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

W ostatnich latach członkowie konsorcjum AOM wprowadzają na rynek nowy standard kompresji wideo - AV1. W niektórych zastosowaniach kodek AV1 jest w stanie osiągnąć taką samą jakość wizualną jak używany od 2003 standard AVC, zajmując przy tym zaledwie połowę miejsca. Nowy standard ma zatem duże znaczenie dla członków konsorcjum, których działalność opiera się w znacznej części na strumieniowaniu dużej ilości wideo do wielu klientów, takich jak Google, Netflix, lub Amazon, którzy grają pierwsze skrzypce w jego adopcji. Niestety, ponieważ implementacje koderów AV1 pozostają znacznie wolniejsze od już przyjętych kodeków, aplikacje strumieniujące wideo w czasie rzeczywistym, takie jak komunikatory internetowe lub platformy wideokonferencyjne, nie miały jeszcze okazji skorzystać z nowego kodeka aby usprawnić jakość transmitowanego wideo.

Tematem pracy jest zbadanie praktyczności wykorzystania nowoczesnego kodeka wideo AV1 w komunikatorze wideo. Wykonano w tym celu prosty komunikator internetowy składający się z serwera oraz aplikacji graficznej na systemy Linux wykonanej w języku programowania Rust z użyciem frameworków GTK oraz GStreamer. Niestety dostępne w GStreamer implementacje kodeków nie były w stanie osiągnąć prędkości kodowania pozwalającej na strumieniowanie wideo w czasie rzeczywistym i aplikacja wykorzystuje zamiast tego kodek VP8. W miarę rozwoju enkoderów programowych a także coraz większej dostępności enkoderów sprzętowych w nowoczesnych układach graficznych, użycie AV1 w komunikatorach wideo może jednak stać się praktyczne w niedalekiej przyszłości.

Słowa kluczowe: AV1

Abstract

In recent years the members of the AOM consortium have been rolling out the new video compression standard - AV1. The new standard can maintain the same visual quality while taking up as little as half of the space compared to AVC, an older standard used from 2003. As such, fast adoption of the new standard was pursued by the members of the consortium, whose business largely consists of providing video streaming services for many customers, such as Google, Netflix, or Amazon. Unfortunately because recent AV1 production codecs sometimes still struggle with encoding speed when compared to the older, more established codecs, real-time applications like video-chat or videoconferencing applications were not yet able to take advantage of the new standard to provide a better quality video streams for its users.

This paper aims to examine the feasibility of the recently developed AV1 standard when used in a video-chat application. For this purpose, such application was developed using the Rust programming language, consisting of a server and a GUI application for Linux systems utilizing GTK and GStreamer frameworks. Unfortunately available software codecs were not able to achieve a real-time encoding speed and the application was made to use the VP8 codec instead. As software codecs improve and hardware codecs continue to appear in more products, the usage of AV1 in real-time applications may nonetheless become feasible in a near future.

Keywords: AV1

Spis treści

Spis rysunków

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*)
IO (ang. *Input/Output*)
MPEG (ang. *Moving Picture Experts Group*)
P2P (ang. *Peer to peer*)
SDP (ang. *Session Description Protocol*)
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

W rozdziale 2 wykonana zostanie analiza wymagań, omówione zostaną wymagania funkcjonalne oraz niefunkcjonalne tworzonego oprogramowania.

W rozdziale 3 omówione zostaną niezbędne do opanowania zagadnienia teoretyczne - poruszone zostaną koncepty i metody związane ze strumieniowaniem wideo oraz różne problemy związane z transmisją wideo przez sieć internetową. Następnie przedstawione zostaną technologie wykorzystane w realizacji zadania: WebRTC, GStreamer, Rust, a także zostaną omówione wykorzystane techniki programowania asynchronicznego.

W rozdziale 4 przedstawiona zostanie koncepcja realizacji projektu, tj. plan projektu prezentujący poszczególne części tworzonego oprogramowania oraz zachowanie tych części w relacji ze sobą.

W rozdziale 5 zaprezentowane zostaną dwie wykonane aplikacje. Pierwszą z nich jest przykładowa aplikacja webowa wykorzystująca API WebRTC w języku Javascript wraz z wykorzystywanymi przez nią technologiami, oraz omówionymi fragmentami kodu źródłowego realizujące kluczowe procesy nawiązywania połączenia poruszone w rozdziale trzecim. Wykonana zostanie uproszczona analiza ruchu sieciowego pomiędzy hostami i uzupełniony zostanie proces nawiązywania połączenia WebRTC poruszony we wcześniejszym rozdziale, zobrazowany konkretnym przykładem.

Drugą z zaprezentowanych aplikacji jest finalna implementacja aplikacji okienkowej wideokomunikatora Piperchat na systemy Linux. Przedstawiona zostanie struktura projektu i detale implementacji serwera oraz klienta, diagramy przedstawiające ich strukturę oraz fragmenty kodu.

Rozdział 6 podsumowuje pracę, omawia jakie cele pracy zostały zrealizowane, przedstawia wnioski oraz możliwe usprawnienia projektu.

Rozdział 2

Analiza wymagań

2.1. Wymagania projektowe

Końcowym celem projektu jest poznanie procesów kluczowych w internetowych transmisjach audio-wideo oraz wykonanie aplikacji okienkowej na systemy Linux pełniącej rolę komunikatora internetowego. Komunikator powinien pozwalać 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 bezpośrednio, 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.
8. (opcjonalne) Użytkownik może zmienić swoją nazwę pod którą jest widoczny bez rozłączania się z serwerem

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. Połączenia pomiędzy użytkownikami powinny odbywać się w trybie peer-to-peer celem minimalizacji opóźnień
5. Z powodu powyższego wymagania, połączenia są ograniczone do dwóch użytkowników, tj. nie ma możliwości tworzenia konferencji z wieloma użytkownikami.
6. (opcjonalne) Aplikacja wykorzystuje kodek AV1 do kompresji wideo
7. (opcjonalne) Aplikacja powinna wdzięcznie obsługiwać brak kamery wideo lub mikrofonu przez użytkownika

Rozdział 3

Zagadnienia teoretyczne

Niniejszy rozdział omawia niektóre techniki kompresji wideo oraz prezentuje pewne problemy związane z nawiązywaniem połączeń P2P, które muszą zostać rozwiązane by możliwe było zrealizowanie przedstawionej w poprzednim rozdziale aplikacji. Przedstawione zostaną także narzędzia oraz technologie które zostaną wykorzystane do realizowania procesów kodowania wideo oraz nawiązywania połączeń P2P.

3.1. Kodowanie wideo

Celem tej sekcji jest zaprezentowanie podstawowych wykorzystywanych przez standardy AVC oraz AV1 technik kompresji, a następnie porównanie tych standardów. Opis procesów kompresji wraz z przykładami został zaczerpnięty z repozytorium digital-video-introduction[1].

Obraz jest najczęściej reprezentowany jako trójwymiarowa macierz, gdzie pierwsze dwa wymiary stanowią szerokość oraz wysokość obrazu, a trzeci wymiar stanowią wartości barw podstawowych: czerwonego, zielonego, oraz niebieskiego. Każdy element tej macierzy jest nazywany pikselem. Każdy piksel reprezentuje nasilenie danego koloru. Każda wartość koloru wymaga danej ilości bitów do zakodowania, np. jeśli istnieją do dyspozycji wartości 0-255 by wyrazić intensywność danego koloru, to informację tą można zawrzeć w 8 bitach. Mówi się wtedy że dany obraz ma 8 bitów **głębii bitowej**. 8 bitów dla każdego z trzech kanałów daje nam także łącznie 24 bitów **głębii koloru**.

