

betreut durch Prof. Dr. rer. nat. Reiner Creutzburg
Wintersemester 2021
Abgabetermin 5. Dezember 2021

Ansätze zum Echtzeit-Video-Streaming im Web

Technische Hochschule Brandenburg
B.Sc. Medieninformatik
Computergrafik

5. Dezember 2021

Zusammenfassung

Das abstract schreibe ich zu letzt!

Inhaltsverzeichnis

I Einleitung / Motivation	1
II Historie der Echzeit-Übertragung	1
II-A 1996: RTP & RTCP	1
II-B 2004: SRTP	2
II-C 2005: RTMP	2
II-D 2006: DTLS	2
II-E 2010: WebRTC	2
III Architekturmuster	2
III-A Peer-To-Peer	2
III-A1 Signaling Server	2
III-A2 Holepunching	3
III-A3 IP-Multicast	3
III-B Relay	3
IV Implementierung eines Kamera-Live-Streams	3
IV-A Ziel	3
IV-B Umfang	3
IV-C Architektur	3
IV-C1 Kamera	4
IV-C2 Signaling-Server	4
IV-C3 Interface	4
IV-D Signaling Server	4
IV-D1 Verbindungsauflaufbau	4
IV-D2 Spezifikation des Signaling-Protokolls	4
IV-D3 Raum-Verwaltung	6
IV-D4 Kommunikation zwischen mehreren Client-Verbindungen	6
IV-E Interface	6
IV-E1 WebSocket-Verbindung	7
IV-E2 Erstellen / Beitreten eines Raumes	7

IV-E3 Verbindungsauflbau	7
IV-E4 Verbindungsabbau	7
IV-F Kamera	7
IV-F1 GStreamer	8
IV-F2 Architektur der Kamera-Software	9
IV-F3 Vorbereitung der Laufzeitumgebung	9
IV-F4 Automatischer Start der Kamera-Software	9
V Benchmarks	10
VI Auswertung	10
VII Nächste Schritte	10
Literatur	10
Anhang	12

Abbildungsverzeichnis

1 Signaling zwischen der Kamera und dem Interface.	3
2 Asynchrone Abarbeitung einer IO intensiven Aufgabe	4
3 Anwendungsbeispiel des Singaling-Protokolls	5
4 Broadcast-Channel Konzept	6
5 Kommunikation von Nutzern über einen Broadcast-Channel	7
6 WebRTC Support-Matrix [1]	7
7 WebRTC-Verbindungsauflbau [2]	8
8 GStreamer Architektur [14]	9
9 Zustandsdiagramm der Kamera-Software	9

Tabellenverzeichnis

1 Einleitung / Motivation

der damit einhergehenden Vernetzung der Arbeitswelt in den letzten Jahren [3] werden auch digitale Meeting-Systeme immer relevanter und müssen immer mehr Nutzer in Echtzeit mit einander verbinden um einen reibungslosen Arbeitsalltag zu gewährleisten. Zu Beginn der Corona-Pandemie stieg der Netzwerktraffic für VoIP und Videokonferenzen im März 2020 um 210%-285% [4].

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 broadcastet? Die Entwicklung der letzten Jahre deuten immer mehr darauf hin, dass *Peer-To-Peer* basierte Lösungssansätze aufgrund der besseren Performance und Skalierbarkeit, in der Regel die bessere Wahl darstellen [5]. *Peer-To-Peer* halten unvorhersehbare Anstiege in der Nutzung [4] 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. [6]

Die Problematik der Echtzeit-Kommunikation im Web beschäftigt auch das *W3C* seit 2011 im Zuge der Standardisierung des seit diesem Jahr zum Web-Standard erklärt Protokoll *WebRTC*. [7]

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 Architekturnuster, 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 *Quality of Service* 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 seitdem einen Grundbaustein der datenformatsagafestischen Echtzeit-Übertragung dar. 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. [8]

RTP und *RTCP* siedeln sich im TCP/IP-Stack über *UDP* an. [8]

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.

In der ersten Version aus 1996 gab es einige Probleme bezüglich [Probleme suchen](#). Diese wurden 2003 in dem RFC3550 überarbeitet - somit wurde der RFC1889 durch die neuere Version 3550 obsolet. In der aktuelleren Version wurden einige Änderungen eingearbeitet: Im wesentlichen "RTCP Packet Send and Receive Rules" "Layered Encodings" "Congestion Control" "Security Considerations" "IANA Considerations" [Dummer text](#).

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*. [9, 10]

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. [11] 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 Telekommunikations-Gesellschaft Cisco bereitgestellt. Diese hat den Namen *libsrtplib* und ist öffentlich auf der Entwickler-Plattform GitHub öffentlich einsehbar (siehe 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 [siehe RTMP spec 1.0](#), 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 Anwendung findet, da mittlerweile viele Browser ihren Flash-Player-Support eingestellt haben ([siehe Chrome und Firefox](#))

Außerdem spricht die hohe Latenz von bis zu 30 Sekunden, die durch die Verwendung von *TCP* zustandekommt ([siehe \[restram.io/streaming-protocols\]\(http://restram.io/streaming-protocols\)](#)), gegen den Einsatz von *RTMP* in einem Echtzeit-Übertragungs-Kontext.

Desweiteren gibt es noch Protokollvarianten die *HTTP* bzw. *HTTPS* als zugrundeliegendes Protokoll verwenden. [Siehe XYZ Quelle suchen](#)

2.4 2006: DTLS

Das *Datagram-Transport-Layer-Security* Protokoll bietet starke Sicherheitsgarantien (equivalent zu *TLS*) für Datagramm basierte Protokolle wie z. B. *UDP*. [12]

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. Sie

WebRTC stellt eine direkte Verbindung zwischen zwei Endgeräten her und verwendet einen mix aus SRTP, DTLS, SDP, (ICE Candidates) und einem Signaling-Channel. Es unterstützt beliebte video codecs wie V8 / V9 und ist in der zukunft auf AV1 angelegt. Der standard audio codec ist opus.

Die ersten *Requests for Comments* oder auch *RFC* zu den grundlegenden Protokollen *RTP* und *Protokoll X*, auf denen die heutigen Protokolle weitestgehend aufbauen, wurden bereits in den späten 1990er Jahren veröffentlicht und seit dem immer weiterentwickelt. [quelle anhängen RFC35XX](#).

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 peers bekannt und dient als kommunikationsplattform, damit sich die beiden (sich gegenseitig initial unbekannten) peers gegenseitig vorstellen können.

Signaling server sind interessanterweise keine feste anforderung für peer to peer muster. Wenn alle peers immer statische addressen hätten, könnten sie auch offline ihre ips / ice candidates / sdp offers etc. austauschen und so eine session aufbauen. Wichtig ist leidlglich eine initiale out of band kommunikation.

Der signaling server kann außerdem noch nach verbindungsauftbau dazu verwendet werden um die bestehende verbindung zu optimieren. Peers können neue ICE candidates vorschlagen um die Übertragung an neue gegebenheiten im internet (z.B. ausfall eines routers) anzupassen. (siehe mdn)

3.1.2 Holepunching

Holepunching beschreibt den prozess lokale firewalls und nats zu durchbrechen um eine direkte verbindung zwischen zwei endgeräten herzustellen.

3.1.3 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 vermindierung 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. (Siehe grafik)

Vergleichs grafik einfügen..

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.

