Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka techniczna**
Specjalność: **Grafika i Systemy Multimedialne**

PRACA DYPLOMOWA INŻYNIERSKA

Wideokomunikator internetowy implementujący nowoczesne kodeki wideo w języku Rust

Internet videochat application implementing modern video codecs using Rust programming language

Marcel Guzik

Opiekun pracy
dr inż. Marek Woda (K30W04ND03)

WROCŁAW, 2022

Streszczenie

Pandemia COVID-19 oraz zdobywające coraz większą popularność zdalne formy zatrudnienia obrazują jak ważna jest rola połączeń wideo we współczesnym społeczeństwie. W wielu przypadkach kontakt "twarzą w twarz"jest preferowalny, a nawet niezbędny do realizacji pewnych zadań. W takich wypadkach niezawodność i jakość transmisji wideo stają się bardzo ważnymi problemami.

W ciągu ostatnich 20 lat poczynione zostały ogromne postępy w rozwoju infrastruktury internetowej w Polsce. Liczba internautów wzrosła z 16 mln w roku 2011 do 29,7 mln w roku 2021. Znacząco wzrosły szerokości pasm, pozwalające

Lorem ipsum dolor sit amet eleifend et, congue arcu. Morbi tellus sit amet, massa. Vivamus est id risus. Sed sit amet, libero. Aenean ac ipsum. Mauris vel lectus.

Nam id nulla a adipiscing tortor, dictum ut, lobortis urna. Donec non dui. Cras tempus orci ipsum, molestie quis, lacinia varius nunc, rhoncus purus, consectetuer congue risus.

Słowa kluczowe: raz, dwa, trzy, cztery

Abstract

Streszczenie in Polish should fit on the half of the page (the other half should be covered by the abstract in English).

Lorem ipsum dolor sit amet eleifend et, congue arcu. Morbi tellus sit amet, massa. Vivamus est id risus. Sed sit amet, libero. Aenean ac ipsum. Mauris vel lectus.

Nam id nulla a adipiscing tortor, dictum ut, lobortis urna. Donec non dui. Cras tempus orci ipsum, molestie quis, lacinia varius nunc, rhoncus purus, consectetuer congue risus.

Keywords: one, two, three, four

Spis treści

Spis rysunków

Spis tabel

Spis listingów

Skróty

AOM (ang. *Alliance for Open Media*)
AVC (ang. *Advanced Video Coding*)
HEVC (ang. *High Efficiency Video Coding*)
ICE (ang. *Interactive Connectivity Establishment*)
MPEG (ang. *Moving Picture Experts Group*)
STUN (ang. *Session Traversal Utilities for NAT*)
TURN (ang. *Traversal Using Relays around NAT*)
WebRTC (ang. *Web Real Time Communications*)

Rozdział 1

Wstęp

1.1. Wprowadzenie

Pandemia COVID-19 oraz zdobywające coraz większą popularność zdalne formy zatrudnienia obrazują jak ważna jest rola internetowych połączeń wideo we współczesnym społeczeństwie. W wielu przypadkach kontakt "twarzą w twarz"jest preferowalny, a nawet niezbędny do realizacji pewnych zadań. W takich wypadkach niezawodność i jakość transmisji wideo stają się bardzo ważnymi problemami.

W ciągu ostatnich 20 lat poczynione zostały ogromne postępy w rozwoju infrastruktury internetowej w Polsce. Liczba internautów wzrosła z 16 mln w roku 2011 do 29,7 mln w roku 2021. Dzięki rozpowszechnieniu i coraz szerszemu użyciu technologii światłowodowej znacząco wzrosły szerokości pasm, dzięki czemu możliwe stało się transmitowanie jeszcze większej ilości danych w tym samym czasie, a także znaczącemu obniżeniu uległy opóźnienia, dzięki czemu mogły powstać i rozpowszechnić się aplikacje wykorzustujące internet do komunikacji w czasie rzeczywistym, takie jak gry wideo oraz komunikatory internetowe. Rozwój internetu mobilnego umożliwił dostęp do szerokopasmowego internetu na terenach mniej zamożnych i rzadziej zaludnionych.

W porównaniu do tak ogromnego rozwoju internetu, postępy w jakości transmisji wideo były jednak skromne. Rozwój infrastruktury internetowej zapewnił większą niezawodność i wyższe szerokości pasma dzięki czemu transmisje wideo mogły zawierać więcej danych, usprawniając jakość, jednak bardzo szybko trafiliśmy na sufit, który ograniczył postępy w poprawie jakości transmisji wideo:

- Szerokości pasma podawane przez dostawców internetowych są wartościami optymistycznymi, maksymalne wykorzystanie pasma jest możliwe jeżeli dane wysyłane przez łącze jest odpowiednio wysoko buforowane, czyli jeżeli istnieje duża kolejka danych która może zostać wysłana przez łącze naraz. Ta potrzeba kolejkowania sprawia że efektywna szerokość łącza jest mniejsza dla aplikacji czasu rzeczywistego niż innych aplikacji. Oczywiście aplikacje czasu rzeczywistego też wykorzystują buforowanie, ale ponieważ opóźnienie jest w ich wypadku kluczowe i wysłane dane muszą trafić do odbiorcy w ciągu 500ms od ich wysłania przez odbiorcę, buforowanie jakie mogą one robić jest ograniczone.
- O ile szerokość pasma mogąca być wykorzystana do transmisji wideo nie jest już czynnikiem limitującym dla osób prywatnych mających dostęp do połączeń światłowodowych często zapewniających prędkości ponad 100Mb/s, to nadal są one ograniczeniem dla internetowych dostawców wideo, jak np. Youtube, Twitch, lub Netflix. Youtube po otrzymaniu filmu wysłanego przez użytkownika udostępnia jego jeszcze bardziej skompresowaną wersję, Twitch ogranicza bitrate streamów 1080p do 4.5Mb/s, a w trakcie pandemii łączny udział Netflixa w internecie był tak duży, że ten musiał ograniczać ja-

kość strumieni wideo (https://www.forbes.com/sites/johnarcher/2020/05/12/netflix-starts-to-lift-its-coronavirus-streaming-restrictions/)
- Internet mobilny poprawił dostęp do internetu na terenach mniej zamożnych i mniej gęsto zaludnionych, jednak nie jest on w stanie zastąpić światłowodu. Internet mobilny jest wolniejszy od przewodowego, charakteryzuje się też większymi opóźnieniami i większą podatnością na zakłócenia. W takich warunkach nie jest możliwe poprawienie jakości obrazu przez zwiększanie obiętości strumienia wideo, i trzeba polegać na lepszych technikach kompresji.

Mając na uwadze powyższe, pojawiają się pytania: "Czy możliwe jest jeszcze bardziej poprawić jakość transmisji wideo w internecie? W jaki sposób to zrobić jeżeli zwiększanie ilości danych jest problematyczne i podlega malejącym zwrotom?".

Odpowiedź można znaleźć w lepszych metodach kompresji wideo. Aktualny powszechnie używany standard, opracowany przez MPEG standard AVC (Advanced Video Coding) jest używany od roku 2003, a jego następca, HEVC nie uzyskał tak szerokiej adopcji, głównie za sprawą zbyt restrykcyjnych zapisów patentowych i licencyjnych.

Niezadowolone z kształtu HEVC, firmy technologiczne takie jak Google, Mozilla, Microsoft, Apple, etc. założyły konsorcjum *Alliance for Open Media (AOM)*, które w roku 2018 wytworzyło AV1, otwarty i darmowy kodek wideo, będący następcą kodeka VP9 wytworzonego przez Google. AV1 jest aktualnie w fazie adopcji przez dostawców zawartości wideo oraz producentów sprzętu.

AV1 dzięki nowym technikom osiąga lepszą kompresję danych, co ma zastosowanie dla dostawców wideo, którzy dzięki nowemu kodekowi będą w stanie zapewnić oglądającym lepszy obraz jednocześnie zmniejszając obiętość danych do wysłania. Nie jest jednak jasne czy AV1 ma zastosowanie w internetowych komunikatorach wideo pomiędzy dwoma użytkownikami używającymi do transmisji komputerów PC lub urządzeń mobilnych. Najważniejszą rzeczą w połączeniach wideo czasu rzeczywistego jest opóźnienie, jakość obrazu pełni rolę drugorzędną dopóki spełnia ona pewne minimum oczekiwań uczestników. Aby zapewnić wyższy poziom kompresji niezbędne są bardziej złożone i obliczeniowo intensywne algorytmy, co może pogorszyć opóźnienia takiego połączenia. Aby usprawnić proces kompresji/dekompresji używa się także akceleratorów sprzętowych, będących zazwyczaj częścią układu graficznego danego urządzenia, jednak urządzenia wyposażone we wsparcie dla AV1 zaczęły się pojawiać relatywnie niedawno.

Czy zatem AV1 ma zastosowanie do transmisji wideo w czasie rzeczywistym?

1.2. Cel i zakres pracy

Celem niniejszej pracy jest analiza połączeń wideo czasu rzeczywistego w każdym ich etapie, badanie procesów składających się na nie, i wreszcie utworzenie internetowego komunikatora wideo wykorzystującego poznane koncepty i rozwiązania.

Najpierw wykonana zostanie aplikacja webowa wykorzystująca dostępne w przeglądarkach API WebRTC, zapewniające przeglądarkom możliwości obsługiwania strumieni multimedialnych czasu rzeczywistego i pozwalające na nawiązywanie połączeń peer-to-peer z innymi klientami, dzięki wykorzystaniu mechanizmów STUN/TURN. Na przykładzie tej aplikacji, zaprezentowane zostaną procesy i protokoły umożliwiające nawiązywanie połączeń wideo peer-to-peer.

Następnie, za pomocą języka programowania Rust, wykonana zostanie aplikacja okienkowa na systemy Linux, prezentująca na niższym poziomie przechwytywanie obrazu i dźwięku, kompresję strumieni wideo/audio oraz transmisję danych pomiędzy klientami. Aplikacja będzie nawiązywać połączenia wideo peer-to-peer, a także będzie wykorzystywać kodek AV1 do kompresji wideo.

