Möglichkeiten zum Echtzeit-Video-Streaming im Web

Mara Schulke *(Matrikel-Nr. 20215853)*

Technische Hochschule Brandenburg

B.Sc. Medieninformatik

Computergrafik

Wintersemester 2021

betreut durch Prof. Dr. rer. nat. Reiner Creutzburg

6. Juni 2022

Zusammenfassung

Durch die steigende dezentralisierung der Welt gewinnt die Echtzeit-Übertragung von Daten exponentiell an Relevanz. Die vorliegende Arbeit beschäftigt sich mit verschiedenen Protokollen zur Lösung dieses Problems und führt eine experimentelle Implementierung eines *Peer-To-Peer* Kamera-Systems durch und wertet diese anschließend hinsichtlich der Qualität der Echtzeit-Übertragung, verbrauchten Ressourcen und ihres Implementierungsaufwandes aus.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung / Motivation

Mit der zunehmenden dezentralisierung und der damit einhergehenden Vernetzung der Arbeitswelt in den letzten Jahren [**?**] werden auch digitale Meeting-Systeme immer relevanter und müssen immer mehr Nutzer in Echtzeit mit einander verbinden um einen reibungslosen Arbeitsalltag zu gewährleisten. Zu begin der Corona-Pandemie stieg der Netzwerktraffic für *Voice over IP* (kurz *VoIP*) und Videokonferenzen im März 2020 um 210%-285% [**?**].

Damit die Konferenzsysteme ihren Einsatzzweck erfüllen können, ist es erforderlich, dass eine ausreichend gute *Quality of Service (QoS)* gegeben ist.

Aber wie können skalierbare Meeting-Systeme realisiert werden ohne große Datenmengen über einen zentralisierten Streaming-Server zu schicken der diese an alle Teilnehmer weiterleitet? Die Entwicklungen der letzten Jahre deuten immer mehr darauf hin, dass *Peer-To-Peer* basierte Lösungensansätze aufgrund der besseren Performance und Skalierbarkeit, in der Regel die bessere Wahl darstellen [**?**]. *Peer-To-Peer*-Architekturen halten einen unvorhersehbaren Anstieg in der Nutzung deutlich besser aus, da die Echtzeit-Datenverarbeitung nicht bei dem Konferenzsystem-Anbieter stattfindet, sondern bei den Teilnehmern.

Natürlich spielen zur Auswahl der Architektur noch weitere Parameter eine wichtige Rolle (z. B. die maximale Bandbreite und Rechenleistung der Endgeräte), aber mit immer größer werdenden Heimnetz-Leitungen und zunehmender Rechenleistung der Endgeräte stellt dies mittlerweile für einfache Videokonferenzen kein Problem mehr da. [**?**]

Die Problematik der Echtzeit-Kommunikation im Web beschäftigt auch das *W3C* seit 2011 im Zuge der Standardisierung des seit diesem Jahr zum Web-Standard erklärten Protokoll *WebRTC*. [**?**]

Es ist also immer noch eine sehr aktuelle Thematik in der Informatik Echtzeit- oder Nahe-Zu-Echtzeit-Kommunikation zuverlässig zu bewältigen.

Die Aufgabe dieser Arbeit soll sein, einen Überblick über den Stand der Architekturmuster, Protokolle und möglicher Problematiken bei der Implementierung von eigenen Echtzeit-Video-Streaming-Diensten geben. Dafür wird im Rahmen eines Experiments ein Kamera-System implementiert. Die Herangehensweise und Ansätze für die Implementierung werden erläutert und anschließend wird das Ergebnis hinsichtlich der *QoS* im Zusammenhang mit dem Arbeitsaufwand ausgewertet.

2 Historie der Echzeit-Übertragung

Um zu verstehen wie sich die Protokolle hin zum heutigen Stand entwickelt haben, ist es besonders interessant nachzuvollziehen wie die ersten Schritte der IEFT oder auch *Internet Engineering Task Force* bezüglich der Echtzeit-Kommunikation aussahen, welche Probleme erkannt und behoben wurden und welche Protokolle heute der Standard sind.

2.1 1996: RTP & RTCP

Am weitesten reicht das *Real-Time Transport Protocol* oder auch *RTP* zurück. Es wurde erstmals 1996 von der IEFT standardisiert und stellt seit dem einen Grundbaustein der datenformatsaganostischen Echtzeit-Übertragung da. Es kann für diverse Echtzeit-Übertragungs-Problematiken dienlich sein, da jegliche Binärdaten verschickt werden können; Somit gibt es keinen "Lock-In" auf bestimmte Audio- oder Video-Codecs.

Realtime-Transport-Control-Protocol oder auch *RTCP* ist das mit *RTP* einhergehende Kontrollprotokoll. Es wird primär dazu verwendet, die Übertragungsparameter der Sender zu beeinflussen – z. B. durch ein Feedback zur Übertragungsqualität oder ein Abmelden der Session.

Desweiteren bietet es eine persistente ID für die *RTP*-Mitglieder, die über Programm-Neustarts hinweg zur Identifikation von Mitgliedern und der zuordnung von Datenströmen verwendet werden können. [**?**]

RTP und RTCP siedeln sich im TCP/IP-Stack über *UDP* an. [**?**]

RTP fügt wichtige Informationen zu UDP-Datagrammen hinzu: Im wesentlichen eine Sequenznummer um die Sende-Reihenfolge zu codieren und einen Payload-Type, der den Codec des Segments angibt. Somit kann auch bei nicht sequenziell übertragenden Datagrammen die Ursprüngliche Reihenfolge rekonstruiert werden und es können bei verlorenen Segmenten Interpolationsalgorithmen verwendet werden. Der Payload-Type ist essenziell um ohne Session-Aushandlung zu kommunizieren wie der Empfänger die Daten zu decodieren hat, um eine sinnvolle Nachricht zu erhalten.