3.2 Relay

4 Implementierung eines Kamera-Live-Streams

In diesem Kapitel wird ein experimentelles Kamera-System entwickelt, das die oben erläuterten Protokolle / Technologien verwendet.

4.1 Ziel

Ziel dieses Experiments ist es eine möglichst einfach ein funktionierendes System zu entwickeln und mögliche Probleme, benötigten Zeitaufwand usw. zu ermitteln. Desweiteren ist ein “Performance” Vergleich mit anderen Live-Stream systemen interessant. Es gilt also die frage zu klären, wie viel arbeit benötigt ein grundsätzlich funktionierendes Software-Produkt das auf Echtzeit-Übertragung beruht.

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 ein User Interface. Dieses wird im Browser laufen, muss aber ausreichend Alternativen für andere Plattformen aufweisen (Nativ, Smartphone usw.). Das Interface und die Kamera starten die Aushandlung des WebRTC-Streams über den Signaling Server. D.h. sie versenden SDP (Offer / Answer) und tauschen ihre ICE-Candidates aus.

4.3 Architektur

Die angedachte Architektur wurde möglichst einfach gehalten. Sie besteht lediglich aus 3 Elementen:

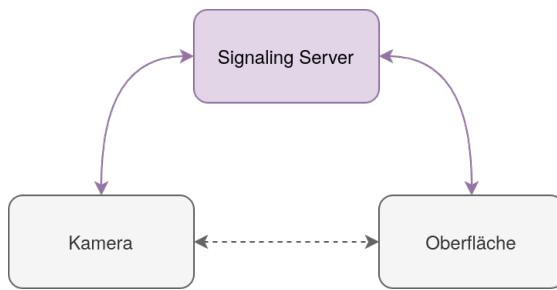


Abbildung 1: Signaling zwischen der Kamera und dem Interface.

Quelle: Schulke 2021, nach [2]

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 Abhängigkeiten) und eine angeschlossene Kamera die von Linux erkannt wird.

Auf der Einplatinencomputer ist das mitgelieferte Betriebssystem “Raspberry PI OS” und die entsprechenden Software-Abhängigkeiten installiert. Siehe [GitHub für Dependencies](#).

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 Clients 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. Clients werden durch eine 128-Bit-Entropie identifiziert.

So können sich zwei Clients 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 Interface

Das Interface 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 User beigetreten ist, fängt der in [Kapitel XY](#) 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 Clients hat. Die Kamera-Software und Interface setzen beide jeweils den laufenden Signaling Server vorraus um korrekt zu funktionieren.

Asynchronität

Eine logische Anforderungen an den Server ist das gleichzeitige verarbeiten mehrerer Verbindungen – ansonsten könnte immer nur ein Client 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.

HIER ZIETIEREN WAS DAS ZEUG HÄLT / ERKLÄRGRAFIKEN NUTZEN

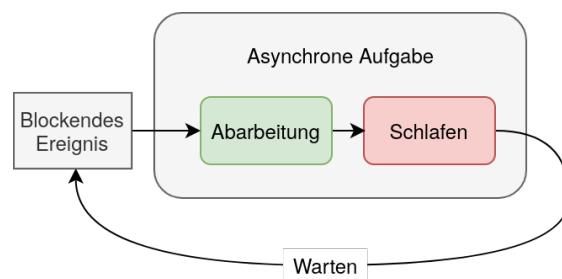


Abbildung 2: Asynchrone Abarbeitung einer IO intensiven Aufgabe

Quelle: Schulke 2021

4.4.1 Verbindungsaufbau

Um eine WebSocket-Secure-Verbindung aufzubauen, muss zuerst eine TCP- und dann darüber eine TLS-Verbindung zu dem Client aufgebaut werden. Über diese wird dann zuerst in HTTP kommuniziert (siehe WebSocket Verbindungsaufbau Quelle suchen) 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.

Grafik einfügen

4.4.2 Spezifikation des Signaling-Protokolls

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

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

Server-Nachrichten

- Hello {Client-ID}

- Joined
- Error
- Room/Join {Client-ID}
- Room/Leave {Client-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 Client-ID. Darauf hin muss der Client ein *JoinOrCreate* senden um einem Raum beizutreten. Bevor der Client die Nachricht *Joined* vom Server erhält, darf dieser keine Signale verschicken.

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

Analog wird eine *Room/Leave* Nachricht verschickt wenn ein Client 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.

4.4.3 Raum-Verwaltung

Die Raum-Verwaltung fällt unter anderem in die Kategorie der Synchronisationsprobleme. Es gibt unter Umständen n Client-Verbindungen, die den Status von einem Raum erfragen wollen, und ggf. einen Anlegen möchten. Dies soll für alle Verbindungen öffentlich geschehen, Clients 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 Client-Verbindung, die aktuelle Raum-Informationen braucht (was hauptsächlich bei dem erstellen / betreten von Räumen der Fall ist), muss nun vorher erst den mutex locken.

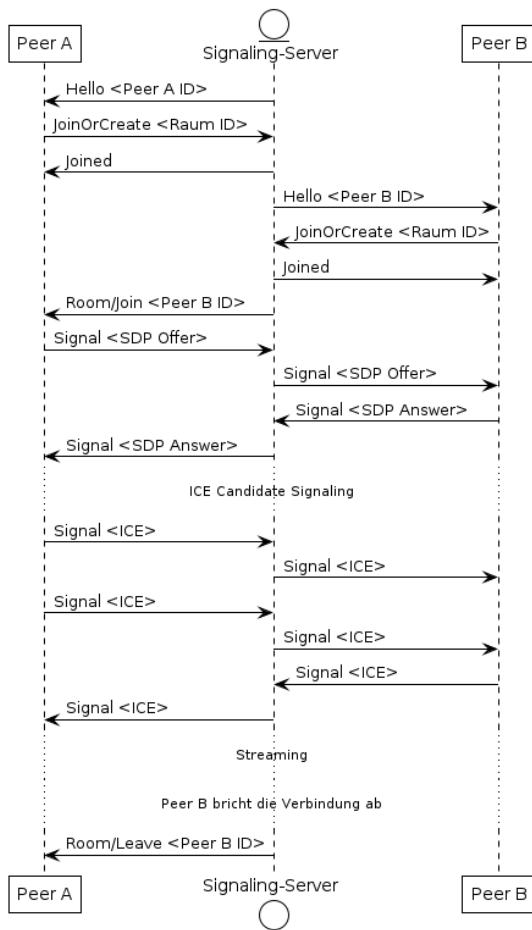


Abbildung 3: Anwendungsbeispiel des Singaling-Protokolls

Quelle: Schulke 2021

4.4.4 Kommunikation zwischen mehreren Client-Verbindungen

Bis zu diesem Punkt ist der Aufbau der Verbindung und das zu implementierende Protokoll klar, allerdings fehlt noch das Verknüpfen von Client-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 Clients Signals 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 funktionieren analog zu dem Broadcast aus der Netzwerktechnik.

Diagramm einfügen..

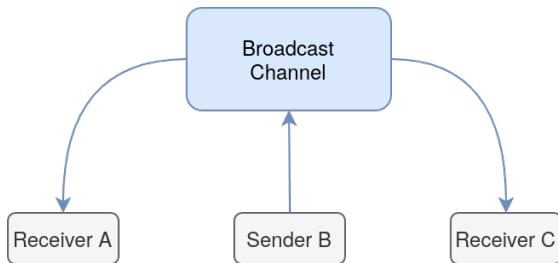


Abbildung 4: Broadcast-Channel Konzept

Quelle: Schulke 2021

Somit kriegt jeder Client bei betreten eines Raumes Zugriff auf einen Sender und einen Empfänger dieses Broadcast-Channels. In diesen schreibt er bei erhält einer Peer-Nachricht vom Typ *Signal* das Signal. Eine Client-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.