1.3. Układ pracy - DO ZMIANY

W rozdziale 2 omówione zostaną koncepty i metody związane ze strumieniowaniem wideo. Przedstawiony zostanie uproszczony opis procesu transmisji wideo w czasie rzeczywistym, od pobrania klatki obrazu przez kamerę internetową, do wyświetlenia tejże klatki na monitorze rozmówcy. Poruszone zostaną problemy związane z transmisją wideo przez sieć internetową, np. problem ustanowienia kanału P2P pomiędzy klientami za siecią NAT, negocjowanie sesji pomiędzy klientami, adaptive bitrate, etc.

W rozdziale 3 omówiony zostanie projekt WebRTC i jego protokoły składowe, czytelnik dowie się również w jaki sposób WebRTC rozwiązuje problemy poruszone we wcześniejszym rozdziale i tym samym niezwykle upraszcza tworzenie multimedialnych aplikacji webowych.

W rozdziale 4 zaprezentowana zostanie aplikacja webowa wykorzystująca WebRTC, nastąpi ogólny przegląd wykorzystywanych technologii oraz oprogramowania, zaprezentowane zostaną fragmenty kodu źródłowego realizujące kluczowe procesy nawiązywania połączenia poruszone we wcześniejszym rozdziale. Wykonana zostanie dokładna analiza ruchu sieciowego pomiędzy hostami i uzupełniony zostanie proces nawiązywania połączenia WebRTC poruszony we wcześniejszym rozdziale, zobrazowany konkretnym przykładem.

W rozdziale 5 omówione zostaną problemy, generalne mechanizmy i koncepty związane z kompresją wideo, kontenery, strumienie oraz ich muxowanie do kontenerów, rodzaje kodeków wideo, proces enkodowania/dekodowania, sprzętowe oraz programowe implementacje koderów.

W rozdziale 6 poruszone zostaną problemy i rozwiązania związane ze strumieniowaniem AV1. Zrealizowane zostanie "zejście na niższy poziom", co pozwoli na głębsze zapoznanie się z krokami realizowanymi celem przygotowania strumienia wideo do transmisji oraz wyświetlenia przychodzącego wideo na ekranie monitora. Zrealizowane zostanie min.: zastąpienie całości lub części stosu WebRTC rozwiązaniami natywnymi, oferowanymi przez sprzęt lub system operacyjny, wyeliminowanie abstrakcji oferowanych przez przeglądarkę, wybór optymalnego kodera oraz jego parametrów, wybór technologii GUI, zaplanowanie i omówienie procesu strumieniowania z rozdziału 1szego, wraz z elementami które będą go realizowały.

W rozdziale 7 zaprezentowana zostanie wykonana aplikacja desktopowa wykorzystująca biblioteki oraz inne rozwiązania udostępniane przez system operacyjny. Omówienie architektury, wykorzystywanych technologii i fragmentów kodu aplikacji okienkowej. Zaprezentowanie wybranych etapów procesu strumieniowania realizowanych przez aplikację.

Rozdział 2

Koncepcja pracy

2.1. Wymagania projektowe

Końcowym celem projektu jest poznanie procesów kluczowych w internetowych transmisjach audio-wideo oraz wykonanie aplikacji aplikacji okienkowej na systemy Linux pełniącej rolę komunikatora internetowego. Komunikator powinien pozwalać użytkującym go użytkownikom na odkrywanie innych użytkowników i prowadzenie z nimi połączeń audio-wideo. Połączenia audio-wideo pomiędzy użytkownikami będą odbywać się w trybie peer-to-peer, tj. transmisje wideo i audio w tych połączeniach trafiają na drugą stronę połączenia bezpośrenio, bez pośrednictwa serwera, który będzie służył tylko i wyłącznie do dwóch celów: by umożliwiać użytkownikom odkrywanie innych dostępnych użytkowników do których można wykonać połączenie, oraz jako mechanizm początkowej wymiany danych pomiędzy stronami połączenia celem sygnalizacji połączenia oraz późniejszego ustanowienia bezpośredniego kanału wymiany danych peer-to-peer.

2.2. Wymagania funkcjonalne

1. Użytkownik może wybrać imię pod którym widoczny będzie dla innych użytkowników
2. Użytkownik widzi listę aktualnie dostępnych użytkowników
3. Użytkownik X może zadzwonić do użytkownika Y
4. Użytkownik X w trakcie oczekiwania na odpowiedź od użytkownika Y, może zrezygnować z połączenia, rozłączyć się
5. Użytkownik Y, gdy dzwoni do niego użytkownik X, może odrzucić lub odebrać połączenie
6. Jeśli połączenie zostanie odebrane i poprawnie nawiązane, każdy z użytkowników powinien móc zaobserwować transmisję z kamery wideo oraz usłyszeć transmisję audio z mikrofonu drugiego użytkownika
7. W trakcie trwania połączenia, każdy z użytkowników może rozłączyć się, unilateralnie terminując połączenie.

2.3. Wymagania niefunkcjonalne

1. Do realizacji projektu zostanie wykorzystany język programowania Rust
2. Aplikacja wykorzystuje frameworki i biblioteki natywne dla systemów Linux
3. Aplikacja powinna działać nie tylko w sieci lokalnej, ale także w sieci Internet
4. Aplikacja wykorzystuje kodek AV1 do kompresji wideo

5. Aplikacja powinna wdzięcznie obsługiwać brak kamery wideo lub mikrofonu przez użytkownika
6. Połączenia pomiędzy użytkownikami powinny odbywać się w trybie peer-to-peer celem minimalizacji opóźnień
7. Z powodu powyższego wymagania, połączenia są ograniczone do dwóch użytkowników, tj. nie ma możliwości tworzenia konferencji z wieloma użytkownikami.

Rozdział 3

Zagadnienia teoretyczne

3.1. Kodowanie wideo

https://github.com/leandromoreira/digital_video_introduction#basic-terminology

3.1.1. Podstawy

An image can be thought of as a 2D matrix. If we think about colors, we can extrapolate this idea seeing this image as a 3D matrix where the additional dimensions are used to provide color data.

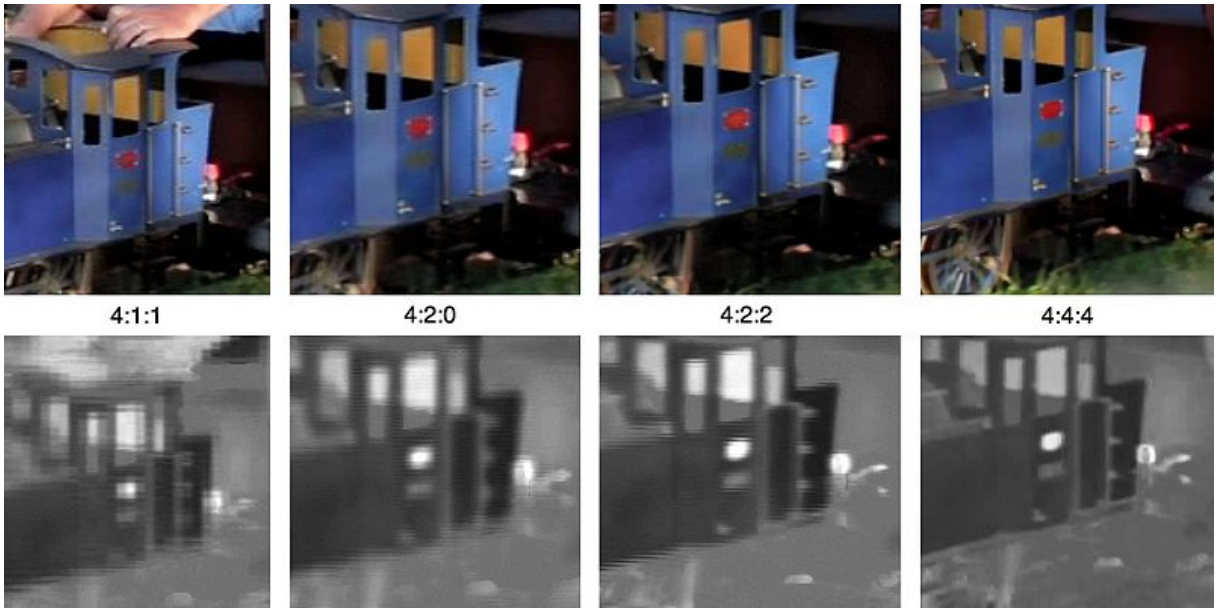
If we chose to represent these colors using the primary colors (red, green and blue), we define three planes: the first one for red, the second for green, and the last one for the blue color.

We'll call each point in this matrix a pixel (picture element). One pixel represents the intensity (usually a numeric value) of a given color. For example, a red pixel means 0 of green, 0 of blue and maximum of red. The pink color pixel can be formed with a combination of the three colors. Using a representative numeric range from 0 to 255, the pink pixel is defined by Red=255, Green=192 and Blue=203.

3.1.2. Mechanizmy kompresji

Chroma subsampling

Our eyes are more sensitive to brightness than colors. With the image represented as luma and chroma components, we can take advantage of the human visual system's greater sensitivity for luma resolution rather than chroma to selectively remove information. Chroma subsampling is the technique of encoding images using less resolution for chroma than for luma.



Rys. 3.1: Przykład różnych typów chroma subsampling. Pomimo bardzo niskiego próbkowania informacji o kolorze, w górnym rzędzie nie widać znaczących różnic pomiędzy pierwszym i ostanim obrazkiem.

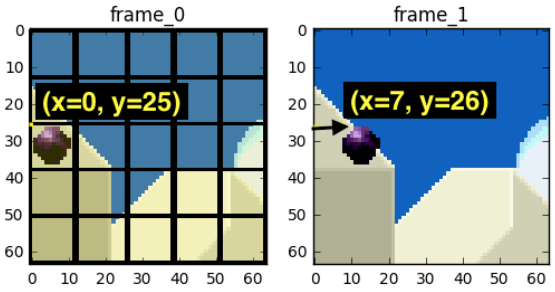
Now we can move on and try to eliminate the redundancy in time.

3.1.3. Inter-frame coding

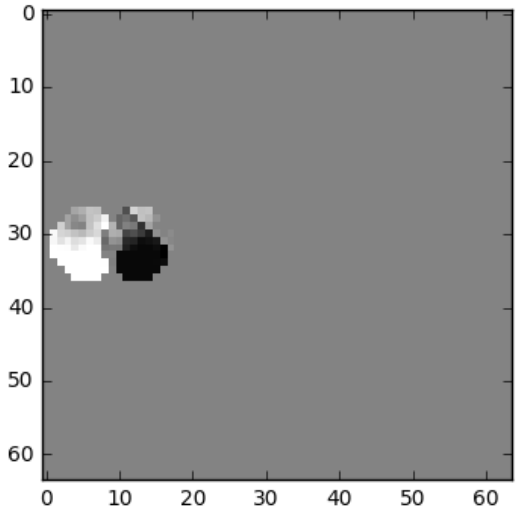
Inter-frame coding is reusing information between frames.