Wideo jest w takim razie sekwencją wielu takich obrazów w danym czasie, którego ważną charakterystyką jest liczba wyświetlanych klatek na sekundę, **FPS** (ang. frames per second). Aby wyświetlić wideo, potrzebna jest zatem pewna liczba bitów na sekundę, lub też zwyczajnie **szybkość transmisji** (ang. bitrate).

$$bitrate = wysokość * szerokość * głębiakoloru * FPS$$

Na przykład wideo w 30FPS, 24 bitami na piksel, w rozdzielczości 480x240 będzie potrzebować ok. 82.94 Mb (megabitów) na sekundę, jeżeli nie zostanie wykorzystana żadna kompresja. Algorytmy kompresji bezstratnej pomagają częściowo, są jednak bardzo ograniczone przez fakt że nie są w stanie odrzucać informacji. Standardy kodowania wideo wykorzystują zatem w zdecydowanej większości techniki kompresji stratnej, sprytnie wykorzystujące percepcję człowieka tak by pozbywać się informacji które i tak są przez człowieka niezauważalne, znacznie zwiększając w ten sposób stopień kompresji.

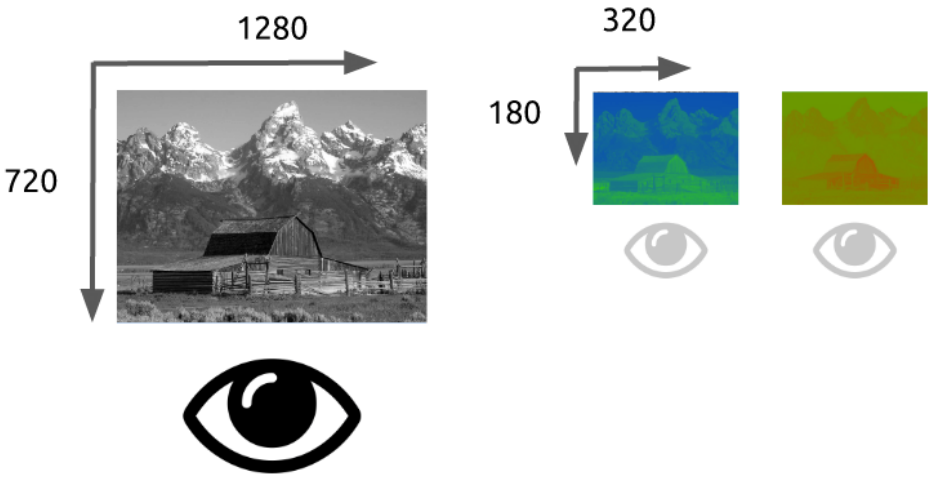
Nowoczesne standardy kodowania wideo wykorzystują następujące techniki:

1. podpróbkowanie chrominancji
2. kompensacja ruchu

3. predykcja wewnątrzklatkowa
4. dyskretna transformacja cosinusowa i kwantyzacja
5. kodowanie entropijne

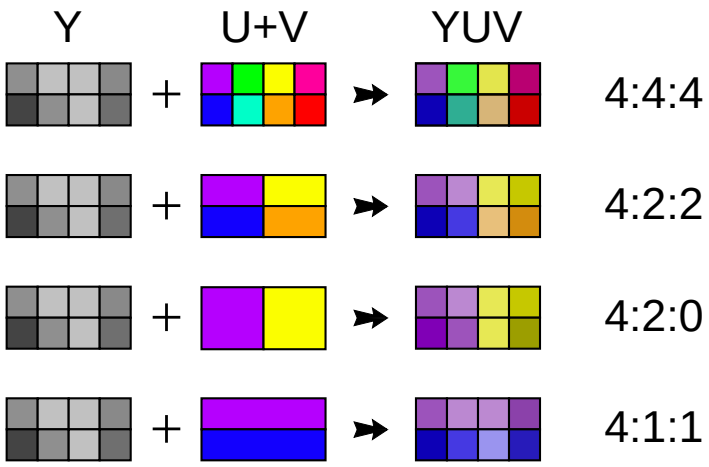
Podpróbkowanie chrominancji

Oczy człowieka są bardziej czułe na zmiany w jasności niż zmiany w kolorze. Jeżeli obraz zostanie przedstawiony w przestrzeni kolorów składającej się ze składowych jasności i koloru (lub luminancji i chrominancji), możemy wykorzystać ten fakt próbkując informacje o kolorze z mniejszą rozdzielczością niż informacje o jasności. Ta technika zwie się podpróbkowaniem chrominancji (ang. chroma subsampling). Rysunek 3.1 prezentuje różnice w rozdzielczości kanałów chroma i luma w najpopularniejszym trybie 4:2:0, a rysunek 3.2 prezentuje formaty, tzn. tryby które wykorzystują różną liczbę próbek luma pokrywającą w różny sposób powierzchnię.



Rys. 3.1: Prezentacja działania chroma subsampling

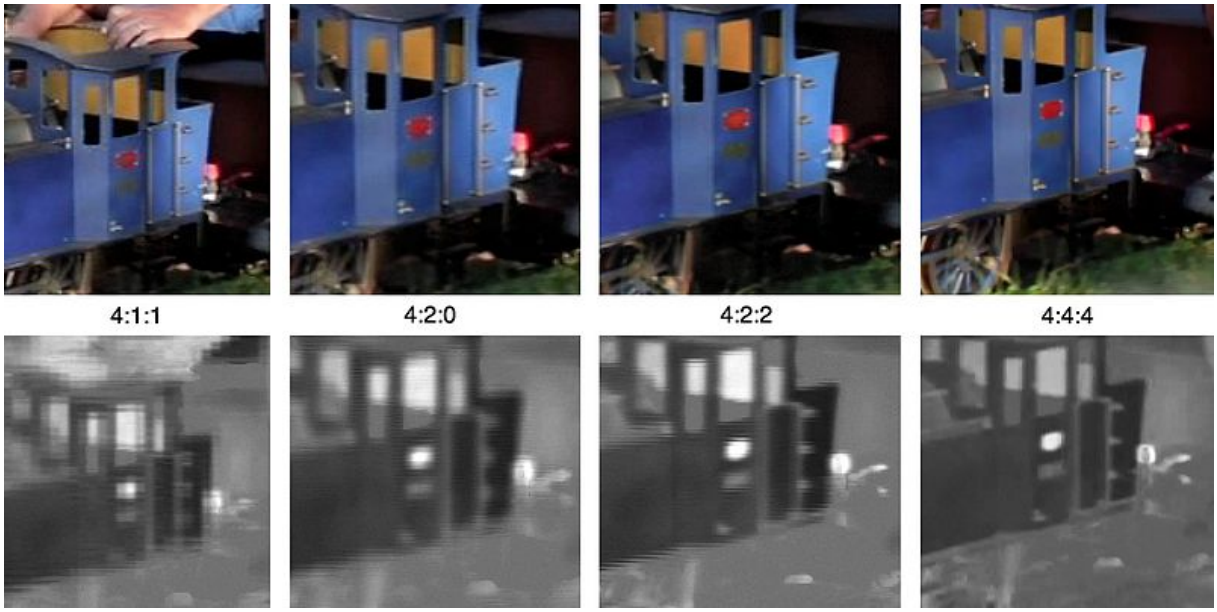
Źródło: repozytorium digital-video-introduction [1]



Rys. 3.2: Zasada działania różnych formatów chroma subsampling

Źródło: repozytorium digital-video-introduction [1]

Jak widać na obrazku 3.3, pomimo bardzo niskiego próbkowania informacji o kolorze, w górnym rzędzie nie widać znaczących różnic pomiędzy pierwszym i ostanim obrazkiem.