Die ersten Version aus 1996 wurde 2003 durch die überarbeitet Version, spezifiziert in dem RFC3550, abgelöst. [**?**]

Eine wichtige Voraussetzung zur Verwendung von *RTP* ist ein externer *Signaling-Server*, den alle Beteiligten zur Session-Aushandlung verwenden. Ein Standard hierfür ist im *RTP-Framework* selbst nicht definiert, eine beliebte Wahl für ein Protokoll zur Session-Aushandlung ist allerdings das *Session Initiation Protokoll* oder kurz *SIP*. [**?**, **?**]

2.2 2004: SRTP

Im März 2004 wurden *SRTP* und *SRTCP* vorgestellt – die verschlüsselten Version von *RTP* bzw. *RTCP*. Diese bauen weitestgehend auf dem *Advanced Encryption Standard* (kurz. *AES*) auf. [**?**] Der RFC beschäftigt sich weitestgehend mit den kryptografischen Aspekten des Protokolls und ist in dieser Hinsicht für dies vorliegende Arbeit nur bedingt interessant. Dennoch bildet *SRTP* eine wichtige Grundlage für sichere Echtzeitübertragung und wird im WebRTC-Standard verwendet.

Eine weitverbreitete und offene Implementierung für *SRTP / SRTCP* wird von der US-Amerikanischen Telekomunikations-Gesellschaft Cisco bereitgestellt. Diese hat den Namen *libsrtp* und ist öffentlich auf der Entwickler-Plattform GitHub öffentlich einsehbar (siehe `https://github.com/cisco/libsrtp`).

2.3 2005: RTMP

RTMP ist ein von der Firma *Adobe Systems Incorporated* spezifiziertes Protokoll zur Echtzeit-Übertragung von Multimedia-Streams. *RTMP* wurde laut Spezifikation [**?**] entwickelt um über dem Transport-Protokoll *TCP* verwendet zu werden.

Das Protokoll wurde dazu entwickelt um im Kontext eines Flash-Players verwendet zu werden. Dies führt dazu, dass es heutzutage nurnoch bedingt in Browsern Anwendung findet, da mittlerweile viele Browser ihren Flash-Player-Support eingestellt haben. [**?**]

Außerdem spricht die hohe Latenz von bis zu 30 Sekunden, die durch die Verwendung von *TCP* zustandekommt [**?**] gegen den Einsatz von *RTMP* in einem Echtzeit-Übertragungs-Kontext.

Desweiteren gibt nach [**?**] noch weitere Protokollvarianten:

- *RTMPT* (Tunneled) verwendet *HTTP* bzw. *HTTPS* als Transport-Protokoll.
- *RTMPE* (Encrypted)
- *RTMPTE* (Tunneled und Encrypted)
- *RTMPS* verwendet *SSL*
- *RTMFP* verwendet *UDP* als Transport-Protokoll anstelle von *TCP*

Auch wenn *RTMP* aufgrund der hohen Latenz für Videokonferenzen ungeeignet ist, findet es heutzutage immernoch Anwendungen in unidirektionalen Datenübertragungen an eine große Anzahl von Teilnehmern, da es sich gut skalieren lässt. [**?**]

2.4 2006: DTLS

Das *Datagram-Transport-Layer-Security*-Protokoll bietet starke Sicherheitsgarantien (equivalent zu *TLS*) für Datagram basierte Protokolle wie z. B. *UDP*. [**?**] Es bildet heutzutage die Grundlage für sichere, verbindungslose Kommunikation.

2.5 2010: WebRTC

WebRTC ist eine Peer-To-Peer-Technologie die über mehrere Protokolle und Audio- und Video-Codecs performante und generische Echtzeit-Kommunikations-Kanäle zwischen Nutzern (oder auch *Peers*) realisiert. Dafür stellt eine *WebRTC*-Implementierung durch einen komplexen Signaling-Ablauf eine direkte Verbindung zu anderen Endgeräten her. [**?**]

2.5.1 Signaling

Während der Signaling-Phase generiert zunächst einer der Teilnehmer eine *SDP-Offer*. Diese wird über einen, nicht näher im Standard beschreibenen, Signaling-Channel an den anderen Teilnehmer gesendet. Dieser verarbeitet die *SDP-Offer* und generiert eine *SDP-Answer* und sendet diese über den gleichen Signaling-Channel an den Absender zurück.

Nach erfolgreichem austauschen der *SDP*-Nachrichten, verwendet generiert jeder Teilnehmer mithilfe des *Interactive Connectivity Establishment*-Frameworks *ICE*-Candidates und versendet sie über den Signaling-Channel. Diese werden verwendet um einen eindeutigen Weg durch das Internet zum anderen Teilnehmer zu finden. Dieser Vorgang ist notwendig, da durch die Einführung *Network-Address-Translation*, in den meisten Netzwerken Endgeräte keine globale eindeutige IPv4-Adresse haben. [**?**]

2.5.2 Übertragung

Sobald die Signaling-Phase erfolgreich beendet wurde und beide Teilnehmer sich im Internet gefunden haben, wird eine direkte *SRTP*-Verbindung hergestellt. [**?**] Diese wird über *DTLS* verschlüsselt. [**?**]

Anschließend können die zu übertragenden Daten in den entsprechenden Codecs über diese Verbindung versendet werden.

3 Architekturmuster

In diesem Kapitel werden zwei der bekanntesten Architekturmuster in der Echtzeit-Übertragung vorgestellt und verglichen. Es geht primär um die verteilte Peer-To-Peer Architektur und die zentralisierte Relay / Broadcast Architektur.