Let's explore the options we have to reduce the repetitions in time, this type of redundancy can be solved with techniques of inter prediction.

We will try to spend fewer bits to encode the sequence of frames 0 and 1.



One thing we can do it's a subtraction, we simply subtract frame 1 from frame 0 and we get just what we need to encode the residual.

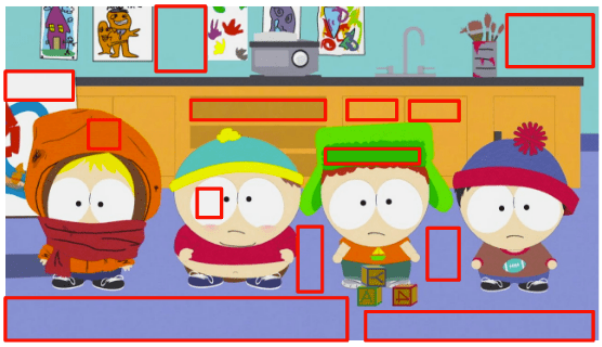


But what if I tell you that there is a better method which uses even fewer bits?! First, let's treat the frame 0 as a collection of well-defined partitions and then we'll try to match the blocks from frame 0 on frame 1. We can think of it as motion estimation.

Also motion vectors.

3.1.4. Intra-frame coding

If we analyze each frame in a video we'll see that there are also many areas that are correlated.



3.1.5. Standardy kodowania

Podstawy

AVC

AVC jest standardem kodowania wideo przyjętym w roku 2003 przez MPEG. Jego najpopularniejszą implementacją jest biblioteka x264.

AV1

3.2. Internetowe strumienie wideo

Jakieś wprowadzenie o wideo ogólnie, w jakich kontekstach występuje (pliki wideo, streaming np. Youtube/Netflix, strumienie czasu rzeczywistego np. Twitch/Discord/Teams), czym te konteksty się różnią.

Możemy wyróżnić 3 rodzaje wideo:

- lokalne pliki wideo - pliki wideo na dysku w kontenerze mp4, mkv, lub innym
- streamowanie plików wideo (np. Youtube albo Netflix) - mamy przygotowane segmenty pliku wideo w różnych rozdzielczościach i wysyłamy je po kolei, dobieramy rozdzielczość wg. dostępnego pasma pomiędzy serwerem a klientem
- strumieniowanie w czasie rzeczywistym - priorytetem jest opóźnienie, kodujemy na bieżąco przechwytywane klatki i wysyłamy je najszybciej jak się da; przez to niektóre mechanizmy kompresji są niedostępne (np. B-klatki, które wykorzystują dane z następnej klatki aby zmniejszyć wielkość klatki)

Zidealizowany obraz rozmowy wideo w internecie:

1. Komputery są publicznymi hostami w internecie i program komunikatora słucha na danym porcie
2. Strona nawiązująca połączenie łączy się do hosta odbiorcy po tym porcie, sygnalizuje chęć nawiązania połączenia
3. Strona odbierająca akceptuje
4. Kamera oraz mikrofon nadawcy przechwytują najlepszy możliwy obraz i dzwięk, i przesyłają je do komputera
5. Strumienie wideo i audio są łączone i synchronizowane
6. Komputer wysyła strumień audio-wideo wcześniej ustanowionym kanałem

Natomiast pojawiają się problemy:

1. Komputery znajdują się w sieciach domowych, za NATem, nie można się do nich bezpośrednio połączyć
2. Nieskompresowane wideo jest zbyt duże by wysłać je przez internet, potrzebny jest jakiś mechanizm kompresji
3. Komputery mogą mieć różne możliwości przetwarzania wideo: znajdować się w sieciach o znacząco różnej szybkości, mieć różniące się szybkością procesory, kamery zapisujące klatki w różnych formatach

W następnym rozdziale zaprezentowane zostaną technologie rozwiązujące powyższe problemy.

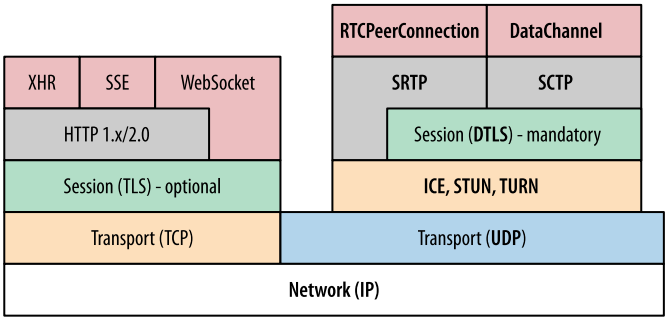
Rozdział 4

Wykorzystane technologie

4.1. WebRTC

[1]

WebRTC jest standardem zapewniającym przeglądarkom możliwości komunikacji peer-to-peer w czasie rzeczywistym, dzięki czemu możliwe jest budowanie komunikatorów internetowych w całości w obrębie zapewnianego przez przeglądarkę javascriptowego API. Aby funkcjonalność ta była możliwa do zrealizowania, niezbędne było użycie wielu protokołów sieciowych, które pokazano na rysunku 4.1.



Rys. 4.1: Stos protokołów w WebRTC - wzięty z HPBN - do zmiany

Głównym targetem WebRTC są przeglądarki, jednak istnieją także implementacje dla innych typów aplikacji. W podrozdziale 4.2.1 opisano implementację dla frameworka GStreamer.

WebRTC pomaga rozwiązać 3 problemy niezbędne do nawiązania multimedialnego połączenia peer-to-peer:

1. W jaki sposób zasygnalizować peerowi chęć nawiązania połączenia tak by zaczął on nasłuchiwać na wysyłane do niego dane?
2. Jak wynegocjować odpowiednie parametry strumieni multimedialnych tak by odpowiadały one obu stronom?
3. Jak zidentyfikować potencjalne trasy/tryby połączenia dla obu jego stron?

4.1.1. Sygnalizacja nawiązania połączenia

Before any connectivity checks or session negotiation can occur, we must find out if the other peer is reachable and if it is willing to establish the connection. We must extend an offer, and the peer must return an answer (Figure 18-5). However, now we have a dilemma: if the other peer is

not listening for incoming packets, how do we notify it of our intent? At a minimum, we need a shared signaling channel.

Rys. 4.2: Strony sygnalizacji - wzięty z HPBN - do zmiany

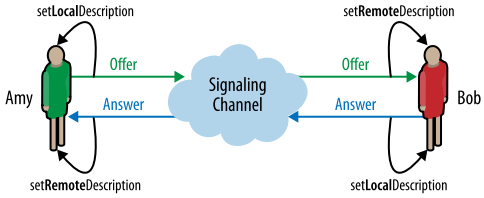
WebRTC defers the choice of signaling transport and protocol to the application; the standard intentionally does not provide any recommendations or implementation for the signaling stack. Why? This allows interoperability with a variety of other signaling protocols powering existing communications infrastructure.

W wypadku naszej aplikacji, rola kanału sygnalizacyjnego będzie pełniona przez serwer aplikacji.

4.1.2. Negocjacja parametrów połączenia

WebRTC uses Session Description Protocol (SDP) to describe the parameters of the peer-to-peer connection. SDP does not deliver any media itself; instead it is used to describe the "session profile,"which represents a list of properties of the connection: types of media to be exchanged (audio, video, and application data), network transports, used codecs and their settings, bandwidth information, and other metadata.

To establish a peer-to-peer connection, both peers must follow a symmetric workflow (rysunek 4.3) to exchange SDP descriptions of their respective audio, video, and other data streams.



Rys. 4.3: Proces negocjacji - wzięty z HPBN - do zmiany

1. The initiator (Amy) registers one or more streams with her local RTCPeerConnection object, creates an offer, and sets it as her "local description"of the session.
2. Amy then sends the generated session offer to the other peer (Bob).
3. Once the offer is received by Bob, he sets Amy's description as the "remote description"of the session, registers his own streams with his own RTCPeerConnection object, generates the "answer"SDP description, and sets it as the "local description"of the session—phew!
4. Bob then sends the generated session answer back to Amy.
5. Once Bob's SDP answer is received by Amy, she sets his answer as the "remote description"of her original session.

With that, once the SDP session descriptions have been exchanged via the signaling channel, both parties have now negotiated the type of streams to be exchanged, and their settings. We are almost ready to begin our peer-to-peer communication! Now, there is just one more detail to take care of: connectivity checks and NAT traversal.

4.1.3. Interactive Connectivity Establishment (ICE)

In order to establish a peer-to-peer connection, by definition, the peers must be able to route packets to each other. A trivial statement on the surface, but hard to achieve in practice due to the numerous layers of firewalls and NAT devices between most peers.

First, let's consider the trivial case, where both peers are located on the same internal network, and there are no firewalls or NATs between them. To establish the connection, each peer can simply query its operating system for its IP address (or multiple, if there are multiple network interfaces), append the provided IP and port tuples to the generated SDP strings, and forward it to the other peer. Once the SDP exchange is complete, both peers can initiate a direct peer-to-peer connection.

So far, so good. However, what would happen if one or both of the peers were on distinct private networks? We could repeat the preceding workflow, discover and embed the private IP addresses of each peer, but the peer-to-peer connections would obviously fail! What we need is a public routing path between the peers. Thankfully, the WebRTC framework manages most of this complexity on our behalf:

- Each RTCPeerConnection connection object contains an "ICE agent."
- ICE agent is responsible for gathering local IP, port tuples (candidates).
- ICE agent is responsible for performing connectivity checks between peers.
- ICE agent is responsible for sending connection keepalives.