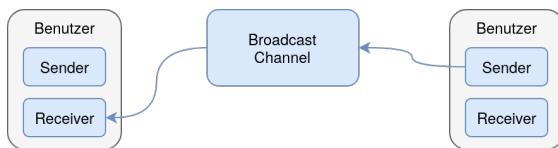


Abbildung 5: Kommunikation von Nutzern über einen Broadcast-Channel

Quelle: Schulke 2021

4.5 Interface

Um den Signaling-Server für ein erstes nutzbares Zwischenergebnis zu verwenden, wurde eine einfache browserbasierte Benutzer-Oberfläche 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 dieses Interface einem beliebigen Raum auf dem Signaling-Server beitreten können. Sobald ein weiterer Benutzer beritt 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 Vorehrungen für die singalisierte Kommunikation mit dem Signaling-Server muss eine WebSocket-Secure Verbindung mit dem Server aufgebaut werden. Dies gelingt ebenfalls über eine Web-API: <https://developer.mozilla.org/de/docs/Web/API/WebSocket>

Nun meldet sich der Server nach dem Verbindungsauftbau 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 / Beitreten 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 Zeitspanne zu verarbeiten, findet die Initialisierung vor der ersten Signalisierungs-Nachricht statt.

4.5.3 Verbindungsauftbau

Moderne Browser wie Firefox, Chrome, Safari etc. stellen eine Implementierung des WebRTC-Standards bereit (siehe Abb. 6), die durch eine übersichtliche API leicht zu integrieren ist.



Abbildung 6: WebRTC Support-Matrix

Quelle: [1]

Um eine vollständige WebRTC-Verbindung aufzubauen ist bei beiden Nutzern eine *RTCPeerConnection* aufzubauen. Auf einer *RTCPeerConnection* ist eine Event-basierte API verfügbar. (Siehe tabelle)

Insbesondere sind für den Verbindungsauftbau 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 singalisierten Server an den Empfänger zu versenden. Der Empfänger verarbeitet diese anschließend und antwortet mit einer SDP-Answer. Nachdem beide Teilnehmer die SDP-Nachrichten für 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.

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

4.5.4 Verbindungsabbau

Nachdem der Signalisierungs-Server eine *Room/Leave*-Nachricht gesendet hat, kann die *RTCPeerConnection* ohne negative Auswirkungen abgebaut werden. Im Interface wird die Anzeige des Video-Streams des anderen Teilnehmers zurückgesetzt und die *RTCPeerConnection* geschlossen.

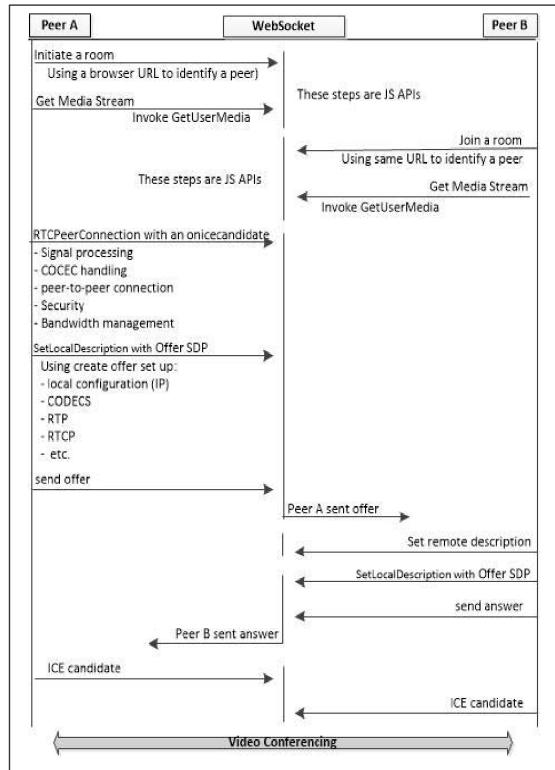


Abbildung 7: WebRTC-Verbindungsauflaufbau

Quelle: [2]

Somit ist das Interface wieder bereit einem weiteren Raum beizutreten und erneut eine WebRTC Verbindung herzustellen.

4.6 Kamera

Zu letzt ist die Kamera zu entwickeln..

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), beherrscht die meistgenutzten Media-Codecs für Audio und Video und kann über Plugins mit weiteren Funktionalitäten erweitert werden.

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.

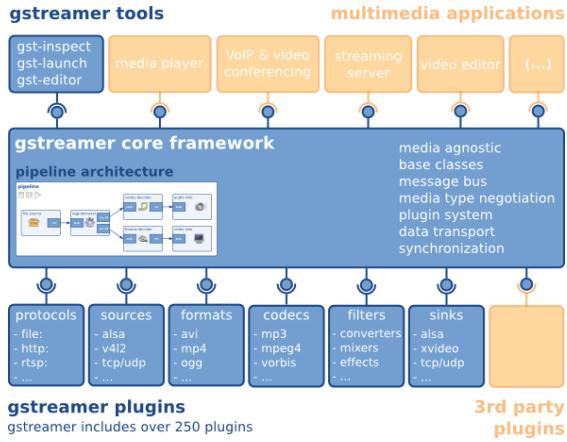


Abbildung 8: GStreamer Architektur

Quelle: [14]

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 Abb. 8 und [14])

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 [14].

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-bad1.0-dev

- libgstreamer-plugins-base1.0-0
- libgstreamer-plugins-base1.0-dev
- libgstreamer1.0-0
- libgstreamer1.0-dev
- libgstrtpserver-1.0-0

4.6.2 Architektur der Kamera-Software

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

1. Verbindungsauftbau zum Signalisierungsserver
2. Streaming via WebRTC

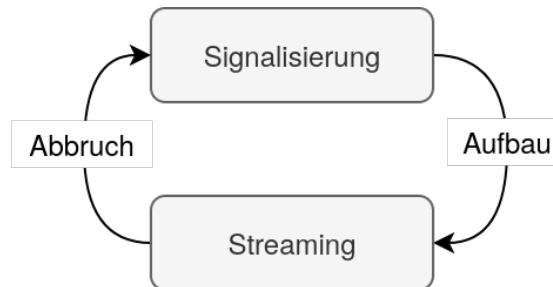


Abbildung 9: Zustandsdiagramm der Kamera-Software

Quelle: Schulke 2021

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 Verbindungsauftbau nach der exakt gleichen Logik ab wie unter IV-E3 erläutert.

4.6.3 Vorbereitung der Laufzeitumgebung

Die Installation des Betriebssystems für die Kamera-Hardware

4.6.4 Automatischer Start der Kamera-Software

5 Benchmarks

- CPU Load - Netzwerkauslastung

6 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 10 Benutzern nicht ansatzweise 1% der CPU-Kapazität. Falls nicht eine große Anzahl an Nutzern gleichzeitig Signale über diesen Server schickt, pendelt sich dieser aufgrund seiner asynchronen Architektur im Ruhemodus bei einer CPU-Last von 0.0% ein, da er 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. Des Weiteren funktioniert auch die Videoübertragung vom Browser über eine WebRTC-Verbindung sehr gut, die Bildübertragungsraten waren i.d.R. 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 ($160\text{px} \times 90\text{px}$) nah an dem angestrebten Wert. Sobald die Qualität diese überschreitet, 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.