3.1 Peer-To-Peer

Auf Grund der hohen Anforderungen an möglichst niedrige Übertragungslatenzen bietet eine Peer-To-Peer-Architektur klare Vorteile durch den stark verkürzten Weg, den die Pakete zurücklegen müssen bis sie bei dem Empfänger ankommen.

Deutlich komplizierter wird allerdings die aushandlung bzw. initialisierung einer Verbindung zwischen zwei peers, da diese sich nicht wie bei der typischen Client-Server-Architektur eine fixe adresse zur verbindung haben. Dieses Problem wird typischerweise über einen Signaling-Server gelöst.

3.1.1 Signaling-Server

Ein Signaling-Server ist allen Teilnehmern bekannt und dient als Kommunikationsplattform. Somit können sich zwei, sich initial unbekannte Teilnehmer, gegenseitig vorstellen.

Signaling-Server sind interessanterweise keine feste Anforderung für Peer-To-Peer-Architekturen. Wenn alle Teilnehmer statische Addressen hätten, könnten sie auch Out-Of-Band ihre IP-Adressen und ggf. weitere Details zum Verbindungsaufbau austauschen, um eine Verbindung aufzubauen.

Der Signaling-Server kann weiterhin noch nach Verbindungsaufbau dazu verwendet werden um die bestehende Verbindung zu optimieren. Peers können neue ICE-Candidates vorschlagen um die Übertragung an neue Gegebenheiten anzupassen.

3.1.2 Theoretische Optimierung durch IP-Multicast

Je mehr Nutzer / Endgeräte an einer Peer-To-Peer-Übertragung teilnehmen, desto größer wird die Belastung der Bandbreite bei den einzelnen Teilnehmern. Jedes Paket muss nicht einmal, sondern N-mal verschickt werden (wobei N die Anzahl der Teilnehmer ist). Dies kann bei labilen oder einfach schwachen Netzwerken entweder zur Verminderung der Übertragungsqualität führen, oder in besonders schlimmen Fällen sogar das restliche Netzwerk eines einzelnen Teilnehmers negativ beeinflussen, da dieser die meiste Bandbreite für die wiederholte Übertragung gleicher Pakete verwendet.

Eine theoretische Lösung für dieses Problem, ist die Verwendung von IP-Multicast-Adressen. Diese erlauben es einem Gerät ein IP-Paket einmalig zu übertragen, und dieses von Multicast-Routern im Internet multiplizieren zu lassen.

Dieser Ansatz hat klare Vorteile: Das gesamte - lokale & globale – Netzwerk wird geschont. So würde die Anzahl möglicher Teilnehmer deutlich steigen, da ein Netzwerk-Bottleneck erst bei einem zu großen Download-Volumen des empfangenen Contents eintreten würde.

Leider ist dies aus verschiedenen Gründen bisher nur in der Theorie möglich. Der Hauptgrund ist die mangelnde Unterstützung für IPv4-Multicast-Pakete bei einem Großteil der Router. Desweiteren würden Heimnetzwerke somit in Backbonenetzwerken deutlich mehr Netzwerklast auslösen können als vorgesehen ist.

3.2 Relay

Die zentralisierte Architektur ist eine der besten Alternativen zum Peer-To-Peer-Ansatz. Sie wird bei diversen Streaming-Plattformen erfolgreich bei einer großen Anzahl von Nutzern betrieben. [**?**, **?**] Der wohl grundlegenste Unterschied liegt darin, dass zwischen den Teilnehmern eine zentralisierte Instanz geschaltet ist, die den Stream vervielfacht um so die Netzwerkauslastung bei einzelnen Teilnehmern zu minimieren.

Diese Architektur lässt im Gegensatz zu Peer-To-Peer-Architekturen deutlich mehr Teilnehmer zu, da nur der Relay-Server die Last skalieren muss und nicht ein Teilnehmer. Wenn ein Live-Stream auf einer Streaming-Plattform beispielsweise einige Hundert Zuschauer hat und dies via Peer-To-Peer laufen würde, wäre dies bereits eine Größe bei der typische Endgeräte und Heimnetzwerke definitiv überlastet wären.

Folglich finden zentralisierte Architekturen auch bei Mobilgeräte anwendung [**?**, **?**], da Mobilgeräte oft (abgesehen von einer stärker limiterten Bandbreite) lediglich über ein fixes Datenvolumen verfügen.

Zu den Vorteilen zählt die einfache Skalierbarkeit, die (weitestgehende) Unabhängigkeit von Teilnehmer Netzwerken und die Kontrolle über den verteilten Datenstrom.

Die Nachteile hingegen zeichnen sich für die Teilnehmer in einer erhöhten Latenz ab, was ein Relay für Live-Videokonferenzen in der Regel uninteressanter macht. Desweiteren fallen Hosting-Kosten in direkter Abhängig von der Nutzeranzahl ab, da jeder Datenstrom vom Server verarbeitet wird.

4 Implementierung eines Kamera-Live-Streams

In diesem Kapitel wird ein experimentelles Kamera-System entwickelt. Dieses baut auf einem WebSocket-Secure basierten Signaling-Protokoll, *WebRTC* und *GStreamer* auf.

4.1 Ziel

Ziel dieses Experiments ist es eine möglichst einfach ein funktionierendes Kamera-System zu entwickeln und mögliche Probleme, benötigten Zeitaufwand etc. zu ermitteln. Zukünftig könnte noch ein Performance-Vergleich mit anderen Echtzeit-Systemen interessant sein.

Es gilt also die Frage zu klären, wie viel Arbeit ein simples, aber funktionierendes Software-Produkt das auf Echtzeit-Übertragung beruht benötigt und wie gut die Übertragungsqualität anschließend ist.