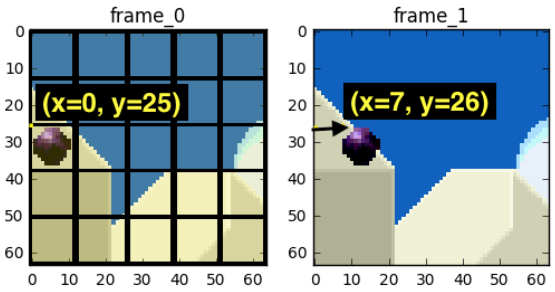


Rys. 3.3: Porównanie kanału chroma oraz rezultatu różnych formatów chroma subsampling

Źródło: repozytorium digital-video-introduction [1]

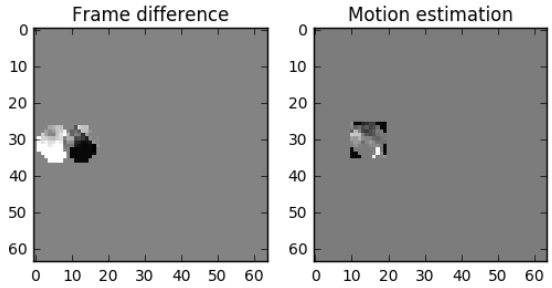
Kompensacja ruchu

Kompensacja ruchu jest jedną z metod kompresji inter-frame, tzn. zmian pomiędzy klatkami. Ponieważ często zawartość klatki zmienia się tylko częściowo, zostawiając inne elementy bez zmiany, możliwe jest zapisanie tylko różnicy pomiędzy sąsiadującymi klatkami. Kompensacja ruchu usprawnia ten proces: dzięki wprowadzeniu możliwości przenoszenia całych bloków, możemy zakodować zmiany w klatce z użyciem mniejszej liczby bitów i zmniejszyć różnicę pomiędzy klatkami którą trzeba będzie zaaplikować. Proces przezentuje kompensacja ruchu obiektu z rysunku 3.4, natomiast na rysunku 3.5 widoczna jest pozostała różnica po procesie kompensacji.



Rys. 3.4: Przykład sąsiadujących klatek na których występuje ruch

Źródło: repozytorium digital-video-introduction [1]

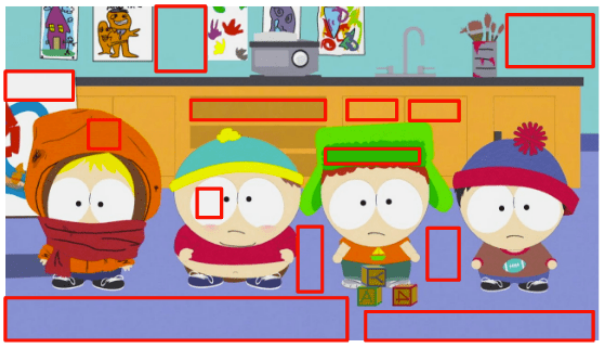


Rys. 3.5: Porównanie różnicy pomiędzy klatkami bez oraz z uwzględnieniem kompensacji ruchu

Źródło: repozytorium digital-video-introduction [1]

Predykcja wewnątrzklatkowa

Predykcja wewnątrzklatkowa (ang. intra-frame prediction) wykorzystuje redundancję w obrębie jednej klatki do opisania jej z użyciem mniejszej liczby bitów. Widoczne na rysunku 3.6 oznaczone obszary zawierają obszary skorelowane, których wartości mogą być przewidziane na podstawie wartości sąsiadujących pikseli.

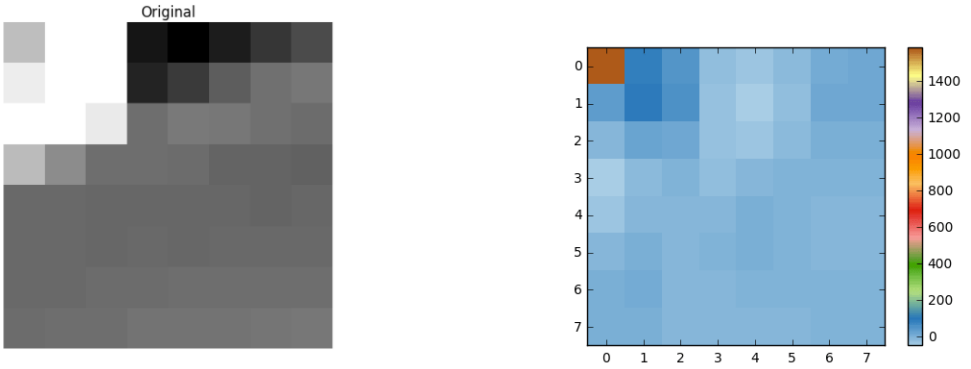


Rys. 3.6: Klatka z serialu South Park z zaznaczonymi skorelowanymi obszarami. Obszary te można skompresować predykcją intra-frame

Źródło: repozytorium digital-video-introduction [1]

Dyskretna transformacja cosinusowa i kwantyzacja

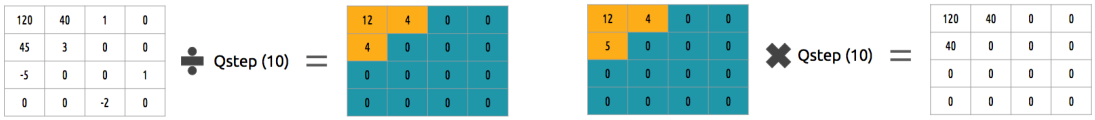
Uzyskany po zaaplikowaniu predykcji blok różnicy (ang. residual block) można następnie przetransformować za pomocą DCT i następnie za pomocą kwantyzacji pozbyć się wyższych współczynników przetransformowanego bloku. Proces przebiega podobnie jak DCT w kodowaniu JPG. Rysunki 3.7 oraz 3.8 przestawiają kolejno proces DCT oraz kwantyzacji (z różnicą że podczas kwantyzacji nie dzieli się przez jedną wartość, tylko przez specjalne macierze kwantyzacji).



(a) Blok różnicy po predykcji (b) Blok współczynników po DCT

Rys. 3.7: Proces DCT

Źródło: repozytorium digital-video-introduction [1]



Rys. 3.8: Proces kwantyzacji

Źródło: repozytorium digital-video-introduction [1]

Kodowanie entropijne

Po kwantyzacji, ostatnim krokiem jest bezstratne skompresowanie powstałej sekwencji współczynników. Można do tego celu wykorzystać różne sposoby kompresji bezstratnej, np. kodowanie Huffmana lub kodowanie arytmetyczne.

3.2. Internetowe strumienie wideo

Strumienie wideo w zależności od źródła mogą mieć różne charakterystyki i ograniczenia. Możemy wyróżnić 3 rodzaje strumieni wideo:

- lokalne pliki wideo - pliki wideo na dysku w kontenerze mp4, mkv, lub innym
- strumieniowanie plików wideo (np. Youtube albo Netflix) - przygotowane segmenty pliku wideo w różnych rozdzielczościach wysyłane są po kolei, rozdzielczość dobierana jest wg. dostępnego pasma pomiędzy serwerem a klientem
- strumieniowanie w czasie rzeczywistym - priorytetem jest opóźnienie, przechwytywane klatki kodowane są na bieżąco i są wysyłane najszybciej jak to możliwe; przez to niektóre mechanizmy kompresji są niedostępne (np. B-klatki, które wykorzystują dane z następnej klatki aby zmniejszyć wielkość klatki). O tych strumieniach jest mowa w pracy.

Zidealizowany obraz rozmowy wideo w internecie wygląda następująco:

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 dźwięk, 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

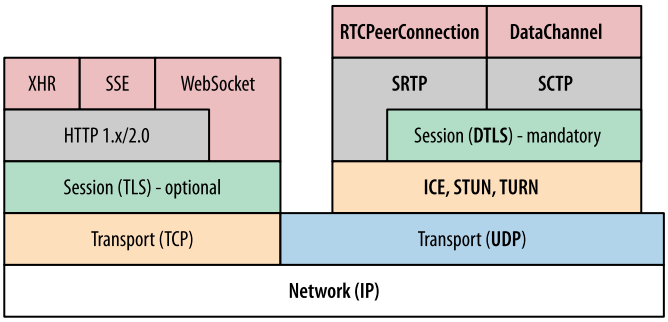
3.3. Wykorzystane technologie

W tej sekcji zostaną opisane technologie użyte w dalszej części pracy.

3.3.1. WebRTC

Niniejszy rozdział przytacza fragmenty książki *High Performance Browser Networking* [2] celem opisu projektu WebRTC.