7 Nächste Schritte

Die nächsten Schritte um dieses Experiment weiterzuführen wären 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.

Literatur

- [1] "WebRTC Peer-to-peer connections | Can I use... Support tables for HTML5, CSS3, etc," Dezember 2021, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://caniuse.com/rtcpeerconnection>
- [2] N. M. Edan, A. Al-Sherbaz, and S. Turner, "Design and evaluation of browser-to-browser video conferencing in WebRTC," in *2017 Global Information Infrastructure and Networking Symposium (GIIS)*, 2017, pp. 75–78.

- [3] A. G. Blom, K. Möhring, E. Naumann, M. Reifenscheid, A. Weiland, A. Wenz, T. Rettig, R. Lehrer, U. Krieger, S. Juhl, S. Friedel, F. Marina, and C. Cornesse, “Schwerpunktbericht zur Nutzung und Akzeptanz von Homeoffice in Deutschland während des Corona-Lockdowns,” in *Die Mannheimer Corona-Studie*. Universität Mannheim, Juni 2020, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://www.uni-mannheim.de/gip/corona-studie/#c215325>
- [4] “COVID-19 Network Update,” Dezember 2020, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://corporate.comcast.com/covid-19/network/may-20-2020>
- [5] Y. Deguo, N. Lianqiang, and W. Xincun, “A P2P-SIP Architecture for Real Time Stream Media Communication,” in *2009 Ninth International Conference on Hybrid Intelligent Systems*, vol. 3, 2009, pp. 24–28.
- [6] A. Hitzig, “Genug Datenvolumen für alles,” *PC-WELT*, Juni 2021, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://www.pcwelt.de/a/wie-viel-datenvolumen-brauche-ich,3448702>
- [7] “WebRTC 1.0: Real-Time Communication Between Browsers,” Januar 2021, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://www.w3.org/TR/webrtc>
- [8] R. Frederick, S. L. Casner, V. Jacobson, and H. Schulzrinne, “RTP: A Transport Protocol for Real-Time Applications,” RFC 1889, Januar 1996. [Online]. Available: <https://rfc-editor.org/rfc/rfc1889.txt>
- [9] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” RFC 3550, jul 2003. [Online]. Available: <https://rfc-editor.org/rfc/rfc3550.txt>
- [10] E. Schooler, J. Rosenberg, H. Schulzrinne, A. Johnston, G. Camarillo, J. Peterson, R. Sparks, and M. J. Handley, “SIP: Session Initiation Protocol,” RFC 3261, Juli 2002. [Online]. Available: <https://rfc-editor.org/rfc/rfc3261.txt>
- [11] K. Norrman, D. McGrew, M. Naslund, E. Carrara, and M. Baugher, “The Secure Real-time Transport Protocol (SRTP),” RFC 3711, März 2004. [Online]. Available: <https://rfc-editor.org/rfc/rfc3711.txt>
- [12] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security,” RFC 4347, April 2006. [Online]. Available: <https://rfc-editor.org/rfc/rfc4347.txt>
- [13] Autoren der Wikimedia-Projekte, “Real Time Messaging Protocol – Wikipedia,” Oktober 2005, [Abgerufen am 5. Dezember 2021]. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Real_Time_Messaging_Protocol&oldid=210122188
- [14] “What is GStreamer?” Dezember 2021, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://gstreamer.freedesktop.org/documentation/application-development/introduction/gstreamer.html?gi-language=c>
- [15] A. Bychok, “Streaming Protocol Comparison: RTMP, WebRTC, FTL, SRT – Restream Blog,” *Ultimate Live Streaming Hub – Restream Blog*, August 2021, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://restream.io/blog/streaming-protocols>
- [16] H. Tschofenig, E. Rescorla, and J. Fischl, “Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS),” RFC 5763, Mai 2010. [Online]. Available: <https://rfc-editor.org/rfc/rfc5763.txt>

- [17] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security Version 1.2,” RFC 6347, Januar 2012. [Online]. Available: <https://rfc-editor.org/rfc/rfc6347.txt>
- [18] H. T. Alvestrand, “Overview: Real-Time Protocols for Browser-Based Applications,” RFC 8825, Januar 2021. [Online]. Available: <https://rfc-editor.org/rfc/rfc8825.txt>
- [19] C. Perkins, M. Westerlund, and J. Ott, “Media Transport and Use of RTP in WebRTC,” RFC 8834, Januar 2021. [Online]. Available: <https://rfc-editor.org/rfc/rfc8834.txt>
- [20] “Signaling and video calling - Web APIs | MDN,” Dezember 2021, [Abgerufen am 5. Dezember 2021]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signalaling_and_video_calling
- [21] “DTLS (Datagram Transport Layer Security) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN,” Dezember 2021, [Abgerufen am 5. Dezember 2021]. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Glossary/DTLS>

```

signaling/main.rs

use anyhow::bail;
use async_std::net::TcpListener;
use async_std::task;
use async_tungstenite::accept_async;
use simple_logger::SimpleLogger;
use structopt::StructOpt;
use std::str::FromStr;

use signaling::protocol;
use signaling::state;
use signaling::tls;

#[derive(Clone, Debug)]
pub enum Environment {
    Production,
    Development
}

impl FromStr for Environment {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> anyhow::Result<Environment> {
        match s {
            "dev" | "development" => Ok(Environment::Development),
            "prod" | "production" => Ok(Environment::Production),
            _ => bail!("Unknown environment")
        }
    }
}

#[derive(StructOpt)]
struct Options {
    #[structopt(short = "e", long = "env")]
    environment: Environment
}

#[async_std::main]
async fn main() -> anyhow::Result<()> {
    SimpleLogger::new()
        .with_level(log::LevelFilter::Debug)
        .init()
        .unwrap();

    let Options { environment } = Options::from_args();
    let state = state::ServerState::new();
    let listener = TcpListener::bind("0.0.0.0:8192").await?;

    log::info!("running in mode {:?}", environment);
    log::info!("started listening on wss://0.0.0.0:8192");

    let acceptor = match environment {
        Environment::Production => tls::create_acme_acceptor().await,
        Environment::Development => tls::create_self_signed_acceptor().await
    };

    let log_state = state.clone();
    task::spawn(async move {
        let lock = log_state.lock().await;
        let mut old_state = format!("{:?}", lock);
        drop(lock);

        log::info!("state initialized {}", old_state);

        loop {
            task::sleep(std::time::Duration::from_millis(1000)).await;

            let lock = log_state.lock().await;
            let current_state = format!("{:?}", lock);
            drop(lock);

            if current_state != old_state {