4.2 Umfang

Das zu implementierende Kamera-System wird lediglich eine einzige Kernfunktion aufweisen: Sobald die Kamera an ist, streamt sie via WebRTC, mit ausreichend FPS (mindestens 15 im durchschnitt) in ausreichender Qualität (720×480), ihre Aufnahmen an eine Benutzeroberfläche. Diese wird im Browser laufen, muss aber ausreichend Alternativen für andere Plattformen aufweisen (Nativ, Smartphone usw.). Die Benutzeroberfläche und die Kamera starten die Aushandlung des WebRTC-Streams über den Signaling-Server. D.h. sie versenden *SDP*-Nachrichten und tauschen ihre *ICE*-Candidates aus (siehe Abbildung **??**).

4.3 Architektur

Die geplante Architektur ist besteht lediglich aus 3 Elementen:

4.3.1 Kamera

Für dieses Experiment wurde ein Einplatinencomputer der Marke Raspberry PI in der 4. Version mit 8GB RAM – erweitert durch ein Drittanbieter-Kamera-Modul – verwendet. Dieser ist allerdings absolut austauschbar, da jeder Rechner der die u.g. Hauptanforderung erfüllt eine geeignete Umgebung darstellt – so kann die Kamera-Software auch auf einem Laptop ausgeführt werden, falls keine Hardware verfügbar ist. Dies wurde zur veranschaulichung mit einem ThinkPad T490 und einer aktuellen NixOS Installation getestet.

Die Hauptanforderung an die Hardware auf der die Kamera-Software läuft ist eine Linux-Installation (mit den entsprechenden installierten Paketen für die Abhangigkeiten) und eine angeschlossene Kamera die von Linux erkannt wird.

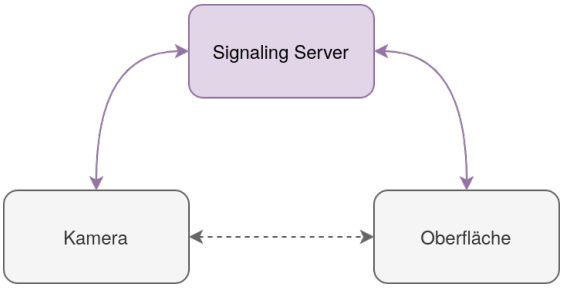

Abbildung 1: Signaling zwischen der Kamera und der Benutzeroberfläche.

Quelle: Schulke 2021, nach [**?**]

Auf der Einplatinencomputer ist das mitgelieferte Betriebssystem "Raspberry PI OS" und die entsprechenden Software-Abhängigkeiten (siehe **??**) installiert.

4.3.2 Signaling-Server

Der Signaling-Server ist ein WebSocket-Secure-Server, der zwei WebSocket-Verbindungen miteinander verknüpft. Er verwaltet sogenannte Räume. Ein Raum besteht aus 0 bis 2 Teilnehmer und dient dazu diese miteinander zu verbinden. Auf dem Server kann es mehrere Räume gleichzeitig geben – dadurch könnte dieser Signaling-Server theoretisch auch noch für weitere WebRTC-Anwendungen verwendet werden. Ein Raum hat eine ID, diese ist eine 256-Bit-Entropie. Teilnehmer werden durch eine 128-Bit-Entropie identifiziert.

So können sich zwei Teilnehmer durch ein Out-Of-Band kommuniziertes Secret (die Raum-ID) über diesen in Kontakt treten. Diese könnte beispielsweise, würde es sich bei der entwickelten Kamera um ein echtes Produkt handeln, bei der Herstellung generiert werden, ausgedruckt und neben die Kamera in die Verpackung gelegt werden.

4.3.3 Benutzeroberfläche

Die Benutzeroberfläche verbindet sich mit dem Signaling-Server und nimmt eine Raum-ID entgegen. Mit dieser Raum-ID wird dann auf dem Signaling-Server entweder ein neuer Raum erstellt, falls noch keiner mit der ID existiert, oder es wird dem bestehenden Raum beigetreten. Nach dem ein weiterer Teilnehmer beigetreten ist, fängt der unter **??** erklärte Aufbau eines WebRTC-Streams zu dem Nutzer an.

4.4 Signaling-Server

Als erstes wurde der Signaling-Server entwickelt, da dieser keine Abhängigkeiten an seine Teilnehmer hat. Die Kamera-Software und Benutzeroberfläche setzen beide jeweils den laufenden Signaling-Server vorraus um korrekt zu funktioneren.

Asynchronität

Eine logische Anforderungen an den Server ist das gleichzeitige verarbeiten mehrerer Verbindungen – ansonsten könnte immer nur ein Teilnehmer alleine mit dem Server Verbunden sein. Dies würde die Signaling-Funktionalität eines Raumes unbrauchbar machen.

Die asynchrone Programmierung ist ein Konzept zur Lösung dieser Problemklasse. Da langlebige Verbindungen in der Regel einen großteil der Zeit ungenutzt sind, ist es naheliegend Verbindungen ohne neue Ereignisse keine CPU-Zeit zu geben um andere Verbindungen in der Zeit abzuarbeiten, in der auf Ereignisse gewartet wird.

4.4.1 Verbindungsaufbau

Um eine WebSocket-Secure-Verbindung aufzubauen, muss zu erst eine TCP- und dann darüber eine TLS-Verbindung zu dem Teilnehmer aufgebaut werden. Über diese wird dann zu erst in HTTP kommuniziert und nach einer erfolgreichen Nachricht des Servers mit dem Status-Code 101 (Switching Protocols) wird die über den gleichen Transport-Weg (TLS) nach dem WebSocket-Protokoll kommuniziert.