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 3.9.



Rys. 3.9: Stos protokołów w WebRTC

Źródło: High Performance Browser Networking[2]

Głównym targetem WebRTC są przeglądarki, jednak istnieją także implementacje dla innych typów aplikacji. W podrozdziale 3.3.2 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?

Sygnalizacja nawiązania połączenia

Przed nawiązaniem komunikacji P2P i negocjacji parametrów strumieni mediów, należy najpierw powiadomić drugą stronę połączenia o intencji nawiązania połączenia oraz ustalić czy peer jest osiągalny. Ponieważ peer nie nasłuchuje jeszcze pakietów, potrzebny jest wspólny kanał sygnalizacyjny przez który możliwa będzie sygnalizacja połączenia. Sytuację przedstawiono na rysunku 3.10.

Rys. 3.10: Strony sygnalizacji

Źródło: High Performance Browser Networking[2]

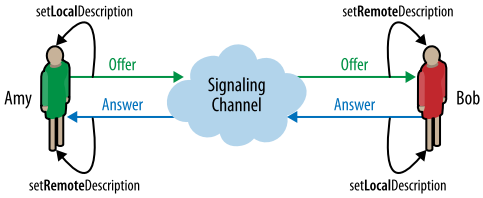
WebRTC nie wymusza żadnego protokołu transportowego do celów sygnalizacji, zamiast tego pozostawiając wybór aplikacji. Pozwala to na interoperacyjność z istniejącymi już protokołami i infrastrukturą.

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

Negocjacja parametrów połączenia

WebRTC używa protokołu SDP (ang. Session Description Protocol) do opisu parametrów połączenia P2P. Samo SDP nie zawiera żadnych mediów, zamiast tego opisuje tylko profil sesji, tj. zawiera listę różnych właściwości połączenia, takich jak rodzaje strumieni, użyte kodeki i ich ustawienia, prędkość transmisji, etc.

Aby ustanowić połączenie P2P, oba peery muszą wymienić swoje opisy SDP w procesie widocznym na rysunku 3.11.



Rys. 3.11: Proces negocjacji

Źródło: High Performance Browser Networking[2]

1. Strona inicjująca (Amy) rejestruje swoje lokalne strumienie mediów, tworzy z nich ofertę połączenia i ustawia ją jako lokalny opis sesji.
2. Amy następnie wysyła wygenerowaną ofertę do drugiej strony połączenia (Bob).
3. Gdy Bob otrzyma ofertę, ustawia on otrzymany opis sesji jako opis zdalny po swojej stronie, następnie rejestruje swoje strumienie, generuje z nich odpowiedź i ustawia ją jako lokalny opis sesji.
4. Bob następnie wysyła swoją odpowiedź do Amy.
5. Gdy Amy otrzyma odpowiedź od Boba, ustawia ona tą odpowiedź jako zdalny opis sesji.

Po zakończeniu procesu wymiany opisów sesji przez kanał sygnalizacyjny, obie strony połączenia wynegocjowały jego parametry i ustawienia. Ostatnim krokiem jest nawiązanie połączenia P2P.

Interactive Connectivity Establishment (ICE)

By możliwe było nawiązanie połączenia P2P, oba peery powinny muszą być w stanie trasować pakiety do siebie nawzajem. Niestety istnieje wiele przeszkód mogących to uniemożliwić, np. NAT lub firewalle. Jeżeli oba peery znajdują się w tej samej sieci lokalnej, każdy peer może po prostu wysłać drugiej stronie swój adres IP w sieci lokalnej. Co jednak w sytuacji gdy peery znajdują się w różnych sieciach? Peery musiałyby wtedy znaleźć swój publiczny IP oraz w jakiś sposób otworzyć NAT na ruch od drugiej strony połączenia.

Na szczęście WebRTC również pomaga w tym aspekcie nawiązywania połączenia. Robi to za pomocą mechanizmu ICE (Interactive Connectivity Establishment).

- Każda strona połączenia zawiera agenta ICE
- agent ICE jest odpowiedzialny za znajdywanie adresów IP oraz numerów portów (kandydatów)
- agent ICE jest odpowiedzialny za sprawdzanie połączeń pomiędzy stronami
- agent ICE jest odpowiedzialny za podtrzymywanie połączenia

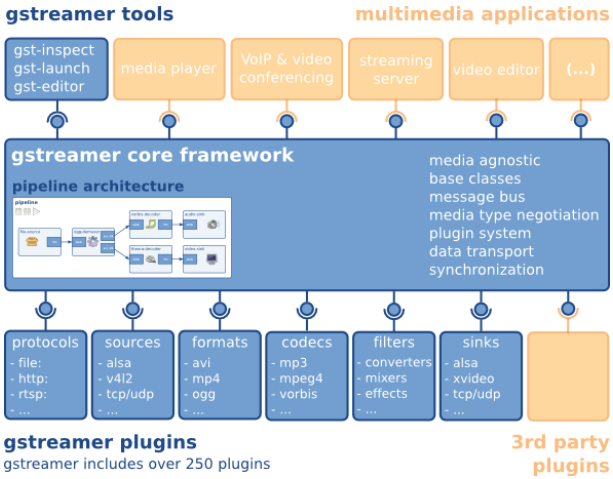
Gdy opis sesji zostanie ustawiony, lokalny agent ICE rozpoczyna proces odkrywania wszystkich możliwych adresów pod którym możliwe jest połączenie z lokalnym peerem

1. agent ICE pyta system operacyjny o lokalny adres IP
2. Jeśli istnieje, agent ICE pyta zdalny serwer STUN o publiczny adres IP i numer portu lokalnego peera.
3. Jeśli istnieje, agent ICE wykorzystuje serwer TURN jako pośrednik do ruchu pomiędzy peerami jeżeli niemożliwe jest ustanowienie połączenia P2P z powodu np. zbyt restrykcyjnego NAT.

Powyższe procesy są wykonywane automatycznie, w tle, programista musi tylko w odpowiedni sposób obsłużyć zdarzenia generowania i otrzymywania kandydatów ICE.

3.3.2. GStreamer

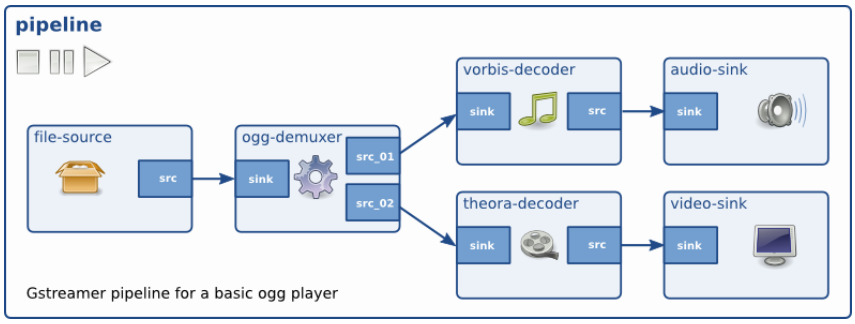
GStreamer jest frameworkiem do tworzenia strumieniujących aplikacji multimedialnych [3]. Aplikacje są wyrażane jako "rurociąg"składający się z bloków. Rdzeń GStreamer definiuje architekturę aplikacji oraz API dla elementów, a same elementy są zawarte w różnych pluginach. Jego modularna architektura sprawia że nie jest ograniczony do aplikacji audio-wideo, lecz może być wykorzystany do każdej aplikacji, która strumieniowo przetwarza dowolne rodzaje danych.



Rys. 3.12: Architektura projektu GStreamer

Źródło: Dokumentacja GStreamer[3]

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. 3.13: Przykładowy rurociąg aplikacji GStreamer

Źródło: Dokumentacja GStreamer[3]

Na rysunku 3.13 przedstawiono rurociąg przykładowego odtwarzacza plików OGG. Dane przechodzą przez rurociąg od lewej do prawej strony, będąc transformowane w każdym elemencie.