```

```

        log::info!("state changed {}", current_state);
        old_state = current_state;
    }
};

loop {
    match listener.accept().await {
        Ok((tcp, _)) => {
            let acceptor = acceptor.clone();
            let state = state.clone();

            task::spawn(async move {
                let tls = match acceptor.accept(tcp).await {
                    Ok(tls) => tls,
                    _ => bail!("tls: failed to accept tls stream")
                };

                let websocket = match accept_async(tls).await {
                    Ok(websocket) => websocket,
                    Err(e) => bail!("failed to accept websocket connection! ({:?})", e)
                };

                match protocol::handler(websocket, state).await {
                    Ok(_) => log::info!("websocket handler exited!"),
                    Err(e) => log::error!("websocket handler crashed! ({:?})", e),
                }

                Ok(())
            });
        }
        Err(err) => log::error!("accept: {:?}", err),
    }
}
}

signaling/lib.rs

pub mod protocol;
pub mod rooms;
pub mod signals;

#[cfg(feature = "server")]
pub mod tls;
#[cfg(feature = "server")]
pub mod state;

signaling/tls.rs

use async_std::fs;
use async_std::sync::Arc;
use async_rustls::rustls::{NoClientAuth, ServerConfig};
use async_std::net::TcpStream;
use async_std::path::Path;
use async_std::task;
use rustls_acme::{acme, ResolvesServerCertUsingAcme, TlsAcceptor};

#[derive(Clone)]
pub enum Acceptor {
    ACME(rustls_acme::TlsAcceptor),
    SelfSigned(async_rustls::TlsAcceptor)
}

impl Acceptor {
    pub async fn accept(&self, tcp: TcpStream) -> Result<async_rustls::server::TlsStream<TcpStream>, std::io::Error> {
        match self {
            Acceptor::SelfSigned(self_signed_acceptor) => self_signed_acceptor.accept(tcp).await,
            Acceptor::ACME(acme_acceptor) => acme_acceptor.accept(tcp).await
        }
    }
}

```

```

pub async fn create_acme_acceptor() -> Acceptor {
    let resolver = ResolvesServerCertUsingAcme::new();
    let config = ServerConfig::new(NoClientAuth::new());
    let acceptor = TlsAccepter::new(config, resolver.clone());
    let domains = vec!["signaling.schulke.xyz".into()];
    let cache_dir = Path::new("/tmp/signaling-cert-cache");

    task::spawn(async move {
        resolver
            .run(
                if cfg!(debug_assertions) {
                    acme::LETS_ENCRYPT_STAGING_DIRECTORY
                } else {
                    acme::LETS_ENCRYPT_PRODUCTION_DIRECTORY
                },
                domains,
                Some(cache_dir),
            )
            .await;
    });

    Acceptor::ACME(acceptor)
}

pub async fn create_self_signed_acceptor() -> Acceptor {
    const BASE_PATH: &str = "./keys";
    const KEY_FILE: &str = "localhost.key";
    const CERT_FILE: &str = "localhost.crt";

    let mut tls_config = ServerConfig::new(async_rustls::rustls::NoClientAuth::new());

    let key = {
        let key = fs::read(Path::new(BASE_PATH).join(KEY_FILE)).await.unwrap();
        rustls::internal::pemfile::pkcs8_private_keys(&mut key.as_slice()).unwrap().remove(0)
    };

    let cert = {
        let cert = fs::read(Path::new(BASE_PATH).join(CERT_FILE)).await.unwrap();
        rustls::internal::pemfile::certs(&mut cert.as_slice()).unwrap()
    };

    tls_config
        .set_single_cert(cert, key)
        .unwrap();

    Acceptor::SelfSigned(
        async_rustls::TlsAccepter::from(Arc::new(tls_config))
    )
}

signaling/rooms.rs

use uuid::Uuid;
use anyhow::ensure;
use std::convert::TryFrom;
use std::fmt;
use serde::{Serialize, Deserialize};

#[cfg(feature = "server")]
use std::collections::{HashMap, HashSet};
#[cfg(feature = "server")]
use async_broadcast::{Receiver, Sender, broadcast};
#[cfg(feature = "server")]
use async_std::sync::{Mutex, Arc};
#[cfg(feature = "server")]
use async_std::task::block_on;

#[derive(Clone, Copy, Hash, PartialEq, Eq, PartialOrd, Ord, Serialize, Deserialize)]
#[serde(into = "String", try_from = "String")]
pub struct RoomId(Uuid, Uuid);

impl RoomId {

```

```

pub fn new(l: Uuid, r: Uuid) -> Self {
    Self(l, r)
}

impl TryFrom<String> for RoomId {
    type Error = anyhow::Error;

    fn try_from(s: String) -> anyhow::Result<Self> {
        ensure!(s.len() == 64, "Should be a 64 char hex string");
        Ok(Self(Uuid::parse_str(&s[0..32])?, Uuid::parse_str(&s[32..64])?))
    }
}

impl From<RoomId> for String {
    fn from(id: RoomId) -> String {
        format!("{}{}", &id.0.to_simple(), &id.1.to_simple())
    }
}

impl fmt::Display for RoomId {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}", String::from(*self))
    }
}

impl fmt::Debug for RoomId {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        f.debug_tuple("RoomId").field(&String::from(*self)).finish()
    }
}

#[cfg(feature = "server")]
pub type RoomMap = HashMap<RoomId, Room>;

#[derive(Clone, Debug, Serialize, Deserialize)]
#[serde(tag = "type", content = "data", rename_all = "SCREAMING_SNAKE_CASE")]
pub enum Message {
    Join {
        peer: Uuid
    },
    Leave {
        peer: Uuid
    },
    Signal {
        peer: Uuid,
        signal: crate::signals::IceOrSdp
    }
}

#[cfg(feature = "server")]
#[derive(Clone)]
pub struct Room {
    id: RoomId,
    creator: Uuid,
    inner: Arc<Mutex<Inner>>
}

#[cfg(feature = "server")]
impl fmt::Debug for Room {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        f.debug_struct("Room")
            .field("id", &self.id)
            .field("creator", &self.creator)
            .finish_non_exhaustive()
    }
}

#[cfg(feature = "server")]
#[derive(Clone, Debug)]
struct Inner {
    peers: HashSet<Uuid>,
    channel: (Sender<Message>, Receiver<Message>)
}

```

```

}

#[cfg(feature = "server")]
impl Room {
    pub fn new(creator: Uuid, id: RoomId) -> Room {
        Room {
            id,
            creator,
            inner: Arc::new(Mutex::new(Inner {
                peers: HashSet::new(),
                channel: broadcast(4096)
            })),
        }
    }

    pub async fn join(&self, peer: Uuid) -> RoomHandle {
        RoomHandle::new(peer, self.clone()).await
    }
}

#[cfg(feature = "server")]
pub struct RoomHandle(Uuid, Room, Sender<Message>, Receiver<Message>);

#[cfg(feature = "server")]
impl RoomHandle {
    pub async fn new(peer: Uuid, room: Room) -> Self {
        let mut inner = room.inner.lock().await;
        inner.peers.insert(peer);
        let (s, r) = inner.channel.clone();
        s.broadcast(Message::Join { peer }).await.ok();
        drop(inner);
        Self(peer, room, s, r)
    }

    pub async fn send(&mut self, msg: Message) -> anyhow::Result<()> {
        self.2.broadcast(msg).await?;
        Ok(())
    }

    pub async fn recv(&mut self) -> anyhow::Result<Message> {
        self.3.recv().await.map_err(|e| e.into())
    }
}

#[cfg(feature = "server")]
impl Drop for RoomHandle {
    fn drop(&mut self) {
        let mut inner = block_on(self.1.inner.lock());
        inner.peers.remove(&self.0);
        block_on(self.2.broadcast(Message::Leave { peer: self.0 })).ok();
    }
}

signaling/state.rs

use async_std::sync::{Arc, Mutex, MutexGuard};
use uuid::Uuid;
use std::collections::HashSet;
use crate::rooms::RoomMap;

#[derive(Clone, Debug)]
pub struct ServerState(Arc<Mutex<(RoomMap, HashSet<Uuid>)>>);

impl<'a> ServerState {
    pub fn new() -> Self {
        Self(Arc::new(Mutex::new((RoomMap::new(), HashSet::new()))))
    }

    pub async fn lock(&'a self) -> MutexGuard<'a, (RoomMap, HashSet<Uuid>)> {
        self.0.lock().await
    }
}