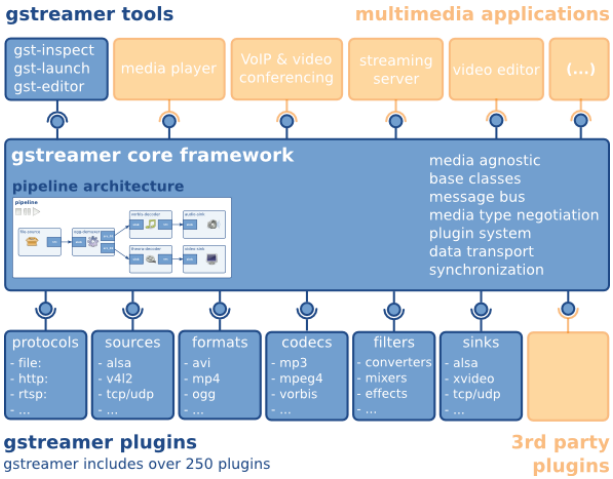
Once a session description (local or remote) is set, local ICE agent automatically begins the process of discovering all the possible candidate IP, port tuples for the local peer:

1. ICE agent queries the operating system for local IP addresses.
2. If configured, ICE agent queries an external STUN server to retrieve the public IP and port tuple of the peer.
3. If configured, ICE agent appends the TURN server as a last resort candidate. If the peer-to-peer connection fails, the data will be relayed through the specified intermediary.

As the example illustrates, the ICE agent handles most of the complexity on our behalf: the ICE gathering process is triggered automatically, STUN lookups are performed in the background, and the discovered candidates are registered with the RTCPeerConnection object. Once the process is complete, we can generate the SDP offer and use the signaling channel to deliver it to the other peer.

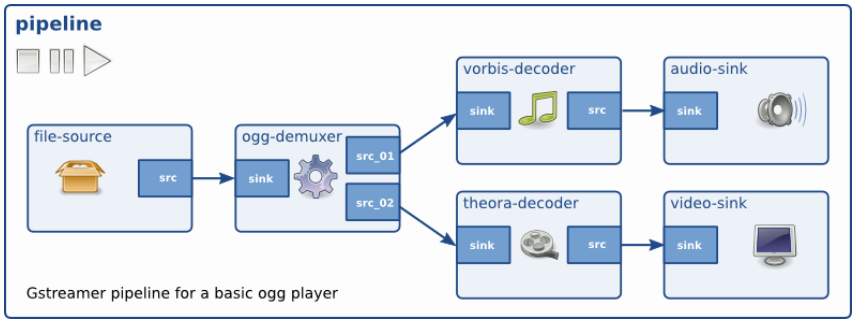
4.2. GStreamer

GStreamer jest frameworkiem do tworzenia strumieniujących aplikacji multimedialnych [2]. Aplikacje są wyrażane jako "rurociąg"składający się z bloków. Rdzeń GStreamer definiuje architekturę aplikacji oraz API dla elementów, a same elementy są zawarte w różnych pluginach. Jego modularna architektura sprawia że nie jest ograniczony do aplikacji audio-wideo, lecz może być wykorzystany do każdej aplikacji, która strumieniowo przetwarza dowolne rodzaje danych.



Rys. 4.4: Architektura projektu GStreamer

Dane generowane są elementach-źródłach (source), są przetwarzane przez elementy-filtry (filter), a następnie są konsumowane w elementach-zlewach (sinks). Elementy są częścią rurociągu, który w danym czasie może być zatrzymany, zapauzowany, lub uruchomiony. Zmiana stanu rurociągu zmienia stan wszystkich jego elementów.



Rys. 4.5: Przykładowy rurociąg aplikacji GStreamer

Rysunek 4.5 przedstawia rurociąg przykładowego odtwarzacza plików OGG. Dane przechodzą przez rurociąg od lewej do prawej strony, będąc transformowane w każdym elemencie.

4.2.1. GStreamer WebRTC

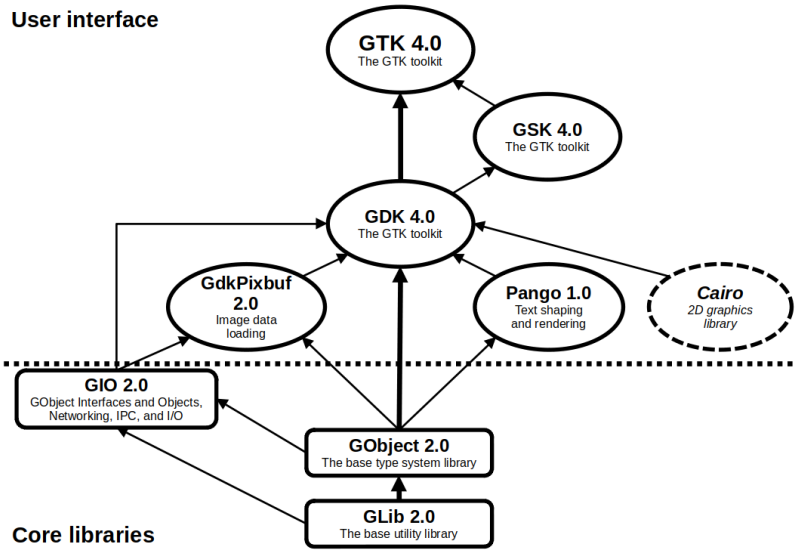
GStreamer posiada swoją własną implementację WebRTC, wykonaną przez firmę Centricular. Jej głównym elementem jest element webrtcbin. Ten element zajmuje się procesami negocjacji parametrów połączenia oraz zbierania i udostępniania kandydatów ICE omówionymi w podrozdziałach 4.1.2 i 4.1.3. Zdarzenia takie jak pojawienie się nowego strumienia multimedialnego są komunikowane aplikacji za pomocą sygnałów, które są obsługiwane zazwyczaj poprzez tworzenie nowych elementów i dodanie ich do rurociągu celem prezentacji przychodzącego strumienia użytkownikowi.

4.3. GTK4 i libadwaita

Do wykonania aplikacji graficznej wykorzystany został darmowy, otwartoźródłowy, wieloplatformowy framework GTK4 oraz biblioteka libadwaita implementująca często wykorzystywane

komponenty i wzorce projektowe będące częścią wytycznych projektu GNOME w projektowaniu interfejsów (GNOME Human Interface Guidelines).

GTK jest najpopularniejszą biblioteką używaną do tworzenia graficznych aplikacji dla systemów Linux. Napisana w C, wykorzystuje jednak paradygmat obiektowy; używa biblioteki GObject która zapewnia system klas i obiektów, a także inne zależności widoczne na rysunku 4.6.



Rys. 4.6: Architektura zestawu narzędzi GTK

Bezpośrednie używanie tych bibliotek w języku Rust nie jest jednak rekomendowane, i gdzie tylko możliwe wykorzystywane są biblioteki natywne dla języka Rust. Nie można ich jednak wyeliminować lub zastąpić, ponieważ są twardymi zależnościami frameworka GUI.

Pomimo tego, GTK zostało wybrane z dwóch powodów:

- GTK jest najpopularniejszym, najbardziej sprawdzonym oraz dojrzałym frameworkiem GUI dla systemów Linux
- GStreamer i GTK są częściami tego samego projektu freedesktop i w związku z tym interoperują ze sobą; m.in. wykorzystują te same biblioteki GLib i GObject, a także GStreamer może rysować zawartość strumienia mediów na powierzchnię jeżeli ta implementuje interfejs `GstVideoOverlay`.

4.4. Rust

Rust jest językiem programowania generalnego zastosowania rozwijanym przez fundację Mozilla. Stworzony z myślą o bezpieczeństwie, współbieżności, i praktyczności, zapewnia wydajność bliską języka C jednocześnie gwarantując bezpieczeństwo pamięci nie wykorzystując przy tym mechanizmu Garbage Collection. Zamiast tego Rust wykorzystuje "borrow checker", który śledzi czas życia referencji do obiektów podczas kompilacji. Dzięki temu niektóre klasy błędów są niemożliwe do popełnienia, dzięki czemu programista może wykorzystywać wielowątkowość bez obaw przed ezoterycznymi i nietrywialnymi do zdebugowania błędami typu race-condition. Ponadto dzięki darmowym abstrakcjom, kod w języku Rust jest czytelny jak język wysokopoziomowy, jednocześnie zapewniając wydajność charakterystyczną zazwyczaj tylko dla języków niskopoziomowych.

4.5. Programowanie asynchroniczne

Programowanie asynchroniczne jest paradygmatem programowania umożliwiającym konkurentne wykonywanie wielu zadań bez użycia mechanizmów wielowątkowości. Zamiast tego, główny wątek programu może zapisać stan aktualnie wykonywanego zadania i zająć się wykonywaniem innego zadania. Paradygmat ten jest popularny przy programowaniu serwerów, ponieważ wielowątkowość charakteryzowała się wieloma wadami:

- Wątki są powolne do tworzenia, i zajmują pewną minimalną ilość pamięci - zadania asynchroniczne można tworzyć bardzo szybko i zajmują one o wiele mniej pamięci
- Użycie wielu wątków może prowadzić do błędów typu race-condition - zadania asynchroniczne mogą być przypisane do jednego wątku eliminując ten problem
- Jeżeli używana jest bardzo duża liczba wątków, overhead systemu operacyjnego staje się znaczący i wydajność systemu może ucierpieć - w systemie asynchronicznym planowanie zadań jest prostsze ponieważ dzieje się w userspace przez egzekutor systemu asynchronicznego, nie występują zatem drogie context-switche, a także egzekutor lepiej zna charakterystyki wykonywanego kodu więc może planować tak by zużycie zasobów było mniejsze
- Większość sytuacji wymagających konkurentności w serwerach to czekanie na jakieś zdarzenie: czekanie na połączenie, czekanie na odebranie danych od klienta, etc. tzw. IO-bound. W takiej sytuacji nie ma potrzeby zwiększać złożoności systemu angażując kolejne fizyczne procesory i pamięć, zamiast tego wątek może w tym czasie pracować nad innymi zadaniami.

Ekosystem programowania asynchronicznego składa się z asynchronicznych tzw. runtimes zawierających egzekutory zadań asynchronicznych oraz samych zadań asynchronicznych znanych jako Futures. Future (ang. przyszłość) reprezentuje operacje która zwróci wartość w przyszłości i musi być w tym celu możliwie wielokrotnie wykonywana funkcją `poll()`, która może zwrócić wariant `Poll::Ready(T)` reprezentujący zakończenie działania i zwrócenie wartości, lub wariant `Poll::Pending` wskazujący że wykonany został pewien postęp w wykonaniu, jednak funkcja będzie musiała zostać wykonana ponownie.

Sam egzekutor wykorzystuje asynchroniczne system calle takie jak epoll, które sprawiają że gdy wydarzy się zdarzenie, w kontekście future wywołuje się funkcja `wake()`, która powiadamia egzekutor o tym że dana future może być pollnięta ponownie.