GStreamer WebRTC

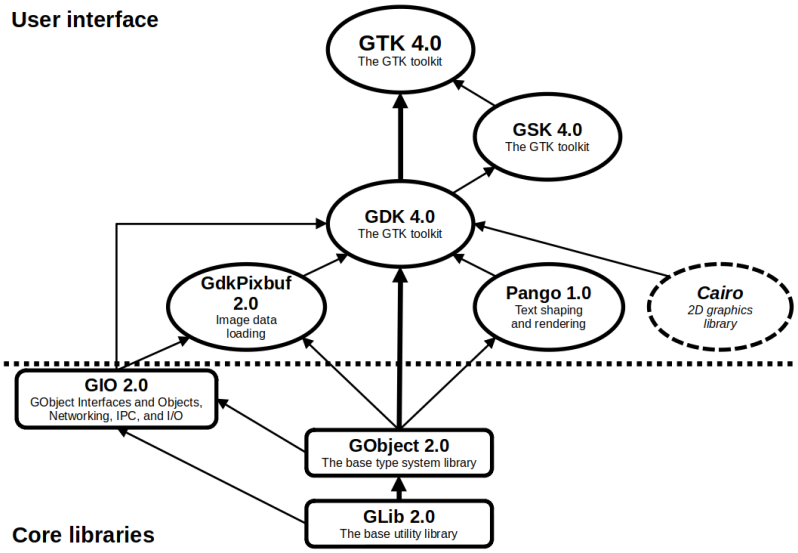
GStreamer posiada swoją własną implementację WebRTC, wykonaną przez firmę Centricular. Jej głównym elementem jest element webrtcbin. Ten element zajmuje się procesami negocjacji parametrów połączenia oraz zbierania i udostępniania kandydatów ICE omówionymi w poprzednich sekcjach. Zdarzenia takie jak 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.

3.3.3. 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 3.14.



Rys. 3.14: Architektura zestawu narzędzi GTK

Źródło: Wikipedia [4]

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`.

3.3.4. 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.

3.3.5. 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 thread pool do uruchamiania futures, dzięki czemu mogą być one wykonywane nie tylko współbieżnie 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 współbieżnego 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 4

Koncepcja realizacji

W niniejszym rozdziale zostanie przedstawiona koncepcja realizacji projektu.

4.1. Plan projektu

Projekt składa się z serwera oraz klienta (aplikacji graficznej komunikatora). Klienci łącząc się do serwera będą podawać swoją nazwę i będą "wpuszczani"jeżeli do serwera nie jest już podłączony klient z taką samą nazwą.

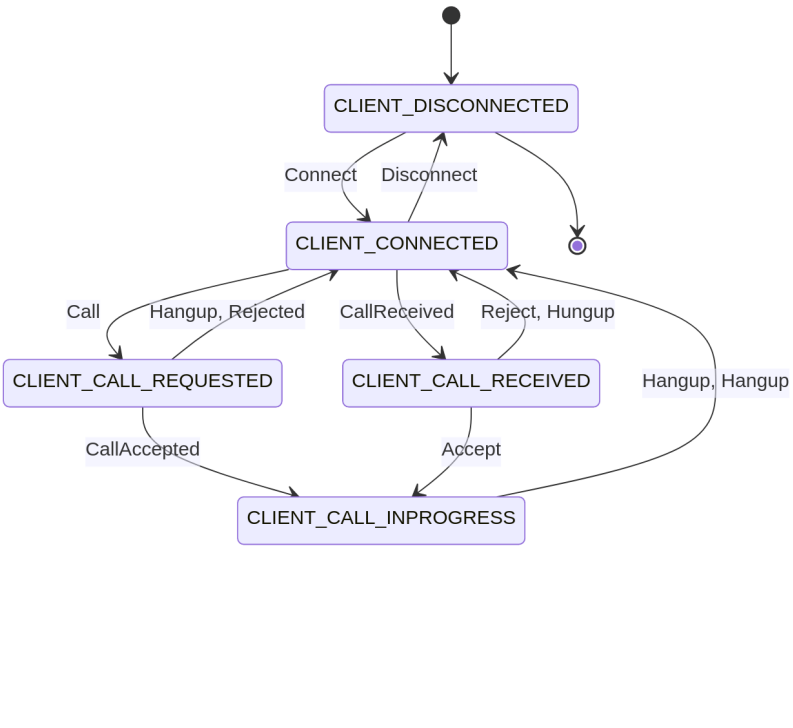
Serwer pełni dwie funkcje:

- odkrywanie - pozwala połączonym klientom ogłosić swoją dostępność w celu otrzymywania połączeń. W tym celu serwer wysyła wszystkim połączonym klientom pełną listę innych połączonych klientów za każdym razem gdy ulegnie ona zmianie (czyli np. jeżeli połączy się nowy klient lub dowolny z połączonych klientów rozłączy się).
- 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 nasłuchiwanie i reagowania na wiadomości od serwera, a także reagowanie na zdarzenia pochodzące od użytkownika. Gdy ustanawiane jest połączenie, klient powinien negocjować parametry połączenia z drugim klientem za pomocą zapewnionego przez serwer kanału, a następnie nawiązać bezpośrednie połączenie P2P z drugim klientem.

Interfejs graficzny klienta zostanie wykonany we frameworku GTK [5].

Na diagramie 4.1 przedstawiono wszystkie możliwe stany w których może znajdować się klient oraz wiadomości wysłane lub otrzymane które powodują zmianę danego stanu na inny.



Rys. 4.1: Diagram stanów klienta

Stany

- `CLIENT_DISCONNECTED` - klient nie jest połączony z serwerem. Klient znajduje się w tym stanie zaraz po wykonaniu WebSocket handshake, ale przed wysłaniem wiadomości `CONNECT`. W tym wypadku klient musi połączyć się wysyłając wiadomość `CONNECT` i nie może wykonywać żadnych innych akcji.
- `CLIENT_CONNECTED` - klient jest połączony, może rozpocząć nowe połączenie lub otrzymywać połączenia.
- `CLIENT_CALL_REQUESTED` - klient zadzwonił do innego klienta, może zakończyć rozmowę wysyłając `HANGUP` wracając do stanu `CLIENT_CONNECTED` lub poczekać aż drugi klient odpowie, i przejść do stanu `CLIENT_CALL_INPROGRESS`.
- `CLIENT_CALL_RECEIVED` - klient otrzymał zaproszenie do rozmowy z innym klientem. Może zaakceptować wysyłając `ACCEPT`, lub odmówić wysyłając `REJECT`. Nie może rozpoczynać połączeń ani otrzymywać innych połączeń dopóki nie odpowie na przychodzące połączenie.
- `CLIENT_CALL_INPROGRESS` - klienci są w połączeniu, każdy z nich może zakończyć rozmowę wysyłając wiadomość `HANGUP`, przywracając obu klientów do stanu `CLIENT_CONNECTED`.

Wiadomości

- `CONNECT name` - łączy się z serwerem oraz ogłasza dostępność innym klientom pod nazwą `name`. Serwer odsyła `CONNECT_OK` lub `CONNECT_ERR` z powodem dla którego nie można się połączyć.
- `CALL name` - wysyła zaproszenie do rozmowy klientowi pod nazwą `name`.
- `ACCEPT` - akceptuje połączenie przychodzące.
- `REJECT` - odrzuca połączenie przychodzące.
- `HANGUP` - kończy aktualne połączenie.

- `USERLIST userlist` - w stanie `CLIENT_CONNECTED` klient od razu po wejściu w ten stan, a także w dowolnym czasie może otrzymać wiadomość `USERLIST` zawierającą listę aktualnie dostępnych klientów. Jeżeli klient nie jest w stanie `CLIENT_CONNECTED`, to wysłanie tej wiadomości zostanie opóźnione aż klient nie będzie w tym stanie.

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.

