert – was auch dazu führen kann, dass es vermehrt sprunghafte Wechsel gibt. Doch durch den konzeptionellen Aufbau des Toolkits lässt sich das jeweilige Verhalten zur Anpassung innerhalb der Strategien nachträglich beeinflussen. Idealerweise sollte eine Veränderung gleichförmig oder gar nur ein einziges Mal erfolgen und damit das richtige Verhältnis zwischen verfügbarer und benötigter Bandbreite treffen. Das Austarieren der passenden Strategien erforderte aber viel Zeit für die Auswertung und Deutung der protokollierten Verbindungsparameter. Insgesamt sollte man die Quality-of-Experience Faktoren auch nicht allzu streng auf eine Echtzeitvideoübertragung anwenden. Die Einhaltung der Echtzeitanforderungen erfordert hin und wieder auch mal drastische Maßnahmen, damit Verzögerungen vermieden werden.

6.3 Ausblick

Das Toolkit ist in der derzeitigen Form, ungeachtet kleiner Mängel und der möglichen Optimierung der Videosignalanpassung, ein mächtiges Werkzeug. Obwohl die Arbeit noch den Charakter eines Proof-of-Concepts genießt, lässt es sich bereits durch die einfache Integration für prototyping Zwecke verwenden. Zu einem späteren Zeitpunkt denkbar wären eine Verbreitung über *dependency manager* wie *cocoapods* oder *carthage*. Doch bevor es veröffentlicht werden würde, müssten die zuvor genannten Punkte noch nachgebessert werden. Nicht zuletzt mit dem Release von Swift 3.0, das noch im Jahr 2016 erscheinen soll, werden einige Korrekturen notwendig sein, um die korrekte Funktion nach wie vor zu gewährleisten.

Abschließend lässt sich festhalten, dass durch die Konzeption und Implementation der Echtzeitvideoübertragung vor allem ein persönlicher Mehrwert, in Bezug auf die Entwicklung mobiler Anwendungen, erreicht wurde. Die im Zuge der Arbeit geleistet Einarbeitung in Themengebiete der Videosignalverarbeitung und diversen Netzwerkprotokollen ist dabei von ebenso großer Bedeutung. So gesehen sind durch die Umsetzung des Toolkits weitaus mehr Ziele erreicht worden, als ursprünglich definiert. Nun bleibt abzuwarten, ob durch das breite Spektrum weiterer zu Verfügung stehender Schnittstellen, interessante Projekte mithilfe der Echtzeitübertragung umgesetzt werden.

Abbildungsverzeichnis

Listing

Literaturverzeichnis

Adobe Systems Inc. (2012). *Real-Time Messaging Protocol (RTMP) specification* . Abgerufen am 23. Mai 2016 von http://www.adobe.com/devnet/rtmp.html

Apple Inc. (2. Juni 2014). *Direct Access to Video Encoding and Decoding.* Abgerufen am 23. Mai 2016 von http://devstreaming.apple.com/videos/wwdc/2014/513xxhfudagscto/513/513_direct_access_to_media_encoding_and_decoding.pdf

Balachandran, A., Sekar, V., Akella, A., Seshan, S., Stoica, I., & Zhang, H. (29-30. Oktober 2012). *A Quest for an Internet Video Quality-of-Experience Metric.* Abgerufen am 22. April 2016 von http://www.cs.cmu.edu/~xia/resources/Documents/Balachandran-hotnets2012.pdf

Diquet, A. (2014). *It Just (Net)works. The Truth About iOS' Multipeer Connectivity Framework.* Abgerufen am 30. März 2016 von https://nabla-c0d3.github.io/documents/BH_MultipeerConnectivity.pdf

DivX, LLC. (kein Datum). *Ein neuer Standard für digitales Video*. Abgerufen am 23. Mai 2016 von http://www.divx.com/de/was-ist-h264

Fischer, W. (2016). *Digitale Fernseh- und Hörfunktechnik in Theorie und Praxis MPEG-Quellcodierung und Multiplexbildung, analoge und digitale Hörfunkund Fernsehstandards, DVB, DAB/DAB+, ATSC, ISDB-T, DTMB, terrestrische, kabelgebundene und Satelliten- Übertragungstechnik, Messtechnik.* Berlin, Heidelberg: Springer Vieweg.

Harrison, K. (6. Juni 2010). *useyourloaf.com*. Abgerufen am 23. Mai 2016 von Delegation or Notification: http://useyourloaf.com/blog/delegation-or-notification/

Kang, J., Burza, M., & Stok, P. (2006). Adaptive Streaming of Combined Audio/Video Content over Wireless Networks. In *Autonomic Management of Mobile Multimedia Services: 9th IFIP/IEEE International Conference on Management of Multimedia and Mobile Networks and Services, MMNS 2006, Dublin, Ireland, October 25-27, 2006. Proceedings* (S. 13-24). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kappes, M. (2013). *Netzwerk- und Datensicherheit.* Wiesbaden: Springer Vieweg.

Luan, H., Kwong, K.-W., & Tsang, D. (2009). Adaptive topology formation for peer-to-peer video streaming. In *Peer-to-Peer Networking and Applications* (S. 186-207). Springer Science + Business Media.

Peter, M., Andreas, B., & Johannes, W. (2008). *Grundkurs Datenkommunikation : TCP/IP-basierte Kommunikation: Grundlagen,*

Konzepte und Standards. Wiesbaden: Vieweg + Teubner in GWV Fachverlage.

Plag, F., & Riempp, R. (2007). *Interaktive Videos im Internet mit Flash Konzeption und Produktion von Videos für das WWW.* Berlin: Springer.

Sack, H., & Meinel, C. (2009). Multimediale Daten und ihre Kodierung. In *Digitale Kommunikation* (S. 161-305). Berlin Heidelberg: Springer.

Schmitz, R., Kiefer, R., Maucher, J., Schulze, J., & Suchy, T. (2006). *Kompendium Medieninformatik.* Berlin, Heidelberg: Springer.

Schulzrinne, H., Rao, A., & Lanphier, R. (April 1998). Real Time Streaming Protocol (RTSP). (T. I. Society, Hrsg.) USA: https://tools.ietf.org/html/rfc2326.

Tanenbaum, A. S., & Wetherall, D. J. (2012). *Computernetzwerke* (5. Auflage Ausg.). München: Pearson.

Wenger, S., Hannuksela, M., Stockhammer, T., & Westerlud, M. (Februar 2005). RTP Payload Format for H.264 Video. (T. I. Society, Hrsg.) USA: https://tools.ietf.org/html/rfc3984.

Eigenständigkeitserklärung

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich oder dem Sinn nach entnommenen Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Hamburg den, 31.05.2016 (Jens Woltering)