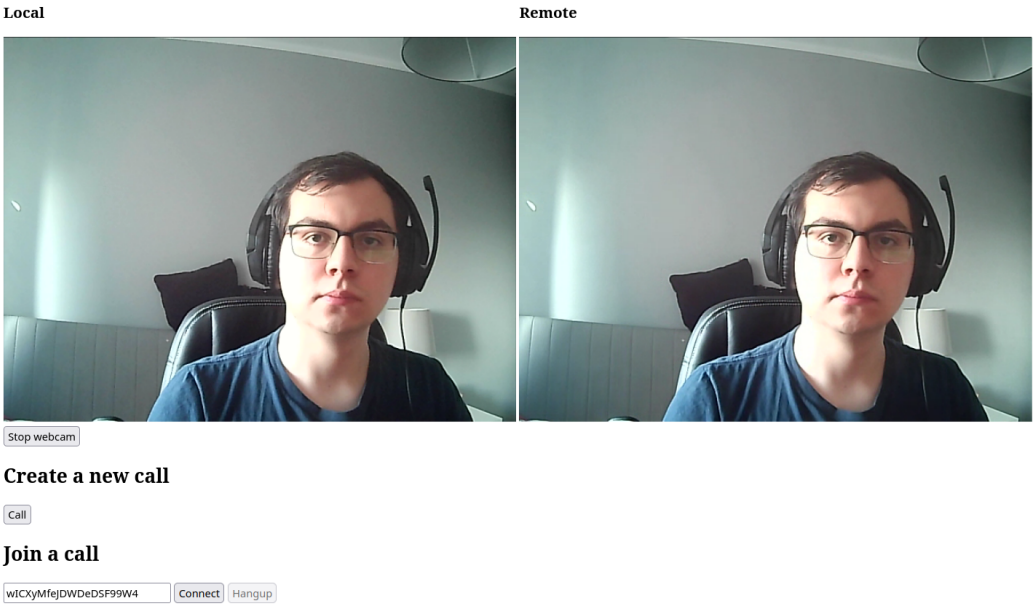
Overhead takiego podejścia jest mały i działa ono bardzo dobrze, dopóki futures są IO-bound, tj. większość czasu to czas spędzony na czekanie na IO. Gdy futures są CPU-bound, tj. wykonują dużą ilość kalkulacji których zakończenie zajmie znaczącą ilość czasu (np. kilka sekund), lub używają synchronicznych blokujących API, wtedy taka future jest w stanie zablokować egzekutor i inne futures nie będą w stanie się wykonać. W tym celu asynchroniczne runtimes zazwyczaj udostępniają specjalny thread pool na zadania blokujące główny wątek. Dodatkowo, egzekutor zazwyczaj też wykorzystuje threadpool do uruchamiania futures, dzięki czemu mogą być one wykonywane nie tylko konkurentnie lecz także równolegle. Dzięki gwarancjom bezpieczeństwa i systemowi ownership w języku Rust, nie powoduje to ryzyka wystąpienia błędów związanych z wielowątkowością. To czy future może zostać wysłana pomiędzy wątkami jest wiadome już w procesie kompilacji, dzięki traitowi `Send`. Funkcje egzekutorów typu `task::spawn()` służące do rozpoczęcia konkurentnego wykonania danego future wymagają aby była ona `Send`, a jeżeli nie jest, emitowany jest błąd kompilacji. Istnieją także funkcje jak `task::block_on()` które synchronicznie wykonują dany future na jednym wątku.

W niniejszym projekcie gdy tylko możliwe preferowane jest używanie asynchronicznych API. Wykorzystywane są oba najpopularniejsze runtime'y asynchroniczne w języku Rust: tokio oraz async-std.

Rozdział 5

Aplikacja webowa z użyciem WebRTC

W poniższym rozdziale zostanie omówiona aplikacja webowa wykorzystująca WebRTC.

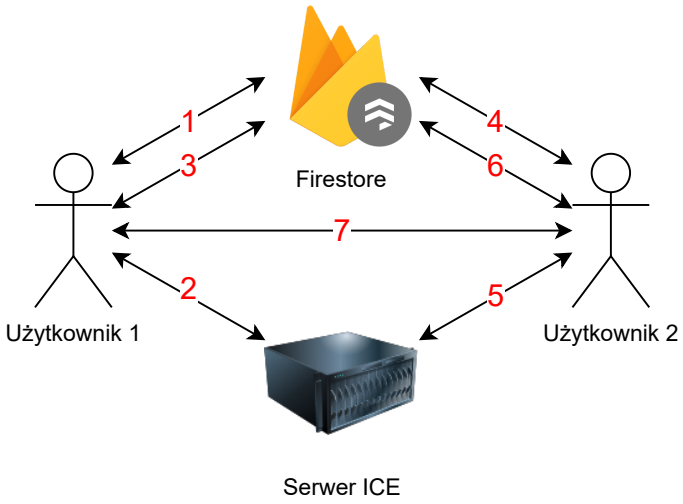


Rys. 5.1: Zrzut ekranu aplikacji WebRTC podczas połączenia na tym samym komputerze

Aby nawiązać połączenie, trzeba wykonać następujące kroki:

1. Użytkownicy 1 i 2 wciskają przycisk **Start webcam** aby udostępnić obraz z kamery aplikacji.
2. Użytkownik 1 wciska przycisk **Call**, tworząc nowe połączenie.
3. Użytkownik 1 odczytuje ID połączenia które pojawiło się w polu tekstowym i przekazuje je użytkownikowi 2.
4. Użytkownik 2 wprowadza ID połączenia uzyskane od użytkownika 1 w to samo pole tekstowe i wciska przycisk **Connect**.

Następnie odbywa się proces nawiązywania połączenia zaprezentowany na diagramie 5.2.



Rys. 5.2: Diagram prezentujący proces nawiązywania połączenia

1. Użytkownik 1 tworzy ofertę połączenia oraz wysyła ją do bazy Firestore. Rozpoczyna także proces nasłuchiwania odpowiedzi i kandydatów ICE drugiej strony.
2. Użytkownik 1 rozpoczyna proces odkrywania kandydatów ICE.
3. Użytkownik 1 wysyła na bieżąco do Firestore otrzymywanych kandydatów ICE (trickle ICE).
4. Użytkownik 2 odczytuje z bazy ofertę użytkownika 1, generuje na nią odpowiedź, i wysyła ją do Firestore.
5. Użytkownik 2 rozpoczyna proces odkrywania kandydatów ICE.
6. Użytkownik 2 wysyła na bieżąco do Firestore otrzymywanych kandydatów ICE (trickle ICE).
7. Świadomi oferty, odpowiedzi, oraz kandydatów ICE drugiej strony, użytkownicy mogą nawiązać połączenie peer-to-peer (lub w sytuacjach kiedy połączenie peer-to-peer jest niemożliwe, łączą się używając serwera TURN jako pośrednika).

5.1. Architektura

Aplikacja składa się z następujących technologii:

- **Frontend**: Node.js jako środowisko, Vite jako transpilator JS, Typescript jako język programowania
- **Backend**: Firestore z platformy Google Firebase jako pośrednik między klientami w procesie nawiązywania połączenia. Firestore jest przede wszystkim bazą danych NoSQL, jednak oferowana przez nią funkcjonalność nasłuchiwania dokumentów oraz otrzymywania ich aktualizacji w czasie rzeczywistym sprawia, że może być zastosowana w tym celu, co zwalnia programistę z obowiązku przygotowania, utrzymywania i zarządzania serwerem.
- **WebRTC**: Do nawiązywania połączeń wykorzystywane jest API WebRTC dostępne w przeglądarkach internetowych. Do realizacji procesu ICE, pozwalającemu stronom na zebranie możliwych ścieżek połączenia peer-to-peer, a także w wypadku jego niepowodzenia skorzystanie z serwera TURN jako pośrednika, skorzystano z serwerów oferowanych przez Open Relay

5.2. Wybrane fragmenty kodu

5.2.1. Przechwytywanie wideo i audio

Listing 5.1: Przechwytywanie wideo i audio z komputera

```
let localStream: MediaStream;
const webcamButton = document.getElementById('webcamButton');

webcamButton?.addEventListener('click', async () => {
    if (localVideo.srcObject) {
        localVideo.srcObject = null;
        return;
    }

    localStream = await navigator.mediaDevices.getUserMedia({ video: true,
    ↪ audio: true, });
    webcamButton.innerHTML = 'Stop webcam';

    remoteStream = new MediaStream();
    remoteVideo.srcObject = remoteStream;


    localStream.getTracks().forEach((track) => { peerConnection.addTrack(
    ↪ track, localStream); });
    peerConnection.addEventListener('track', event => {
        event.streams[0].getTracks().forEach(track => {
            remoteStream.addTrack(track);
        });
    });

    localVideo.srcObject = localStream;
    remoteVideo.srcObject = remoteStream;

    callButton.disabled = false;
    answerButton.disabled = false;
});
```

5.2.2. Tworzenie połączenia

Aby utworzyć połączenie WebRTC, musimy najpierw skomunikować się z drugim klientem przez jakiś inny kanał, aka. out-of-band, wykorzystamy do tego celu bazę danych czasu rzeczywistego Firebase. Przygotujemy zatem uchwyt do bazy danych:

Listing 5.2: Inicjalizacja Firebase

```
import { initializeApp } from "firebase/app";
import { getFirestore, collection, addDoc, getDoc, doc, setDoc, onSnapshot,
    ↪  updateDoc } from "firebase/firestore";


// Your web app's Firebase configuration
const firebaseConfig = {
    apiKey: "AIzaSyCr12-OQV5bgdQPFoexd44O9Ubmht966pw",
    authDomain: "piperchat-2eacd.firebaseapp.com",
    projectId: "piperchat-2eacd",
    storageBucket: "piperchat-2eacd.appspot.com",
    messagingSenderId: "172730710087",
    appId: "1:172730710087:web:3dabdb9a62bee44e095962"
```

```
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
const db = getFirestore(app);
```

Następnie, do przycisku **Call** tworzącego połączenie podpinamy handler:

Listing 5.3: Tworzenie połączenia

```
callButton?.addEventListener('click', async () => {
  const callDoc = doc(collection(db, "calls"));
  const offerCandidates = collection(callDoc, "offerCandidates");
  const answerCandidates = collection(callDoc, "answerCandidates");

  callInput.value = callDoc.id;

  peerConnection.onicecandidate = (event) => {
    if (event.candidate) {
      addDoc(offerCandidates, event.candidate.toJSON());
    }
  }

  const offerDescription = await peerConnection.createOffer();
  await peerConnection.setLocalDescription(offerDescription);

  const offer = {
    sdp: offerDescription.sdp,
    type: offerDescription.type
  };

  await setDoc(callDoc, { offer });

  onSnapshot(callDoc, (snapshot) => {
    const data = snapshot.data();
    if (!peerConnection.currentRemoteDescription && data?.answer) {
      const answerDescription = new RTCSessionDescription(data.answer);
      peerConnection.setRemoteDescription(answerDescription);
    }
  });

  onSnapshot(answerCandidates, (snapshot) => {
    snapshot.docChanges().forEach((change) => {
      if (change.type === "added") {
        const candidate = new RTCIceCandidate(change.doc.data());
        peerConnection.addIceCandidate(candidate);
      }
    })
  })
});
```

5.2.3. Dołączanie do połączenia

Listing 5.4: Dołączanie do połączenia połączenia

```
answerButton?.addEventListener("click", async () => {
  const callId = callInput.value;
  const callDoc = doc(db, "calls", callId);
  const answerCandidates = collection(callDoc, "answerCandidates");
  const offerCandidates = collection(callDoc, "offerCandidates");
```

```
  peerConnection.onicecandidate = (event) => {
    if (event.candidate) {
      addDoc(answerCandidates, event.candidate.toJSON());
    }
  }

  const callData = (await getDoc(callDoc)).data();
  if (!callData) {
    console.error("Call document no longer exists");
    return;
  }
  const offerDescription = callData.offer;
  await peerConnection.setRemoteDescription(new RTCSessionDescription(
     ↪ offerDescription));

  const answerDescription = await peerConnection.createAnswer();
  await peerConnection.setLocalDescription(answerDescription);

  const answer = {
    type: answerDescription.type,
    sdp: answerDescription.sdp,
  };

  await updateDoc(callDoc, { answer });

  onSnapshot(offerCandidates, (snapshot) => {
    snapshot.docChanges().forEach((change) => {
      if (change.type === "added") {
        const data = change.doc.data();
        peerConnection.addIceCandidate(new RTCIceCandidate(data));
      }
    })
  });
});
```

5.3. Omówienie działania aplikacji na przykładzie

5.3.1. Śledzenie procesu nawiązywania połączenia

W poniższym rozdziale zostaną omówione dane wymieniane pomiędzy stronami na rzecz ustanowienia połączenia WebRTC.

Aby ustanowić połączenie peer-to-peer, należy rozwiązać poniższe problemy: [1]

- Należy powiadomić drugą stronę że chcemy ustanowić do niej połączenie, aby wiedziała ona żeby rozpocząć nasłuchiwanie.
- Należy zidentyfikować ścieżki trasowania dla połączenia peer-to-peer i uzgodnić jedną pomiędzy obiema stronami.
- Należy wymienić niezbędne informacje o używanych przez peerów parametrach połączenia - jakich protokołów, kodeków, ustawień, etc. użyć.

W aplikacji webowej, użytkownik 1 chacący utworzyć nowe połączenie tworzy dokument w kolekcji calls. ID rozmowy to ID dokumentu utworzonego w bazie Firestore. Obu użytkowników przeprowadzi początkową wymianę danych pisząc do oraz czytając z tego dokumentu.

Użytkownik 1 tworzy zatem nowy dokument w bazie:

Listing 5.5: Dokument połączenia po utworzeniu przez użytkownika 1

```
{
    id: "YHARFdJoA4lAd8nRu3Uw",
}
```

Następnie użytkownik 1 tworzy ofertę, czyli opis połączenia, pozwalający użytkownikowi 2 na połączenie się:

Listing 5.6: Dokument połączenia po dodaniu opisu sesji w protokole SDP

```
{
    id: "YHARFdJoA4lAd8nRu3Uw",
  offer: "v=0o=mozilla...THIS_IS_SDPARTA-99.0 8615225844821133956 0 IN IP4
      ↪ 0.0.0.0s=-t=0 0a=fingerprint:sha-256 5F:A8:8A:A5:B8:1D:0C:39:21:93:
      ↪ FA:3A:B2:B7:B6:3F:EF:8A:5D:3C:6E:86:2E:A7:0A:D4:F0:E3:58:E0:E2:7B
      ↪ ..."
}
```

Równocześnie, użytkownik 1 rozpoczyna wyszukiwanie kandydatów ICE (Interactive Connecivity Establishment), czyli sposobów na umożliwienie drugiej stronie do nawiązania ze sobą połączenia (problem nr 2):

Listing 5.7: Dokument połączenia po dodaniu kandydatów ICE

```
{
    id: "YHARFdJoA4lAd8nRu3Uw",
  offer: "v=0o=mozilla...THIS_IS_SDPARTA-99.0 8615225844821133956 0 IN IP4
      ↪ 0.0.0.0s=-t=0 0a=fingerprint:sha-256 5F:A8:8A:A5:B8:1D:0C:39:21:93:
      ↪ FA:3A:B2:B7:B6:3F:EF:8A:5D:3C:6E:86:2E:A7:0A:D4:F0:E3:58:E0:E2:7B
      ↪ ...",
  offerCandidates: [
      {
          "candidate": "",
          "sdpMid": "0",
          "usernameFragment": "0b863d52",
          "sdpMLineIndex": 0
      },
      {
          "sdpMLineIndex": 0,
          "sdpMid": "0",
          "candidate": "candidate:1 2 UDP 1686052862 188.122.20.104 42436
              ↪ typ srflx raddr 192.168.1.2 rport 42436",
          "usernameFragment": "0b863d52"
      },
      {
          "sdpMid": "1",
          "candidate": "",
          "sdpMLineIndex": 1,
          "usernameFragment": "0b863d52"
      },
      {
          "sdpMLineIndex": 1,
          "sdpMid": "1",
          "candidate": "candidate:1 2 UDP 1686052862 188.122.20.104 44466
              ↪ typ srflx raddr 192.168.1.2 rport 44466",
          "usernameFragment": "0b863d52"
      },
      ...
  ]

}
```

Na koniec, użytkownik 1 wysłuchuje zmian w dokumencie sygnalizujących próbę nawiązania połączenia. Dokładniej, użytkownik 1 oczekuje na pojawienie się, analogicznie do `offer` i `offerCandidates`, pól `answer` oraz `answerCandidates`. Zawartość tych pól trafi do obiektu `RTCPeerConnection`, które zajmie się ustanowieniem połączenia.

5.3.2. Analiza pakietów protokołu SDP

Parametry ustanowionego połączenia są determinowane przez protokół SDP. Zobaczmy zatem pakiet SDP z poprzedniego podrozdziału:

Listing 5.8: Opis oferty połączenia SDP

```
v=0
o=mozilla...THIS_IS_SDPARTA-99.0 107455341790422027 0 IN IP4 0.0.0.0
s=-
t=0 0
a=fingerprint:sha-256 BB:64:A1:DA:F1:E3:93:63:7A:65:E5:55:EA:FC:E9:1F:B6
    ↪ :43:1D:95:6B:2D:CC:34:3B:67:C9:EB:EE:80:43:3D
a=group:BUNDLE 0 1
a=ice-options:trickle
a=msid-semantic:WMS *
m=audio 9 UDP/TLS/RTP/SAVPF 109 9 0 8 101
c=IN IP4 0.0.0.0
a=sendrecv
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=extmap:2/recvonly urn:ietf:params:rtp-hdrext:csrc-audio-level
a=extmap:3 urn:ietf:params:rtp-hdrext:sdes:mid
a=fmtp:109 maxplaybackrate=48000;stereo=1;useinbandfec=1
a=fmtp:101 0-15
a=ice-pwd:efe34e413f35fb225352e73b89d2fa73
a=ice-ufrag:0b863d52
a=mid:0
a=msid:{89963797-b150-406b-8c17-d3a02e5ab84b} {d20d186b-1018-43b9-805d-
    ↪ a8cd94a4e7af}
a=rtcp-mux
a=rtpmap:109 opus/48000/2
a=rtpmap:9 G722/8000/1
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000/1
a=setup:actpass
a=ssrc:167358539 cname:{a77c69ce-8bfc-4cf6-ac3c-0db7976f374a}
m=video 9 UDP/TLS/RTP/SAVPF 120 124 121 125 126 127 97 98
c=IN IP4 0.0.0.0
a=sendrecv
a=extmap:3 urn:ietf:params:rtp-hdrext:sdes:mid
a=extmap:4 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
a=extmap:5 urn:ietf:params:rtp-hdrext:toffset
a=extmap:6/recvonly http://www.webrtc.org/experiments/rtp-hdrext/playout-
    ↪ delay
a=extmap:7 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-
    ↪ extensions-01
a=fmtp:126 profile-level-id=42e01f;level-asymmetry-allowed=1;packetization-
    ↪ mode=1
a=fmtp:97 profile-level-id=42e01f;level-asymmetry-allowed=1
a=fmtp:120 max-fs=12288;max-fr=60
a=fmtp:124 apt=120
a=fmtp:121 max-fs=12288;max-fr=60
a=fmtp:125 apt=121
a=fmtp:127 apt=126
```

```
a=fmtp:98 apt=97
a=ice-pwd:efe34e413f35fb225352e73b89d2fa73
a=ice-ufrag:0b863d52
a=mid:1
a=msid:{89963797-b150-406b-8c17-d3a02e5ab84b} {ba24bbd7-24aa-42cd-b1b5-7
    ↪ def80a62239}
a=rtcp-fb:120 nack
a=rtcp-fb:120 nack pli
a=rtcp-fb:120 ccm fir
a=rtcp-fb:120 goog-remb
a=rtcp-fb:120 transport-cc
a=rtcp-fb:121 nack
a=rtcp-fb:121 nack pli
a=rtcp-fb:121 ccm fir
a=rtcp-fb:121 goog-remb
a=rtcp-fb:121 transport-cc
a=rtcp-fb:126 nack
a=rtcp-fb:126 nack pli
a=rtcp-fb:126 ccm fir
a=rtcp-fb:126 goog-remb
a=rtcp-fb:126 transport-cc
a=rtcp-fb:97 nack
a=rtcp-fb:97 nack pli
a=rtcp-fb:97 ccm fir
a=rtcp-fb:97 goog-remb
a=rtcp-fb:97 transport-cc
a=rtcp-mux
a=rtcp-rsize
a=rtpmap:120 VP8/90000
a=rtpmap:124 rtx/90000
a=rtpmap:121 VP9/90000
a=rtpmap:125 rtx/90000
a=rtpmap:126 H264/90000
a=rtpmap:127 rtx/90000
a=rtpmap:97 H264/90000
a=rtpmap:98 rtx/90000
a=setup:actpass
a=ssrc:335903003 cname:{a77c69ce-8bfc-4cf6-ac3c-0db7976f374a}
a=ssrc:1910270587 cname:{a77c69ce-8bfc-4cf6-ac3c-0db7976f374a}
a=ssrc-group:FID 335903003 1910270587
```