Rozdział 5

Implementacja

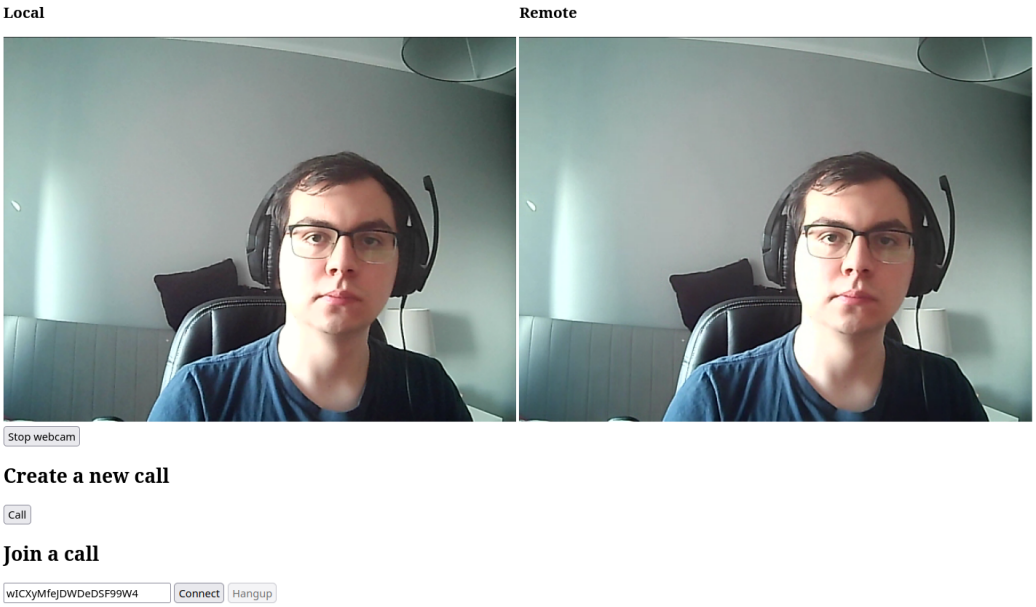
Niniejszy rozdział zawiera prezentację wykonanego oprogramowania wraz z omówieniem detali implementacyjnych. Zostanie omówiona przykładowa aplikacja webowa wykorzystująca API WebRTC w języku Javascript oraz wideokomunikator Piperchat będący aplikacją okienkową na systemy Linux wykorzystującą implementację WebRTC dostępną we frameworku GStreamer.

5.1. Aplikacja webowa z użyciem WebRTC

W niniejszym rozdziale zostanie omówiona aplikacja webowa wykorzystująca javascriptowe API WebRTC dostępne w przeglądarkach internetowych. Została ona wykonana w celu zapoznania się z WebRTC w jego docelowym środowisku w nadziei że zdobyta wiedza będzie kluczowa w realizacji aplikacji na systemy Linux, lecz ta okazała się zbędna. Zawartość poniższego rozdziału zatem nie powinna być traktowana jako potrzebna do zrozumienia zawartości kolejnych rozdziałów, lecz zaledwie jako przykład standaryzacji zapewnionej przez WebRTC przy budowaniu aplikacji multimedialnych P2P w różnych środowiskach.

5.1.1. Opis aplikacji

Aplikacja webowa implementuje minimalną funkcjonalność by umożliwić użytkownikom prowadzenie rozmów wideo P2P z użyciem WebRTC. Na rysunku 5.1 przedstawiono widok aplikacji po rozpoczęciu rozmowy.

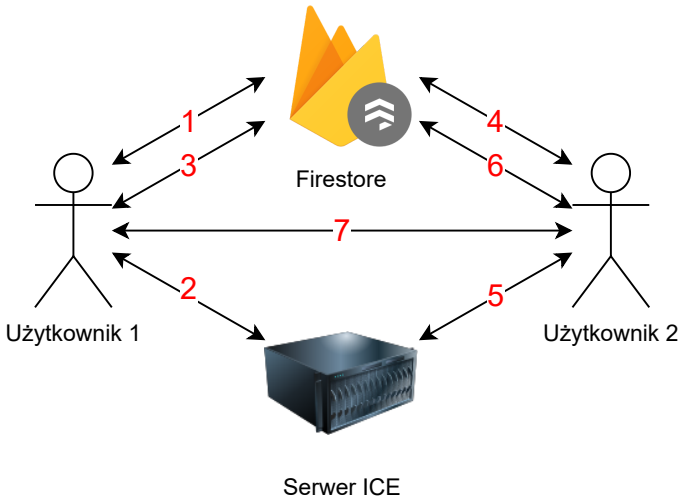


Rys. 5.1: Zrzut ekranu aplikacji WebRTC podczas połączenia na tym samym komputerze

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).

5.1.2. 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

5.1.3. Wybrane fragmenty kodu

W niniejszej sekcji zaprezentowane zostaną wybrane fragmenty kodu realizujące zadania niezbędne do działania aplikacji, takie jak przechwytywanie wideo i audio użytkownika, sygnalizacja połączenia, etc.

Zarządzanie strumieniami wideo i audio

Omówienie fragmentów kodu zostanie rozpoczęte od zaprezentowania przechwytywania oraz prezentacji lokalnych oraz zdalnych strumieni wideo i audio. Aplikacja ma do dyspozycji łącznie 4 strumienie audio:

- lokalny strumień wideo - jest prezentowany użytkownikowi oraz wysyłany w obiekcie `RTCPeerConnection`
- lokalny strumień audio - jest wysyłany w obiekcie `RTCPeerConnection`
- zdalny strumień wideo - jest odbierany z `RTCPeerConnection` i prezentowany użytkownikowi
- zdalny strumień audio - jest odbierany z `RTCPeerConnection` i prezentowany użytkownikowi

Strumienie lokalne otrzymywane są z funkcji `getUserMedia()`, natomiast strumienie zdalne są uzyskiwane ze zdarzenia `ontrack` obiektu `RTCPeerConnection`. Całość przedstawiono na listingu 5.1.

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;
});
```

Tworzenie połączenia

Aby utworzyć połączenie WebRTC, klient X musi skomunikować się z klientem Y przez jakiś inny kanał, tzw. out-of-band, wykorzystano do tego celu bazę danych czasu rzeczywistego Firebase (listing 5.2).

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 przypisano handler (listing 5.3):

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);
      }
    })
  })
});
```

Dołączanie do połączenia

By dołączyć do połączenia należy odczytać wygenerowaną w bazie ofertę i zarejestrować ją z obiektem `RTCPeerConnection`, a następnie wysłać odpowiedź. Proces ten zaprezentowano na listingu 5.4

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));
      }
    })
  });
});
```

5.1.4. Omówienie działania aplikacji na przykładzie

W niniejszej sekcji omówiono niektóre wiadomości generowane i odbierane przez obiekt `RTCPeerConnection` celem zaprezentowania działania WebRTC na konkretnym przykładzie.

Śledzenie procesu nawiązywania połączenia

W poniższej sekcji zostaną omówione dane wymieniane pomiędzy stronami na rzecz ustanowienia połączenia WebRTC.

Aby ustanowić połączenie peer-to-peer, należy rozwiązać poniższe problemy: [2]

- 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):

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):

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):

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.

Analiza pakietów protokołu SDP

Parametry ustanowionego połączenia są determinowane przez protokół SDP. Wygenerowana oferta SDP zawiera wiele pól które mogą być wykorzystane do negocjacji parametrów połączenia, co zaprezentowano na listingu 5.8.

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
```

[6] 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:

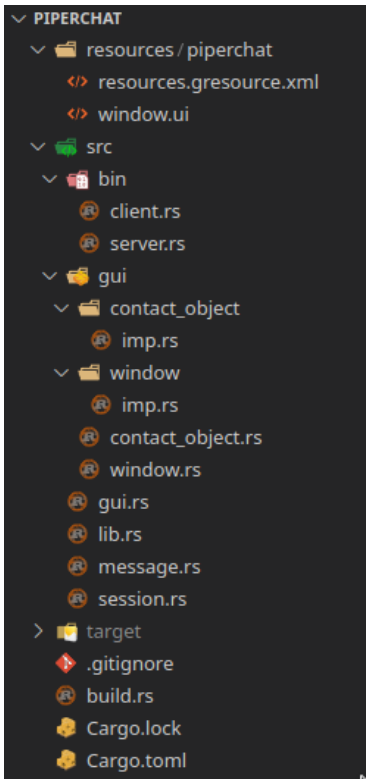
- `v=0`: wersja numeru protokołu - aktualnie 0 jest jedyną możliwą wersją
- `o=<username> <sess-id> <sess-version> <nettype> <addrtype> <unicast-address>`: kolekcja pól o inicjatorze sesji
  - `username: mozilla THIS_IS_SDPARTA-99.0`, co jest referencją do filmu 300
  - `sess-id: 107455341790422027`
  - `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.”[6]
- `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.