4.4.2 Spezifikation des Signaling-Protokolls

Das Signaling-Protokoll ist in 2 Nachrichten-Typen unterteilt: Server-Nachrichten und Peer-Nachrichten. Teilnehmer dürfen nur Peer-Nachrichten senden, ansonsten wird die Verbindung aufgrund eines Protokoll-Verstoßes geschlossen. Der Server verschickt nur Server-Nachrichten.

Alle Protokoll-Nachrichten werden in JSON kodiert und dann über den WebSocket-Nachrichten-Typ Binär an den Empfänger geschickt werden.

Server-Nachrichten

- Hello {Teilnehmer-ID}
- Joined
- Error
- Room/Join {Teilnehmer-ID}
- Room/Leave {Teilnehmer-ID}
- Room/Signal {Signal}

Peer-Nachrichten

- JoinOrCreate {Raum-ID}
- Signal {Signal}

Nach einer erfolgreich aufgebauten Verbindung mit dem Server schickt dieser ein *Hello* mit der zugewiesenen Teilnehmer-ID. Darauf hin muss der Teilnehmer ein *JoinOrCreate* senden um einem Raum beizutreten. Bevor der Teilnehmer die Nachricht *Joined* vom Server erhält, darf dieser keine Signale verschicken.

Eine *Room/Join* Nachricht wird an ggf. andere Teilnehmer im Raum verschickt, sobald ein Teilnehmer diesen betritt. Diese Nachricht kann auf dem Teilnehmer als Anlass genutzt werden eine SDP-Offer zu erstellen, da nun sicher ist, das ein anderer Teilnehmer im Raum ist. Wenn beide Teilnehmer sich an dieses Schema halten, schickt immer der Teilnehmer, der sich zu erst Verbunden hat die Einladung und der andere die Answer.

Analog wird eine *Room/Leave* Nachricht verschickt wenn ein Teilnehmer einen Raum verlässt.

Die *Room/Signal* Nachricht wird an den Empfänger eines Signals geschickt, nachdem der Sender des Signals eine *Signal* Nachricht an den Server geschickt hat.

Siehe Abbildung **??** für einen kompletten WebRTC-Verbindungsaufbau.

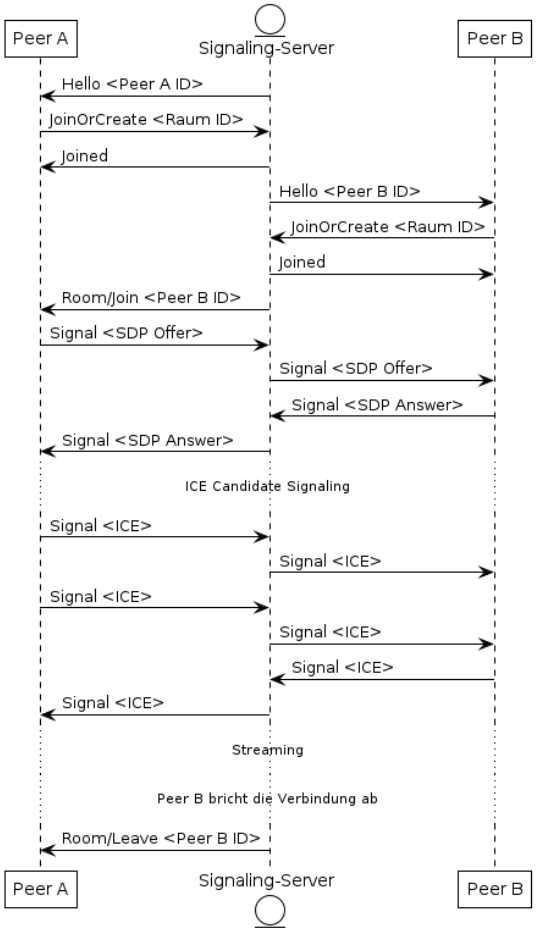

Abbildung 2: Anwendungsbeispiel des Singaling-Protokolls

Quelle: Schulke 2021

4.4.3 Raum-Verwaltung

Die Raum-Verwaltung fällt unter anderem in die Kategorie der Synchronisationsprobleme. Es gibt unter Umständen n Teilnehmer-Verbindungen, die den Status von einem Raum erfragen wollen, und ggf. einen Anlegen möchten. Dies soll für alle Verbindungen öffentlich geschehen, Teilnehmer dürfen sich aber dabei nicht gegenseitig überschreiben.

Um zwei asynchron laufende Programmteile zum sequenziellen Zugriff auf gemeinsamen Speicher zu zwingen, gibt es Möglichkeit einen Semaphor bzw. Mutex zu verwenden. Dieser blockiert den Teil des Programms, der gerade auf den gemeinsamen Speicher zugreifen möchte solange, bis ein ggf. anderer Zugriff beendet ist. Um Deadlocks bzw. Inperformante Code-Abschnitte zu vermeiden, ist es ratsam einen Mutex nicht über länger andauernde Operationen hinweg zu locken, da sonst für diese Zeit alle anderen Teile des Programms, die gerade auf den Speicher zugreifen möchten blockiert sind.

Als konkrete Lösung für das bestehende Problem bietet sich eine HashMap, deren zugriff von

einem Mutex kontrolliert wird, an. Jede Teilnehmer-Verbindung, die aktuelle Raum-Informationen braucht (was hauptsächlich bei dem erstellen / betreten von Räumen der Fall ist), muss nun vorher erst den mutex locken.

4.4.4 Kommunikation zwischen mehreren Teilnehmer-Verbindungen

Bis zu diesem Punkt ist der Aufbau der Verbindung und das zu implementierende Protokoll klar, allerdings fehlt noch das Verknüpfen von Teilnehmer-Verbindungen um Signale weiterzuleiten.