[RFC8866] SDP opisuje sesję jako kolekjcję pól, z których każde zawiera się w jednej linii. Na przykładzie opisu sesji z listingu 5.8, można wyróżnić następujące pola:

- `v=0`: wersja numeru protokołu - aktualnie 0 jest jedyną możliwą wersją
- `o=<username> <sess-id> <sess-version> <nettype> <addrtype> <unicast-address>`: kolekcja pól o inicjatorze sesji
  - `username: mozilla THIS_IS_SDPARTA-99.0`, co jest referencją do filmu 300
  - `sess-id: 107455341790422027`
  - `sess-version: 0`
  - `nettype: IN` - `IN` ma oznaczać internet, inne wartości mogą zostać użyte w przyszłości
  - `addrtype: IP4`
  - `unicast-address: 0.0.0.0`
- `s=-`: nazwa sesji. Z RFC: “ The "s="line MUST NOT be empty. If a session has no meaningful name, then "s= "or "s=-"(i.e., a single space or dash as the session name) is RECOMMENDED.”[RFC8866]

- t=<start-time> <stop-time>: czas rozpoczęcia i zakończenia sesji w czasie unixowym. Wartość wynosi 0, ponieważ sesja nie jest ograniczona czasowo.
- m=<media> <port> <proto> <fmt> ... - opis mediów:
  - m=audio 9 UDP/TLS/RTP/ SAVPF 109 9 0 8 101
  - m=video 9 UDP/TLS/RTP/ SAVPF 120 124 121 125 126 127 97 98

Resztę opisu dominują pola a=: atrybuty, które są głównym sposobem rozszerzania SDP. Mogą one być używane jako atrybuty sesji, atrybuty mediów, lub oba. Pozycje tego pola zaczynające się od rtpmap zawierają oferowane do użycia w połączeniu kodeki audio i wideo.

Rozdział 6

Implementacja

W niniejszym rozdziale zostanie przedstawiona koncepcja realizacji projektu wraz z opisem używanych technologii.

6.1. Opis projektu

Projekt wykorzystuje zarówno architekturę klient-serwer jak i architekturę P2P.

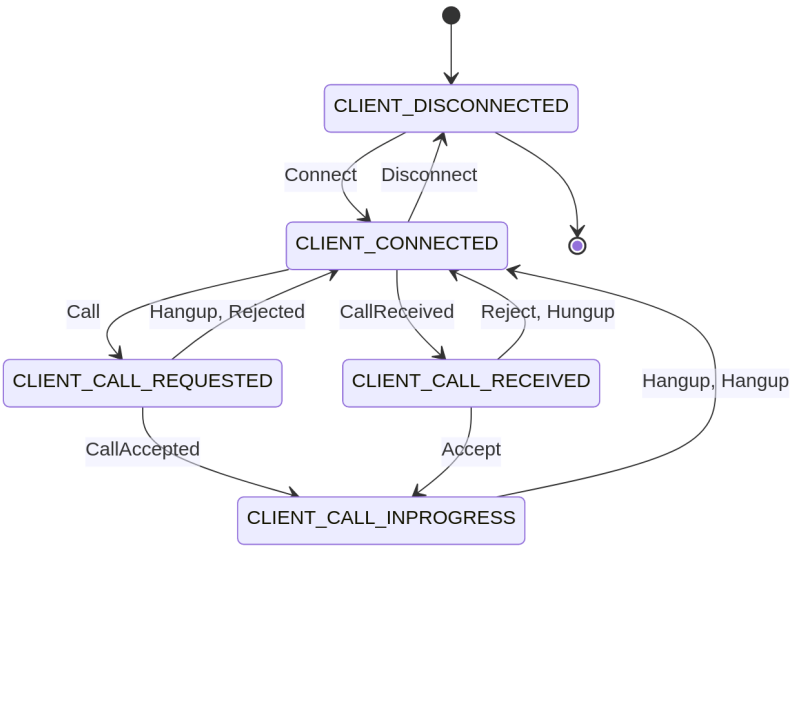
Na projekt składają się dwie części, serwer oraz aplikacja graficzna (klient).

Serwer pełni dwie funkcje:

- odkrywanie - tj. pozwala połączonym klientom ogłosić swoją dostępność oraz wysyła klientom listę innych połączonych klientów
- sygnalizacja - aby utworzyć połączenie peer-to-peer pomiędzy klientami, niezbędna jest między nimi wymiana informacji przez jakiś inny kanał. W tym celu serwer pośredniczy w wymianie wiadomości pomiędzy klientami.

Celem klienta jest połączenie z serwerem, a następnie jednoczesne nasłuchwanie na wiadomości od serwera oraz reagowanie na zdarzenia pochodzące od użytkownika.

Poszczególne połączenia nawiązywane przez klientów realizowane są w architekturze P2P.



Rys. 6.1: Diagram stanów klienta

Zarówno wiadomości od klienta (rozpoczęcie połączenia, rozłączenie się) jak i od serwera (pojawił się nowy użytkownik, otrzymano połączenie) mogą pojawić się w każdym momencie, więc model komunikacji request/response będzie niewystarczający. Potrzebny jest model komunikacji gdzie każda ze stron nasłuchuje na przychodzące zdarzenia od drugiej strony połączenia.

6.1.1. Serwer

Do komunikacji wykorzystywane są protokoły TCP i WebSockets. TCP zapewnia gwarancję poprawności odebranych danych i ich odpowiedniej kolejności, zamienia strumień pakietów na łańcuch bajtów przychodzący w kolejności. Protokół Websockets zapewnia framing, czyli zamienia łańcuch bajtów na strumień wiadomości które mogą mieć różną wielkość i zawierać dane tekstowe lub binarne. Protokół aplikacji wykorzystuje tylko dane tekstowe.

Protokół aplikacji

Rodzaje komunikatów możliwych do wysłania przez klient lub serwer są zebrane w odpowiedni typ wyliczeniowy a następnie w wiadomości tekstowej Websockets wysyłana jest ich serializacja w formacie JSON.

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "lowercase")]
pub enum Message {
    Connect(ConnectMessage),
    ConnectResponse(ConnectResponse),
    UserList(UserList),
    Webrtc(WebrtcMsg),
    Call(CallMessage),
    CallReceived(CallReceivedMessage),
```

```
    CallHangup,
    CallResponse(CallResponseMessage),
}
```

Implementacja serwera wykorzystuje dwie techniki komunikacji wieloprocesowej:

- synchronizacja dostępu do dzielonych danych
- przekazywanie wiadomości

6.2. Klient

Aplikacja okienkowa składa się z czterech konkurentnie wykonujących się zadań:

1. Zadanie GUI, rysuje okienko oraz emituje zdarzenia GUI
2. Zadanie połączenia z serwerem, nasłuchuje wiadomości wysłane przez serwer oraz emituje zdarzenia sieciowe
3. Zadanie połączenia, prowadzi proces sygnalizacji i negocjacji WebRTC, a następnie prowadzi połączenie: prezentuje strumienie audio-wideo użytkownikowi oraz wysyła strumienie wideo i audio drugiej stronie połączenia. Działa od rozpoczęcia połączenia do jego zakończenia.

Rdzeń aplikacji odpowiedzialny jest za odbieranie zdarzeń od zadań i reagowanie na nie wysyłając komendy do innych zadań. Np. odbieranie zdarzenia kliknięcia w przycisk rozpoczęcia połączenia obok użytkownika X powoduje wysłanie komendy "zadzwoń do użytkownika X"do zadania połączenia. Podobnie gdy zadanie połączenia odbierze od serwera wiadomość `UserList`, rdzeń odbiera to zdarzenie i wysyła komendę do zadania GUI by zaktualizowało listę dostępnych użytkowników.

Wyjątkiem jest zadanie GStreamer które komunikuje się bezpośrednio z zadaniem połączenia. Powody takiej decyzji są dwa:

- Zadanie GStreamer wysyła i odbiera dużą ilość danych które nie mają żadnego efektu w GUI, więc bezpośrednie połączenie sprawia że rdzeń nie musi być ich świadom
- Dopóki trwa rozmowa, użytkownik nie może rozpoczynać ani odbierać żadnych nowych rozmów

6.2.1. Zadanie GUI

GUI emituje następujące zdarzenia:

```
#[derive(Debug)]
pub enum GuiEvent {
    CallStart(u32),
    CallAccepted(VideoPreference),
    CallRejected,
    NameEntered(String),
}
```

6.2.2. Zadanie połączenia

Zadanie połączenia emituje następujące zdarzenia:

```
#[derive(Debug)]
pub enum NetworkEvent {
    UserlistReceived(Vec<(u32, String)>),
```