5.2. Piperchat

W niniejszym rozdziale zostanie przedstawiona implementacja wideokomunikatora Piperchat. Projekt składa się z serwera oraz klienta wykonanych w języku Rust. Klient używa także frameworków GTK oraz GStreamer.

Rysunek 5.3 prezentuje drzewo projektu.



Rys. 5.3: Drzewo projektu

- `/resources/piperchat` - zawiera pliki XML wykorzystywane przez framework GTK do budowy okna
  - `resources.gresource.xml` - zawiera listę wszystkich zasobów wykorzystanych w projekcie GTK. W tym wypadku wyliczony jest tylko plik window.ui.
  - `window.ui` zawiera definicję głównego okna wyrażoną w GTK XML
- `/src/` - zawiera cały kod źródłowy w języku Rust
  - `/src/bin/` - zawiera pliki źródłowe kompilowane jako osobne programy
    - `/src/bin/client.rs` - zawiera implementację klienta
    - `/src/bin/server.rs` - zawiera implementację serwera
  - `/src/lib.rs` - jest punktem wejściowym "biblioteki"tzn. zamieszczone tu elementy są dostępne dla wszystkich programów z katalogu `bin`
  - `/src/message.rs` - zawiera definicje wiadomości wysyłanych pomiędzy klientem a serwerem.
  - `/src/gui.rs` - definiuje moduł `gui` składający się z modułów `window` oraz `contact_object`
    - `/src/gui/window.rs` - zawiera definicję okna aplikacji jako obiektu GTK
      - `/src/gui/window/imp.rs` - zawiera implementację okna aplikacji jako obiektu GTK

    - `/src/gui/contact_object.rs` - zawiera definicję kontaktu (tj. innego połączonego i dostępnego użytkownika) jako obiektu GTK
        - `/src/gui/contact_object/imp.rs` - zawiera implementację kontaktu jako obiektu GTK
- `build.rs` zawiera skrypt budujący zasoby z katalogu `resources`
- `Cargo.toml` zawiera manifest projektu Rust wraz z listą bezpośrednich zależności
- `Cargo.lock` lockfile zawierający wyznaczone wersje wszystkich zainstalowanych zależności

5.2.1. 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

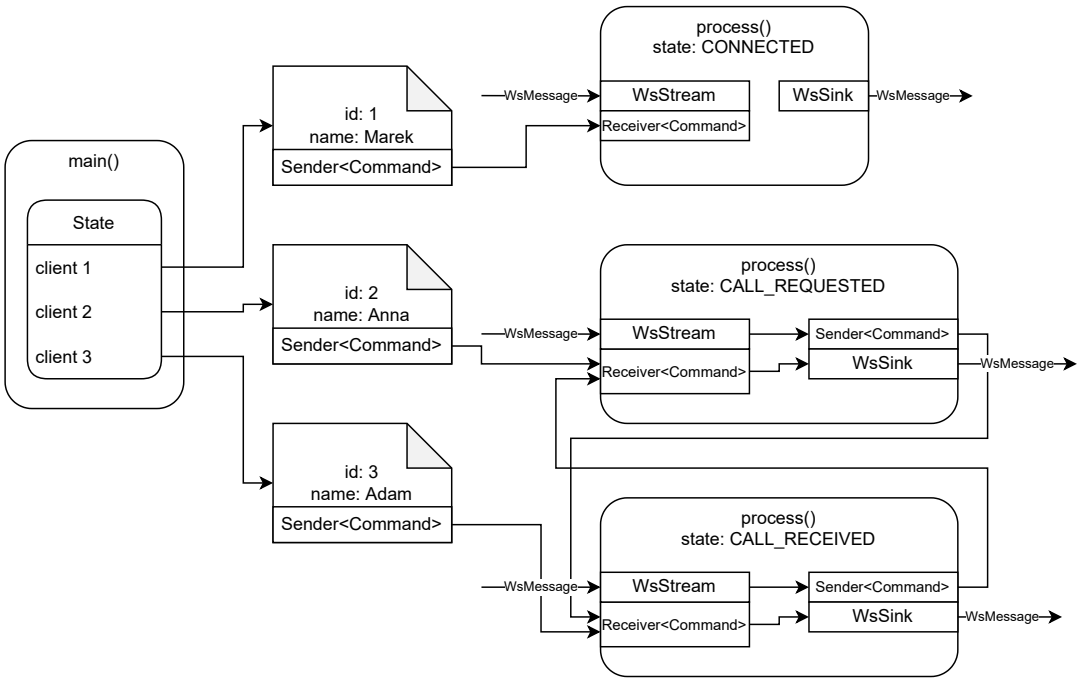
Rodzaje komunikatów możliwych do wysłania przez klient lub serwer są zebrane w odpowiedni typ wyliczeniowy a następnie w wiadomości tekstowej Websockets wysyłana jest ich serializacja w formacie JSON (ang. Javascript Object Notation).

```
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

Diagram 5.4 prezentuje wysokopoziomową strukturę działania serwera: dla każdego połączenia uruchamiane jest nowe zadanie które zajmuje się komunikacją z danym socketem. Wraz z zadaniami tworzone są kanały służące do wysyłania komend przez resztę systemu dla danego połączenia. Gdy rozpoczyna się nowe połączenie pomiędzy użytkownikami X i Y, zadanie użytkownika X otrzymuje wysyłacz komend do użytkownika Y i vice versa. Zadanie na wiadomości otrzymane od użytkownika reaguje wysyłając komendy przez wysyłacz do drugiej strony połączenia, a na otrzymane przez swój własny słuchacz komendy może reagować wysyłając wiadomości na swój socket.



Rys. 5.4: Diagram przepływu danych serwera

Fragmenty kodu

```rust
#[tokio::main]
async fn main() -> color_eyre::Result<()> {
    pretty_env_logger::init();
    color_eyre::install()?;
    let listener = TcpListener::bind("0.0.0.0:2137").await.unwrap();
    let state = Arc::new(Mutex::new(State::new()));

    loop {
        let (socket, _) = listener.accept().await.unwrap();
        let state = state.clone();
        tokio::spawn(async move { process(socket, state).await.unwrap() });
    }
}
```

Listing 5.1: Główna pętla serwera akceptująca przychodzące połączenia TCP

Główna pętla Listing 5.1 zawiera funkcję `main` serwera, która najpierw inicjalizuje handlery systemu logów oraz raportowania błędów, inicjalizuje obiekt `TcpListener`, a także globalny stan aplikacji który zawiera listę aktualnie połączonych użytkowników. Obiekt `State`, którego definicja znajduje się w listingu 5.2, synchronizowany jest pomiędzy wątkami za pomocą muteksu, a następnie muteks pakowany jest w `Arc` (Atomic Reference Counter), który jest odpowiednikiem `std::shared_ptr` z C++ i odpowiada za dzielenie muteksu pomiędzy wątki oraz zwolnienie go gdy liczba żyjących referencji osiągnie 0.

Dzięki użyciu atrybutu `tokio::main`, funkcja main jest asynchroniczna i można w niej korzystać z `.await`.