Da bereits Räume verwaltet werden, liegt es nahe dort einen Kommunikationskanal einzubetten. Eine Implementierung für diesen Kommunikationskanal stellen z.B. sogenannte Channels dar. Diese erlauben Inter-Thread- (und im asynchronen Kontext: Inter-Task-) Kommunikation über ein einfaches Sender-Empfänger-Prinzip. Ein Channel ist unterteilt in eben diese beiden Teile: Der Sender darf Nachrichten schreiben, der Empfänger darf sie aus dem Channel lesen.

In diesem Fall ist allerdings zusätzlich noch bidirektionale Kommunikation gefragt, da beide Teilnehmer die Signale des jeweils anderen erhalten sollen. Dafür bieten sich sogenannte Broadcast-Channel an. Sie sind dafür ausgelegt, dass von mehreren Stellen in diese geschrieben und gelesen wird. Sie funktioneren analog zu dem Broadcast aus der Netzwerktechnik (siehe Abbildung **??**).

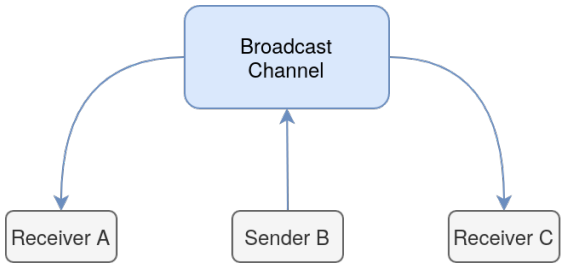

Abbildung 3: Broadcast-Channel Konzept

Quelle: Schulke 2021

Somit kriegt jeder Teilnehmer bei betreten eines Raumes Zugriff auf einen Sender und einen Empfänger dieses Broadcast-Channels. In diesen schreibt er bei erhalt einer Peer-Nachricht vom Typ *Signal* das Signal (siehe Abbildung **??**). Eine Teilnehmer-Verbindung die gerade keine CPU-Zeit erhält, da ansonsten keine Ereignisse aufgetreten sind, wird wieder aktiv sobald eine neue Nachricht über den Channel da ist. Diese kann dann als Server-Nachricht vom Typ *Room/Signal* mit dem Signal aus dem Channel an den Empfänger gesendet werden.

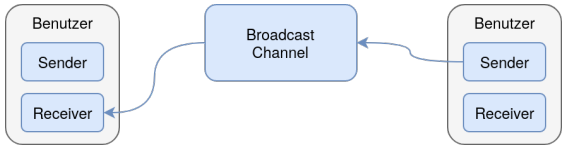

Abbildung 4: Kommunikation von Nutzern über einen Broadcast-Channel

Quelle: Schulke 2021

4.5 Benutzeroberfläche

Um den Signaling-Server für ein erstes nutzbares Zwischenergebnis zu verwenden, wurde eine einfache browserbasierte Benutzeroberfläche implementiert, die analog zu sehr bekannten Videokonferenz-Systemen wie z.B. Zoom oder Google Meet funktioniert. Bei diesem Experiment wurde der Fokus primär auf die Bildübertragung gelegt – eine Audioübertragung könnte aber ohne großen Aufwand implementiert werden.

Ein Benutzer muss über diese Benutzeroberfläche einem beliebigen Raum auf dem Signaling-Server beitreten können. Sobald ein weiterer Benutzer beitritt soll der erste eine Nachricht erhalten, dass Jemand seinem Raum beigetreten ist. Letztendlich soll die Aushandlung der Video-Übertragung zwischen den beiden Benutzern ohne weitere Interaktion des Benutzers stattfinden.

4.5.1 WebSocket-Verbindung

Als Vorkehrungen für die singalisierungs Kommunikation mit dem Signaling-Server muss eine WebSocket-Secure Verbindung mit dem Server aufgebaut werden. Dies gelingt ebenfalls über eine Web-API.

Nun meldet sich der Server nach dem Verbindungsaufbau als erstes mit der *Hello*-Nachricht und weist uns eine ID zu. Nach dieser ersten Nachricht können wir die eigentliche Anwendung starten.

4.5.2 Erstellen / Beitritt eines Raumes

Um festzustellen mit wem sich der Benutzer verbinden möchte, muss dieser eine Raum-Id angeben. Diese ist als 64 Zeichen langen Hex-String einzugeben.

Nach erfolgreicher Eingabe einer Raum-Id wird diese mit der Nachricht *JoinOrCreate* an den Server geschickt, dieser verarbeitet die Anfrage und schickt dem Benutzer eine *Joined*-Nachricht. Ab diesem Zeitpunkt wird eine *RTCPeerConnection* erstellt, da ggf. bereits ein anderer Nutzer im Raum war, der dem neu dazugekommenen eine SDP-Offer schicken wird. Um diese Zeitnah zu verarbeiten, findet die Initialisierung vor der ersten Signalisierungs-Nachricht statt.

4.5.3 Verbindungsaufbau

Moderne Browser wie Firefox, Chrome, Safari etc. stellen eine Implementierung des WebRTC-Standards bereit (siehe Abbildung **??**), die durch eine übersichtliche API leicht zu integrieren ist.

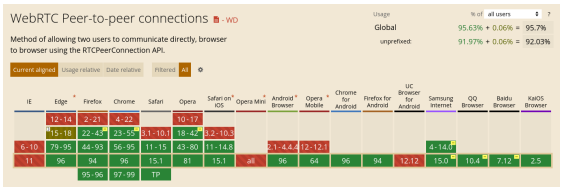

Abbildung 5: WebRTC Support-Matrix

Quelle: [**?**]

Um eine vollständige WebRTC-Verbindung aufzubauen ist bei beiden Nutzern eine RTCPeer-Connection aufzubauen. Auf einer RTCPeerConnection ist eine Event-basierte API verfügbar.