```

```

signaling/signals.rs

use serde::{Serialize, Deserialize};

#[derive(Clone, Debug, Serialize, Deserialize)]
#[serde(untagged)]
pub enum IceOrSdp {
    Ice(Ice),
    Sdp(Sdp),
}

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct Ice {
    pub candidate: String,
    #[serde(rename = "sdpMLineIndex")]
    pub line_index: u32,
}

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct Sdp {
    pub r#type: SdpType,
    pub sdp: String,
}

#[derive(Clone, Debug, Serialize, Deserialize, Eq, PartialEq)]
#[serde(rename_all = "camelCase")]
pub enum SdpType {
    Offer,
    Answer,
}

camera/main.rs

mod webrtc;
mod signaling;

use url::Url;
use simple_logger::SimpleLogger;
use async_std::task;

#[derive(Clone, Copy, Debug)]
pub enum Environment {
    Development
}

const ENVIRONMENT: Environment = Environment::Development;

struct Options {
    environment: Environment,
    signaling_server: Url,
    room_id: signaling_types::rooms::RoomId
}

#[async_std::main]
async fn main() -> anyhow::Result<()> {
    gstreamer::init().unwrap();

    SimpleLogger::new()
        .with_level(log::LevelFilter::Debug)
        .init()
        .unwrap();

    async fn stream() -> anyhow::Result<()> {
        let websocket = signaling::entry(ENVIRONMENT).await?;
        webrtc::entry(websocket).await
    }

    loop {
        match stream().await {
            Ok(_) => log::info!("Stream ended expectedly, restarting"),
            Err(e) => log::error!("Stream ended unexpectedly ({}), trying again..", e),
        }
        task::sleep(std::time::Duration::from_millis(500)).await;
    }
}

```

```

        }
    }

camera/webrtc.rs

use crate::signaling;
use anyhow::{anyhow, bail, Context};
use async_std::task;
use futures::FutureExt;
use gstreamer::prelude::*;
use gstreamer::ElementFlags;
use signaling_types::protocol::{PeerMessage, ServerMessage};
use signaling_types::rooms::Message as RoomMessage;

#[allow(unused)]
const WEBRTC_PIPELINE: &str = "v4l2src device=/dev/video2 name=src ! vp8enc ! rtpvp8pay pt=96 \
    ! webrtcbin. webrtcbin name=webrtcbin stun-server=stun://stun.l.google.com:19302";

#[allow(unused)]
const WEBRTC_PIPELINE_TEST: &str =
    "videotestsrc pattern=ball is-live=true ! vp8enc ! rtpvp8pay pt=96 \
    ! webrtcbin. webrtcbin name=webrtcbin stun-server=stun://stun.l.google.com:19302";

#[allow(unused)]
const WEBRTC_PIPELINE_DESKTOP: &str =
    "ximagesrc ! videorate ! videoscale ! video/x-raw,width=1920,height=1080,framerate=5/1 \
    ! videoconvert ! vp8enc ! rtpvp8pay pt=96 ! webrtcbin. webrtcbin name=webrtcbin";

#[allow(unused)]
const WEBRTC_PIPELINE_FANCY: &str = "v4l2src device=/dev/video0 ! videorate ! videoscale \
    ! video/x-raw,width=640,height=360,framerate=5/1 ! videoconvert ! queue max-size-buffers=1 \
    ! x264enc bitrate=600 speed-preset=ultrafast tune=zerolatency key-int-max=15 \
    ! video/x-h264,profile=constrained-baseline ! queue max-size-time=100000000 ! h264parse \
    ! rtph264pay config-interval=-1 name=payloader aggregate-mode=zero-latency \
    ! application/x-rtp,media=video,encoding-name=H264,payload=96 ! webrtcbin. \
    webrtcbin name=webrtcbin stun-server=stun://stun.l.google.com:19302 ";

#[allow(unused)]
const WEBRTC_PIPELINE_LIBCAMERA: &str = "libcamerasrc ! video/x-raw,width=1600,height=1200 \
    ! videoconvert ! videoflip motion=vertical-flip ! vp8enc ! rtpvp8pay pt=96 \
    ! webrtcbin. webrtcbin name=webrtcbin stun-server=stun://stun.l.google.com:19302";

pub async fn entry(mut websocket: crate::signaling::WebSocket) -> anyhow::Result<()> {
    let pipeline = gstreamer::parse_launch(WEBRTC_PIPELINE_FANCY)
        .unwrap()
        .downcast::<gstreamer::Pipeline>()
        .map_err(|_| anyhow!("Failed to parse gstreamer pipeline"))?;

    let webrtcbin = pipeline
        .by_name("webrtcbin")
        .context("Failed to get element by name")?
        .dynamic_cast::<gstreamer::Element>()
        .map_err(|_| anyhow!("Failed to downcast the pipeline to the webrtc bin"))?;

    webrtcbin.set_element_flags(ElementFlags::SINK);
    pipeline.set_state(gstreamer::State::Ready)?;
    let handler = crate::florian::attach(webrtcbin)?;

    loop {
        futures::select! {
            local = handler.channel().recv().fuse() => {
                signaling::write_to_ws(&mut websocket, &PeerMessage::Signal(local.expect("signal not there"))).await?;
            }
            remote = signaling::read_from_ws(&mut websocket).fuse() => {
                match remote {
                    Ok(ServerMessage::Room(room_msg)) => match room_msg {
                        RoomMessage::Signal { signal, .. } => { handler.process(signal); },
                        RoomMessage::Join { .. } => {
                            pipeline.set_state(gstreamer::State::Playing)?;
                            log::info!("Set pipeline to play, starting negotiation");
                        },
                        RoomMessage::Leave { .. } => {
                    }
                }
            }
        }
    }
}

```

```

        pipeline.set_state(gstreamer::State::Null)?;
        log::info!("Set pipeline to pause, stopped stream, restarting..");
        return Ok(())
    }
}
Ok(msg) => bail!("Unexpected message {:#?}", msg),
Err(e) => bail!("Websocket error {:#?}", e)
}
}
_= task::sleep(std::time::Duration::from_millis(2000)).fuse() => {
    handler.bin().emit_by_name("get-stats", &[
        &None::<gstreamer::Pad>,
        &gstreamer::Promise::with_change_func(move |reply| {
            log::debug!("status:");
            let reply = reply.expect("we needed stats :").expect("same");
            reply.iter().for_each(|(k,v)| log::debug!("{}: v {:?}", k, v));
        })
    ]).ok();
}
}
}
}

camera/signaling.rs

use uuid::Uuid;
use anyhow::{anyhow, bail, Context};
use async_tungstenite::WebSocketStream;
use futures::{SinkExt, StreamExt};
use url::Url;
use crate::Environment;
use signaling_types::rooms::RoomId;
use signaling_types::protocol::{PeerMessage, ServerMessage};

pub type WebSocket = WebSocketStream<async_rustls::TlsStream<async_std::net::TcpStream>>;

const SIGNALING_SERVER_ADDRESS: &str = "wss://signaling.schulke.xyz:8192";
const SIGNALING_SERVER_ADDRESS_LOCAL: &str = "wss://localhost:8192";

pub async fn read_from_ws(websocket: &mut WebSocket) -> anyhow::Result<ServerMessage> {
    use async_tungstenite::tungstenite::Message;

    let message = loop {
        match websocket.next().await {
            None => bail!("websocket stream ended unexpectedly"),
            Some(Ok(Message::Binary(json))) => break serde_json::from_slice(json.as_slice())?,
            Some(Ok(Message::Text(json))) => break serde_json::from_str(json.as_str())?,
            Some(Ok(Message::Ping(_))) | Some(Ok(Message::Pong(_))) => continue,
            Some(Ok(Message::Close(_))) => bail!("websocket: received close"),
            Some(Err(err)) => bail!("websocket stream error: {:?}", err),
        };
    };
    Ok(message)
}

pub async fn write_to_ws(websocket: &mut WebSocket, msg: &PeerMessage) -> anyhow::Result<()> {
    use async_tungstenite::tungstenite::Message;
    let message = Message::Binary(serde_json::to_vec(msg)?);
    websocket.send(message).await.with_context(|| anyhow!("Failed to send message {:?}", msg))
}

pub async fn entry(environment: Environment) -> anyhow::Result<WebSocket> {
    log::info!("Starting signaling!");

    let mut websocket = {
        let url = Url::parse(SIGNALING_SERVER_ADDRESS_LOCAL)?;
        let tls_stream = tls::get_dangerous_tls_stream(&url).await?;
        let (connection, _) = async_tungstenite::client_async(url.to_string(), tls_stream).await?;
        connection
    };

    let uuid = match read_from_ws(&mut websocket).await? {