```
    CallReceived(String),
}
```

6.2.3. Zadanie GStreamer

Zadanie GStreamer jest tworzone gdy rozpoczynane jest nowe połączenie, i jest niszczone gdy połączenie jest zamykane.

Pipeline

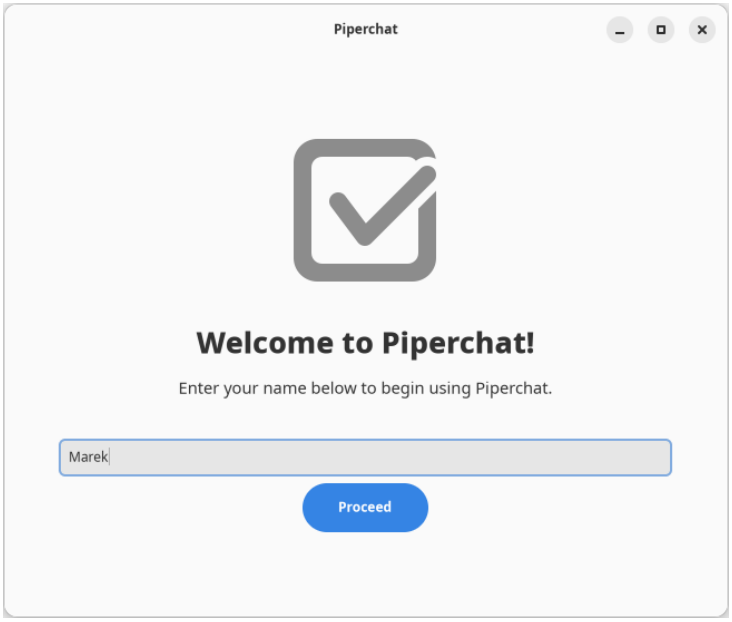
W konstruktorze obiektu tworzony jest pipeline z reprezentacji tekstowej:

```
// Create the GStreamer pipeline
let pipeline = gst::parse_launch(
        "v4l2src ! videoconvert ! vp8enc deadline=1 ! rtpvp8pay pt=96 ! webrtcbin. \
        autoaudiosrc ! opusenc ! rtpopuspay pt=97 ! webrtcbin. \
        webrtcbin name=webrtcbin",
)?;
```

Listing 1: Reprezentacja tekstowa pipeline'u

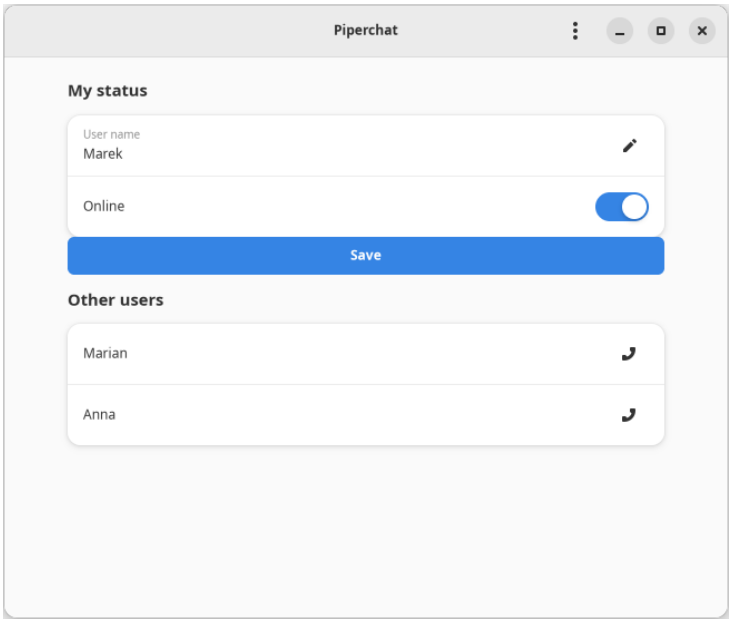
6.2.4. Interfejs użytkownika

Ekran lądowania



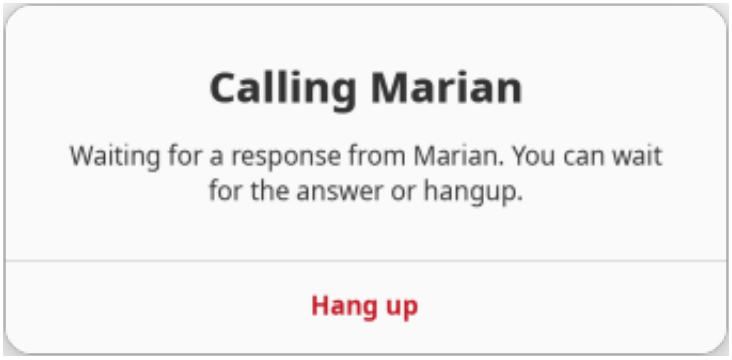
Rys. 6.2: Ekran lądowania

Główny ekran aplikacji



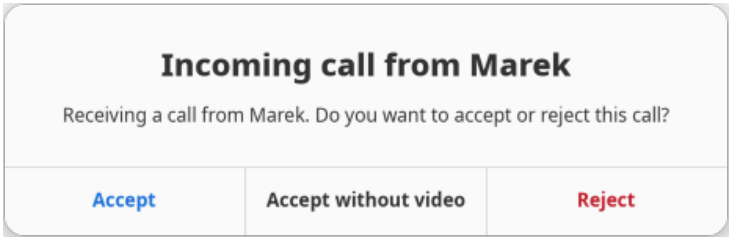
Rys. 6.3: Główny ekran aplikacji

Dialog wysłania połączenia



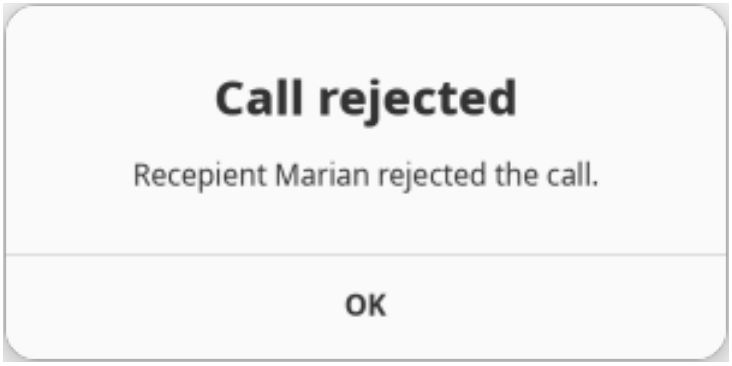
Rys. 6.4: Dialog wysłania połączenia

Dialog otrzymania połączenia



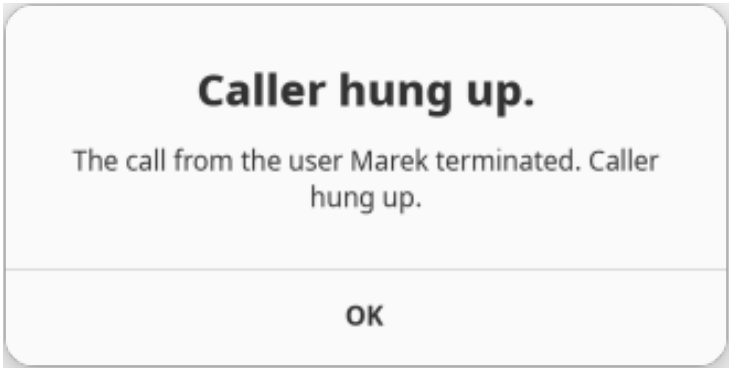
Rys. 6.5: Dialog otrzymania połączenia

Dialog odrzucenia połączenia przez odbiorcę



Rys. 6.6: Dialog odrzucenia połączenia przez odbiorcę

Dialog porzucenia połączenia przez nadawcę



Rys. 6.7: Dialog porzucenia połączenia przez nadawcę

Rozdział 7

Podsumowanie

Podsumowanie jest miejscem, w którym należy zamieścić syntetyczny opis tego, o czym jest dokument. W szczególności w pracach dyplomowych w podsumowaniu powinno znaleźć się jawnie podane stwierdzenie dotyczące stopnia realizacji celu. Czyli powinny pojawić się w niej akapity ze zdaniami typu: „Podczas realizacji pracy udało się zrealizować wszystkie postawione cele". Ponadto powinna pojawić się dyskusja na temat napotkanych przeszkód i sposobów ich pokonania, perspektyw dalszego rozwoju, możliwych zastosowań wyników pracy itp.

7.1. Realizacja celu pracy

Zadanie utworzenia komunikatora prezentującego reguły prowadzenia połączeń peer-to-peer zostały zrealizowane.

7.2. Wnioski

Wykonany komunikator nie jest jednak gotowy do wykorzystania w środowisku produkcyjnym ponieważ operuje on na zasadzie globalnej widoczności wszystkich połączonych użytkowników, którzy identyfikowani są tylko po nazwie, którą każdy z użytkowników może wybrać dowolnie podczas dołączania, o ile nie jest ona już zajęta.

Mimo tego, komunikator obrazuje jak łatwe może być tworzenie multimedialnych aplikacji komunikujących się w architekturze peer-to-peer. Gwarancje bezpieczeństwa języka Rust sprawiają że deweloper może zapomnieć o segfaultach i memory corruption, a ekspresywny system typów sprawia że powstały kod wygląda jak kod napisany w języku wysokopoziomowym takim jak np. Java, jednocześnie będąc szybkim jak kod w języku niskopoziomowym (np. C) dzięki abstrakcjom o zerowych kosztach.

Elementami który dodały najwięcej złożoności i potencjału na wprowadzenie błędów były GStreamer oraz GTK. Są one napisane w C, i ich bindingi w języku Rust są tylko obudową na ogrom kodu C, który replikuje i zastępuje wiele funkcjonalności wykonywanych lepiej przez Rusta.

7.3. Możliwe usprawnienia

- Wyświetlanie statusu użytkowników na liście wszystkich użytkowników
- Kontrolki do włączenia/wyłączenia kamerki/mikrofonu w oknie połączenia
- Wykorzystanie pełnoprawnego frameworka architektury aktorów, np. Actix.

Bibliografia

[1] I. Grigorik, “High performance browser networking”, en, w Sebastopol, CA: O’Reilly Media, wrz. 2013, rozd. WebRTC.

[2] “GStreamer Documentation”. (), adr.: https://gstreamer.freedesktop.org/documentation. (accessed: 20.01.2023).

[RFC8866] A. Begen, P. Kyzivat, C. Perkins i M. Handley, “SDP: Session Description Protocol”, RFC Editor, RFC 8866, sty. 2021. adr.: https://www.rfc-editor.org/info/rfc8866.

Rozdział 8

Instrukcja wdrożeniowa

Jeśli praca skończyła się wykonaniem jakiegoś oprogramowania, to w dodatku powinna pojawić się instrukcja wdrożeniowa (o tym jak skompilować/zainstalować to oprogramowanie). Przydałoby się również krótkie how to (jak uruchomić system i coś w nim zrobić – zademonstrowane na jakimś najprostszym przypadku użycia). Można z tego zrobić osobny dodatek,

Rozdział 9

Opis załączonej płyty CD/DVD

Tutaj jest miejsce na zamieszczenie opisu zawartości załączonej płyty. Należy wymienić, co zawiera.