(Siehe tabelle)

Insbesondere sind für den Verbindungsaufbau die Events onnegotiationneeded und onicecandidate von Interesse. Sobald eine Verbindung erstellt wurde und ein lokaler Stream zu dieser hinzugefügt wurde, wird das onnegotiationneeded Event ausgelöst. An diesem Punkt ist nun eine eigene SDP-Offer zu erstellen und über den singalisierungs Server an den Empfänger zu versenden. Der Empfänger verarbeitet diese anschließend und antwortet mit einer SDP-Answer. Nach dem beide Teilnehmer die SDP-Nachrichten füe ihren Kommunikationspartner und sich selbst gesetzt haben (siehe setRemoteDescription und setLocalDescription), werden vom Browser onicecandidate-Events ausgelöst um eine direkte Verbindung zwischen den beiden Browsern auszuhandeln.

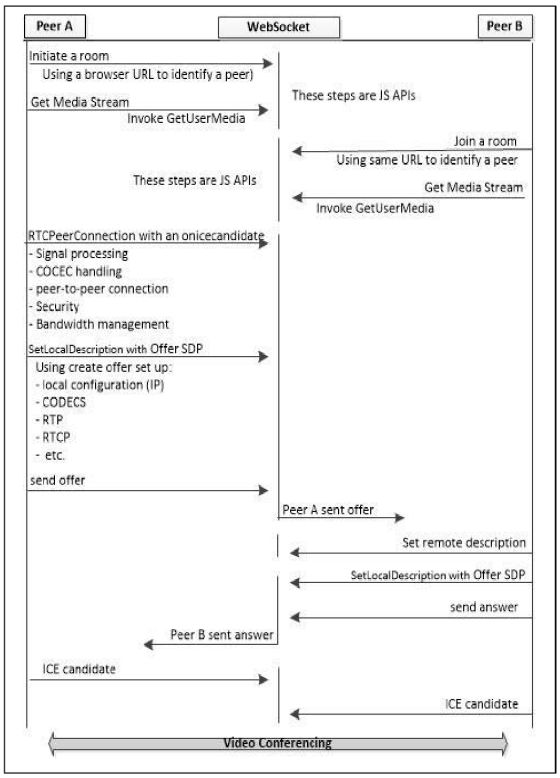

Abbildung 6: WebRTC-Verbindungsaufbau

Quelle: [?]

Nach einem leeren ICE-Kandidaten ist die aushandlung abgeschlossen und der Stream kann nun konsumiert werden.

4.5.4 Verbindungsabbau

Nach dem der Signalisierungs-Server eine *Room/Leave*-Nachricht gesendet hat, kann die RTCPeerConnection ohne negative Auswirkungen abgebaut werden. In der Benutzeroberfläche wird die Anzeige des Video-Streams des anderen Teilnehmers zurückgesetzt und die RTCPeerConnection geschlossen. Somit ist die Benutzeroberfläche wieder bereit einem weiteren Raum beizutreten

und erneut eine WebRTC Verbindung herzustellen.

4.6 Kamera

4.6.1 GStreamer

Anders als in einer Browser-Umgebung bietet Linux abgesehen von der bereitstellung von Hardware-Treibern keine guten Abstraktionen für die Verwendung einer Kamera und automatischer Bildverarbeitung. Dies ist in einer nativen Umgebung die Aufgabe von weiteren Programmen und Bibliotheken.

Hier wird das weitverbreitete Multimedia-Framework GStreamer relevant. Dieses abstrahiert sämtliche Zugriffe auf medienbezogene Hardware (in diesem Fall die Kamera), beherscht die meistgenutzten Media-Codecs für Audio und Video und kann über Plugins mit weiteren Funktionalitäten erweitert werden (siehe Abbildung **??**).

GStreamer ist dafür ausgelegt das erstellen von Audio und/oder Video basierten Anwendung zu erleichtern. Es unterstützt weiterhin, durch die Pipeline-Architektur, jegliche Art von Datenströmen.

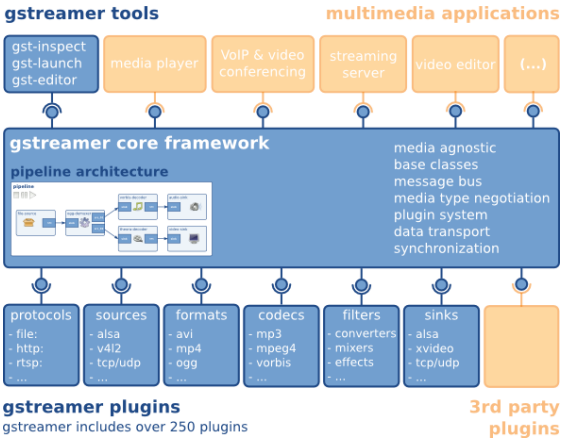

Abbildung 7: GStreamer Architektur

Quelle: [**?**]

Zu den Kern-Features von GStreamer gehören eine API für diverse Multimedia-Anwendungen, eine Pipeline- und Plugin-Architektur, Mechanismen für die Aushandlung von Datenformaten zwischen Pipeline-Elementen, Synchronisation von Datenströmen. (siehe Abbildung **??**, [**?**])

Das Framework baut auf einem ausgereiften Plugin-System mit über 250 Plugins auf, welches den Codec-Support erweitert und weitere Komponenten zur Erstellung von komplexeren Pipelines bereitstellt.

Diese Plugins können in die folgenden Kategorien unterteilt werden:

- Protokolle
- Datenquellen
- Formate

- Codecs
- Filter & Effekte
- Ausgabe

Siehe [?]