```

```

        ServerMessage::Hello(uuid) => uuid,
        _ => bail!("Unexpected message from server")
    };

    log::info!("We have this uuid: {}", uuid.to_simple());

    let room_id = RoomId::new(Uuid::from_slice(&[255; 16])?, Uuid::from_slice(&[255; 16])?);
    write_to_ws(&mut websocket, &PeerMessage::JoinOrCreate(room_id)).await?;

    match read_from_ws(&mut websocket).await? {
        ServerMessage::Joined => log::info!("Joined room {}", room_id),
        msg => bail!("Unexpected message from server: {:?}", msg)
    }

    Ok(websocket)
}

mod tls {
    use async_rustls::{rustls::ClientConfig, webpki::DNSNameRef, TlsConnector};
    use async_std::net::TcpStream;
    use async_std::sync::Arc;
    use url::Url;

    pub type TlsStream = async_rustls::TlsStream<TcpStream>;

    pub async fn get_dangerous_tls_stream(uri: &Url) -> anyhow::Result<TlsStream> {
        mod dangerous {
            use async_rustls::rustls::{Certificate, RootCertStore, TLSError};
            use async_rustls::rustls::{ServerCertVerified, ServerCertVerifier};

            pub struct NoCertificateVerification {}

            impl ServerCertVerifier for NoCertificateVerification {
                fn verify_server_cert(
                    &self,
                    _: &RootCertStore,
                    _: &[Certificate],
                    _: async_rustls::webpki::DNSNameRef<'_>,
                    _: &[u8],
                ) -> Result<ServerCertVerified, TLSError> {
                    Ok(ServerCertVerified::assertion())
                }
            }
        }
        let addrs = uri.socket_addrs(|| None).unwrap();
        let tcp = TcpStream::connect(&*addrs).await?;

        let mut config = ClientConfig::new();
        config
            .dangerous()
            .set_certificate_verifier(Arc::new(dangerous::NoCertificateVerification {}));

        let connector = TlsConnector::from(Arc::new(config));
        let domain = DNSNameRef::try_from_ascii_str("x")?;
        let tls = connector.connect(domain, tcp).await?;

        Ok(tls.into())
    }
}

interface/index.html

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <style>
            .room {
                display: flex;
                flex-direction: column;
                width: 1280px;
                height: 720px;

```

```

        }

    .room video {
        /*height: 50%;*/
        background-color: gray;
    }

    .room video:first-child {
        border-bottom: 1px solid lightgray;
    }

```

</style>

</head>

<body>

<div id="app"></div>

<div class="room">

!--<video id="preview" autoplay playsinline>Your browser does not support video</video>-->

<video id="stream" autoplay playsinline>Your browser does not support video</video>

</div>

<pre id="log"></pre>

<script>

```

const websocket = new WebSocket('wss://localhost:8192');
const log = document.getElementById("log");
const app = document.getElementById("app");

const send = msg => {
    const serialized = JSON.stringify(msg);
    websocket.send(new Blob([serialized], {type : 'application/json'}));
    log.append(`Sent ${serialized}\n`);
}

const message = {
    joinOrCreate: room => ({type: "JOIN_OR_CREATE", data: room}),
    signal: payload => ({type: "SIGNAL", data: payload}),
};

const state = {
    error: null,
    uuid: null,

    stream: {
        preview: null,
        video: null,
        localMedia: null,
        peerConnection: null,
    },
    room: null,
    roomLog: [],
    roomCandidate: null
};

websocket.onmessage = msg =>
    msg.data.text()
        .then(txt => {
            log.append(`Received ${txt}\n`);
            onServerMessage(JSON.parse(txt));
        });

```

```

websocket.onclose = () => {
    state.error = 'Lost the connection';
    render(state);
    logStateUpdate();
};

let onServerMessage = msg => {
    switch (msg.type) {
        case 'HELLO':
            state.uuid = msg.data;
            logStateUpdate();
            break;
        case 'ERROR':
            state.error = msg.data;
            logStateUpdate();
    }
};

```

```

        break;
    case 'JOINED':
        state.room = state.roomCandidate;
        state.roomCandidate = null;
        startCall(state);
        logStateUpdate();
        break;
    case 'ROOM':
        switch (msg.data.type) {
            case 'JOIN':
                state.roomLog.push(msg.data)
                console.log("start negotiation with", msg.data.data.peer)
                if (!state.stream.peerConnection) {
                    startCall(state);
                }
                logStateUpdate();
                break;
            case 'LEAVE':
                state.roomLog.push(msg.data)
                if (state.stream.peerConnection) {
                    stopCall(state);
                    console.log("stop negotiation with", msg.data.data.peer)
                }
                logStateUpdate();
                break;
            case 'SIGNAL':
                if (state.stream.peerConnection) {
                    onSignal(msg.data);
                } else {
                    console.log("ignored signal since no peer connection is up")
                }
                break;
            }
            break;
        default:
            console.log("Unknown", msg);
        }
    render(state);
};

const logStateUpdate = () => {
    log.append(`State ${JSON.stringify(state)}\n`);
    if (state.error) {
        log.append(`Error ${state.error}\n`);
    }
}

let render = state => {
    app.innerHTML = '';
    if (state.error) {
        app.innerHTML += `
            <strong>Error: ${state.error}</strong><br>
            <button onclick="window.location.reload()">Retry</button>
        `;
        return;
    }

    if (state.uuid) {
        app.innerHTML += `<strong>Hello ${state.uuid}</strong><br><br>`;
    }

    if (state.room == null) {
        app.innerHTML += '<button id="join-or-create-room">Join Or Create Room</button>';

        let btn = document.getElementById("join-or-create-room");
        // leaking listeners since we dont clean up
        btn.addEventListener("click", () => {
            let params = new URLSearchParams(window.location.search);
            let roomId = params.get("room");

            while (!roomId || roomId?.length !== 64 || parseInt(roomId, 16) === NaN) {