Für dieses Experiment ist das GStreamer-Plugin für WebRTC besonders von bedeutung.
Somit muss die folgende Liste an Abhängigkeiten installiert werden, damit die Kamera-Software korrekt funktioniert:

- libgstreamer-gl1.0-0
- libgstreamer-opencv1.0-0
- libgstreamer-plugins-bad1.0-0
- libgstreamer-plugins-base1.0-0
- libgstreamer1.0-0
- libgstrtspserver-1.0-0

4.6.2 Architektur der Kamera-Software

Die Kamera-Software ist grundsätzlich in 2 Zustände unterteilt, die sich endlos abwechseln:

1. Verbindungsaufbau zum Signalisierungsserver
2. Streaming via WebRTC

Im Regelfall dauert der 1. Schritt einige Millisekunden. Nach dem die Verbindung zum Signalisierungsserver korrekt aufgebaut wurde (und die Kamera dem festgelegten Raum beitreten konnte) wird eine GStreamer-Pipeline initialisiert und auf den Beitritt eines Benutzers in den Raum erwartet. Nach dem die *Room/Join*-Nachricht eingetroffen wird die Pipeline auf Playing gesetzt.

GStreamer unterstützt nach erfolgreicher Installation oben genannter Plugins eine kompatible API zu der der Browser. Somit läuft der gesamte WebRTC-Spezifische Verbindungsaufbau nach der exakt gleichen Logik ab wie unter **??** erläutert.

Bei der Einbindung von *GStreamer* wurde [?] als Referenz verwendet.

4.6.3 Vorbereitung der Laufzeitumgebung

Sobald die Kamera-Software auf einem herkömlichen Desktop-System lauffähig ist, gilt es die Laufzeitumgebung auf der Kamera-Hardware vorzubereiten.

Hierfür muss zunächst das Betriebssystem der Hardware, Raspberry PI OS, installiert werden. Anschließend sind die unter **??** aufgezählten Software-Abhängigkeiten zu installieren. Sobald diese vorhanden sind, kann die Kamera-Software auf anderen Desktop-Systemen für die entsprechende System-Architektur (*armv7-unknown-linux-gnueabihf*) kompiliert werden und anschließend auf den Einplatinencomputer übertragen werden.

4.6.4 Automatischer Start der Kamera-Software

Da die Kamera-Hardware nicht permanent Stromzufuhr haben wird, ist es hilfreich einen Betriebssystem-Service für anzulegen, der für die automatische Ausführung der Software verantwortlich ist. Somit muss nicht bei jedem Neustart händisch die Software ausgeführt werden.

Unter Raspberry PI OS übernimmt diese aufgabe der Init-Service Systemd.

5 Auswertung

Der Signaling-Server hatte den größten Implementierungsaufwand der 3 Komponenten, hat sich dafür aber als sehr stabil und vielseitig einsetzbar erwiesen. Hinsichtlich der Performance ist die Implementierung ebenfalls ein Erfolg. Auf einem System mit einem Intel i7-8565U-Chip übersteigt der Server mit 50 Benutzern nicht annähernd 1% der CPU-Kapazität. Falls nicht eine große Anzahl an Nutzern gleichzeitig viele Signale über diesen Server schickt, pendelt sich dieser aufgrund seiner asynchronen Architektur bei einer CPU-Last von 0.0% ein, da dieser nur aktiv wird, wenn auch Nachrichten eintreffen.

Die Benutzeroberfläche ist rein funktional geblieben und bietet kein gutes Nutzererlebnis. Abgesehen von der schlechten Nutzbarkeit, ist die erwartete Funktionalität vollständig und mit angemessenem Aufwand implementierbar gewesen. Die Browser-APIs bieten eine exzellente Grundlage um reale Anwendungen in kurzer Zeit mit einer Live-Stream-Funktionalität zu entwickeln. Desweiteren funtkioniert auch die Videoübertragung vom Browser über eine WebRTC-Verbindung sehr gut, die Bildübertragungsraten waren in der Regel über den angestrebten 15 Bildern pro Sekunde.

Die Implementierung einer Kamera-Software ist im Vergleich zu den beiden anderen Komponenten verhältnismäßig aufwändig gewesen. GStreamer bringt ebenfalls vergleichbare WebRTC-Abstraktionen mit sich, wie moderne Browser, allerdings ist GStreamer erheblich instabiler und unzuverlässiger was den Stream-Aufbau angeht als die Browser. Auch die Bildübertragungsrate ist leider erst bei sehr niedriger Qualität (160px×90px) nah an dem angestrebten Wert. Sobald die Qualität diese überschritt, nimmt die Bildübertragungsrate drastisch ab und somit wird der Stream ab einer gewissen Bildqualität unbrauchbar. Dies könnte ggf. auf das 32-Bit-Betriebssystem der Hardware zurückzuführen zu sein.

6 Nächste Schritte

Einer der nächsten Schritte um dieses Experiment weiterzuführen wäre definitiv die Fehlersuche an der Kamera-Software um die Übertragungsqualität zu steigern.

Außerdem sollte die Kamera-Software nicht erneut dem Raum auf dem Signaling-Server beitreten sobald der Stream abbricht, dies zu beheben könnte eine (auch wenn nur geringe) Netzwerkentlastung bringen und würde zusätzlich den Signaling-Server schonen.

Auch die Sicherheit von Räumen könnte durch ausgreiftere Authentifizierungsverfahren drastisch erhöht werden.

Als letzte Verbesserung wäre eine Implementierung von Audio-Übertragung von der Kamera zu der Benutzeroberfläche interessant um auch diesen Aspekt der Echtzeitübertragung im Web auszustesten.

Anhang

signaling/main.rs signaling/lib.rs signaling/tls.rs signaling/rooms.rs signaling/state.rs signaling/signals.rs
camera/main.rs camera/webrtc.rs camera/signaling.rs interface/index.html

Abbildung 8: Rapsberry PI 4

Abbildung 9: Camera Modul

Abbildung 10: Installierte Kamera