```

```

        roomId = prompt("Enter a valid room id (256 bit hex encoded)").toString();
    }

    state.roomCandidate = roomId;
    logStateUpdate();
    send(message.joinOrCreate(roomId));
);
}

if (state.uuid && state.room != null) {
    let log = state.roomLog.map(entry => {
        if (entry.type === "LEAVE") {
            return `Peer ${entry.data.peer} left the room`;
        } else if (entry.type === "JOIN") {
            return `Peer ${entry.data.peer} joined the room`;
        } else {
            return '';
        }
    }).join('\n');

    app.innerHTML =
        `You (${state.uuid}) are in room ${state.room}</strong><br><br>
        <pre>${log}</pre>
        `;

    state.stream.preview = document.getElementById("preview");
    state.stream.video = document.getElementById("stream");
}
};

const startCall = state => {
    console.log('Creating RTCPeerConnection');

    /*
    state.stream.localMedia = getLocalMedia(state).then(stream => {
        console.log('Adding local stream to call', stream, state.stream.peerConnection);
        //stream.getTracks().forEach(track => state.stream.peerConnection.addTrack(track, stream));
        return stream;
}).catch(e => state.error = e);
*/
}

state.stream.peerConnection = new RTCPeerConnection({
    iceServers: [
        {urls: "stun:stun.l.google.com:19302"}
    ],
});

state.stream.peerConnection.ondatachannel = console.warn;
state.stream.peerConnection.oniceconnectionstatechange = ev => console.warn("ice connection state change", ev.currentTarget.iceConnectionState);
state.stream.peerConnection.onicegatheringstatechange = ev => console.warn("ice gathering state change", ev.currentTarget.iceGatheringState);
state.stream.peerConnection.onnegotiationneeded = () => sendOffer(state);
state.stream.peerConnection.onsignalingstatechange = ev => console.warn("signaling state change", ev.currentTarget.signalingState);

state.stream.peerConnection.ontrack = ev => {
    console.log("incoming stream", ev);
    console.log(ev.streams[0].getTracks());

    if (state.stream.video.srcObject !== ev.streams[0]) {
        console.log("storing incoming stream", ev);
        state.stream.video.srcObject = ev.streams[0]
        console.log(state.stream.video)
    }
};

state.stream.peerConnection.onicecandidate = event => {
    if (event.candidate === null || event.candidate === "") {
        console.log("ICE Candidate was null, done");
        return;
    }
}

```

```

        send(message.signal(event.candidate))
    );
};

const stopCall = state => {
    console.log(state.stream.video)
    state.stream.video.srcObject?.getVideoTracks().forEach(track => {
        track.stop();
        state.stream.video.srcObject.removeTrack(track);
    });
    state.stream.video.pause();
    state.stream.video.src = "";
    state.stream.video.load();

    state.stream.peerConnection.close();
    state.stream = {
        ...state.stream,
        localMedia: null,
        peerConnection: null,
    };
};

const sendOffer = state => {
    state.stream.peerConnection.createOffer({ offerToReceiveVideo: true })
        .then(desc => { console.log("got local description: " + JSON.stringify(desc)); return desc})
        .then(desc => state.stream.peerConnection.setLocalDescription(desc))
        .then(() => send(message.signal(state.stream.peerConnection.localDescription)))
};

const onSignal = msg => {
    const signal = msg.data.signal;

    if (signal.candidate && signal.sdpMLineIndex != null) {
        const candidate = new RTCIceCandidate(signal);
        state.stream.peerConnection.addIceCandidate(candidate).catch(e => state.error = e);
        return;
    }

    if (signal.type && signal.sdp) {
        console.log("received sdp signal", signal)
        state.stream.peerConnection.setRemoteDescription(signal)
            .then(() => {
                if (signal.type === "answer") return;

                state.stream.peerConnection.createAnswer()
                    .then(answer => state.stream.peerConnection.setLocalDescription(answer))
                    .then(() => send(message.signal(state.stream.peerConnection.localDescription)))
                    .catch(e => state.error = e);
            });
    }
    return;
}

console.log("Received unknown signal", signal);
}

/*
const getLocalMedia = state => {
    return navigator.mediaDevices.getUserMedia({
        audio: false,
        video: { width: { ideal: 1920 }, height: { ideal: 1080 }, advanced: [{ aspectRatio: 16/9 }] }
    }).then(stream => {
        console.log('Previewing local stream');
        state.stream.preview.srcObject = stream;
        return stream;
    });
};
*/
</script>
</body>
</html>

```



Abbildung 10: Rapsberry PI 4

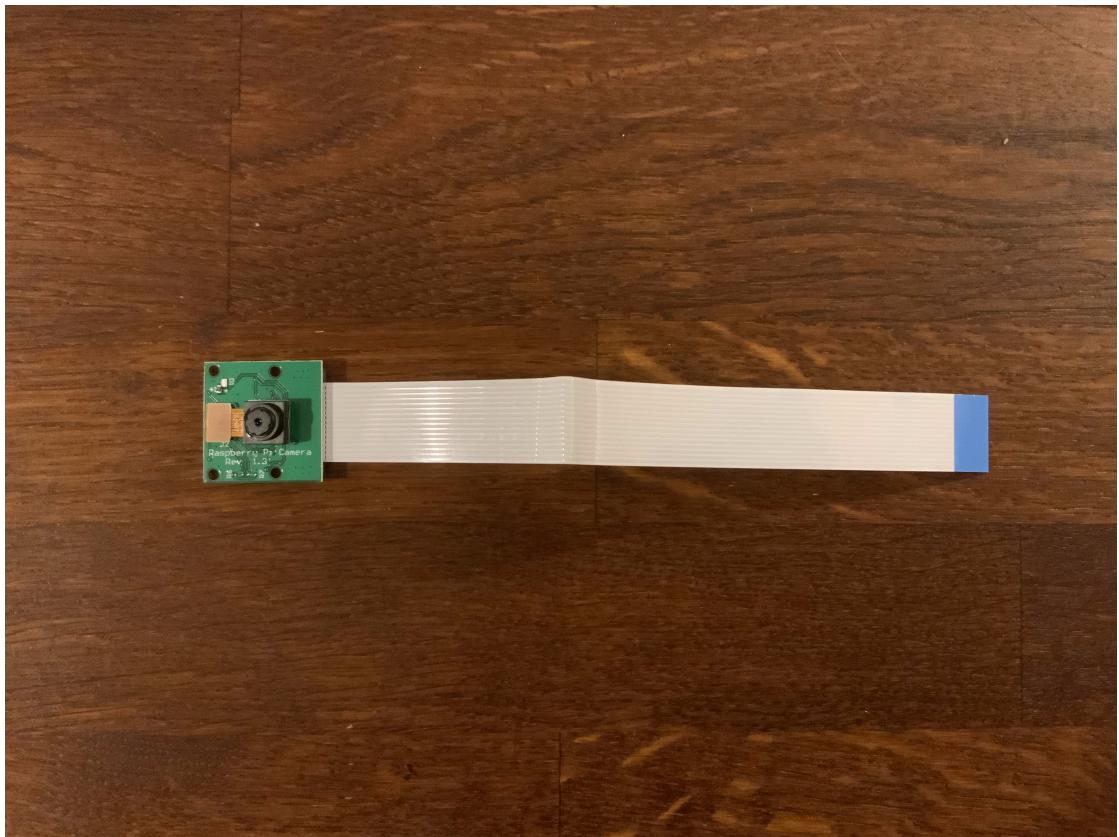


Abbildung 11: Camera Modul



Abbildung 12: Installierte Kamera