```rust
struct State {
    users: HashMap<u32, User>,
}

impl State {
    fn new() -> Self {
        State {
            users: HashMap::new(),
        }
    }
}

#[derive(Debug)]
struct User {
    name: String,
    id: u32,
    tx: mpsc::UnboundedSender<Command>,
}
```

Listing 5.2: Obiekt State zawierający globalny stan serwera

Funkcja process Funkcja process odpowiedzialna jest za komunikację z każdym z klientów. Wykonuje następujące zadania:

1. Otrzymuje wiadomość `Connect` od klienta
2. Dodaje klienta do obiektu State
3. Przechowuje aktualny stan klienta (połączony/wysłał lub otrzymał połączenie/w połączeniu, etc.)
4. Za pomocą makra `select!` nasłuchuje na jedno z możliwych wydarzeń:
   - Wiadomość przychodząca od użytkownika
   - Komenda przychodząca przez kanał
5. Po rozłączeniu, usunięcie z listy użytkowników w obiekcie `State`.

Stan użytkownika Algebraiczne typy danych dostępne w języku Rust znacząco ułatwiają modelowanie stanu programu. Oprócz *product types* dostępnych w także innych językach programowania jako struktury (ponieważ liczba wszystkich możliwych wartości struktury to iloczyn wartości jej pól), mamy także dostęp do *sum types* realizowanych poprzez pola wyliczeniowe `enum`, które dodatkowo mogą zawierać dane. W takim wypadku suma wszystkich możliwych wartości pola wyliczeniowego jest sumą wartości jego wariantów.

```rust
#[derive(Debug)]
enum ClientState {
    Connected,
    CallRequested(mpsc::UnboundedSender<Command>),
    CallReceived(mpsc::UnboundedSender<Command>),
    InCall(mpsc::UnboundedSender<Command>),
}
```

Listing 5.3: Pole wyliczeniowe możliwych stanów klienta

Listing 5.3 zawiera pole wyliczeniowe `ClientState`, które w wariantach Call może zawierać obiekt `UnboundedSender`, który służy do wysyłania wiadomości typu `Command` (listing 5.4) do odbiorników które znajdują się w innych klientach. W ten sposób odbywa się komunikacja pomiędzy klientami.

```
#[derive(Debug)]
enum Command {
    SendMessage(PcMessage),
    CallReceived {
        channel: mpsc::UnboundedSender<Command>,
        name: String,
    },
    CallAccepted(mpsc::UnboundedSender<Command>),
    PeerHungup,
    CallRejected,
}
```

Listing 5.4: Komendy możliwe do wysłania przez jednego klienta do drugiego

5.2.2. Klient

Aplikacja okienkowa składa się z czterech konkurentnie wykonujących się zadań:

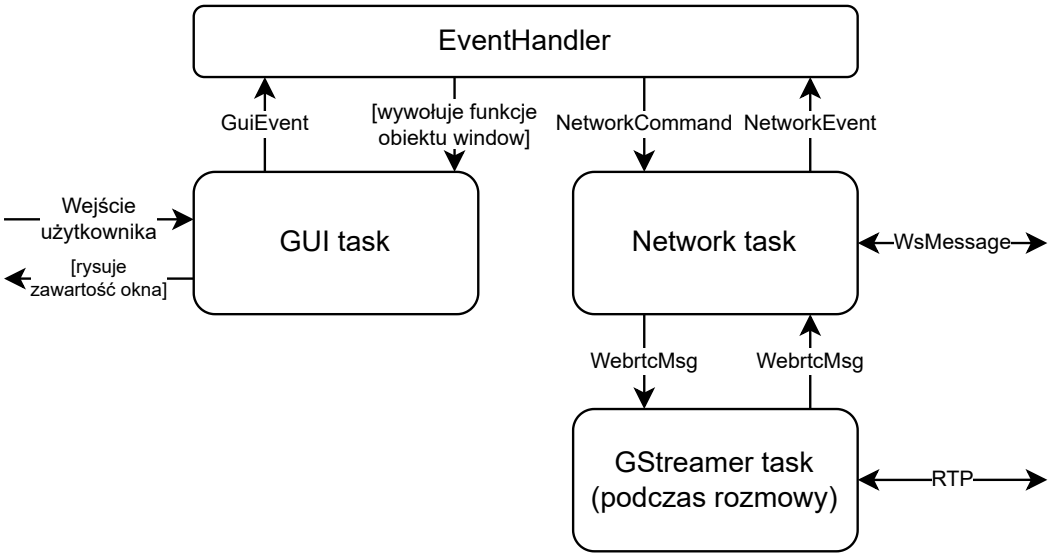
1. Zadanie GUI, rysuje okienko oraz emituje zdarzenia GUI
2. Zadanie połączenia z serwerem, nasłuchuje wiadomości wysłane przez serwer, emituje zdarzenia sieciowe, odbiera komendy do wykonania
3. Zadanie połączenia, prowadzi proces sygnalizacji i negocjacji WebRTC, a następnie prowadzi połączenie: prezentuje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

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

Diagram 5.5 obrazuje wysokopoziomową strukturę danych klienta. Główna funkcja aplikacji tworzy zadania do obsługi interfejsu graficznego (GUI task) oraz do obsługi połączenia z serwerem aplikacji (Network task). Komunikacja pomiędzy zadaniami odbywa się poprzez wysyłanie silnie typowanych wiadomości przez specjalne kanały. Obiekt `EventHandler` jednocześnie nasłuchuje na zdarzenia `GuiEvent` od zadania GUI oraz zdarzenia `NetworkEvent` od zadania połączenia, następnie reaguje na nie wykonując operacje na lub wysyłając komendy do tego drugiego.



Rys. 5.5: Diagram przepływu danych klienta

EventHandler

Obiekt `EventHandler` zajmuje się reagowaniem na zdarzenia z obu zadań. Listing 5.5 zawiera definicję obiektu oraz zawartość funkcji `start`, która uruchamia proces handlowania zdarzeń. Obiekt nasłuchuje na przychodzące zdarzenia, a następnie obsługuje je w osobnych funkcjach, które zostaną omówione w czasie omawiania poszczególnych zadań.

```
struct EventHandler {
    gui_rx: Receiver<GuiEvent>,
    network_rx: Receiver<NetworkEvent>,
    network_command_tx: Sender<NetworkCommand>,
    current_dialog: Option<MessageDialog>,
    window: Window,
}

impl EventHandler {
    fn start(mut self) {
        let event_handler = async move {
            loop {
                select! {
                    network_event = self.network_rx.next() => {
                        match network_event {
                            Some(event) => self.handle_network_event(event).await,
                            None => break
                        }
                    },
                    gui_event = self.gui_rx.next() => {
                        match gui_event {
                            Some(event) => self.handle_gui_event(event).await,
                            None => break
                        }
                    }
                }
            }
        };

        MainContext::default().spawn_local(event_handler);
    }
```

```
    ...
}
```

Listing 5.5: Definicja obiektu EventHandler oraz jego metody obsługującej zdarzenia

Zadanie GUI

GUI emituje następujące zdarzenia (listing 5.6):

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

Listing 5.6: Definicja pola wyliczeniowego GuiEvent

Są one obsługiwane w następujący sposób (listing 5.7):

```
async fn handle_gui_event(&mut self, event: GuiEvent) {
    match event {
        GuiEvent::CallStart(id, name) => {
            let dialog = adw::MessageDialog::new(
                Some(&self.window),
                Some(&format!("Calling {name}")),
                Some(&format!(
                "Waiting for a response from {name}. You can wait for the answer or hangup."
            )),
            );
            dialog.add_responses(&[("hangup", "Hang up")]);
            dialog.set_response_appearance("hangup", ResponseAppearance::Destructive);

            let network_command_tx = self.network_command_tx.clone();
            dialog.run_async(None, move |obj, response| {
                if response == "hangup" {
                    info!("SENDING HANGUP");
                    network_command_tx
                        .send_blocking(NetworkCommand::CallHangup)
                        .unwrap();
                }
            });
            self.current_dialog = Some(dialog);

            self.network_command_tx
                .send_blocking(NetworkCommand::CallStart(id, name))
                .unwrap();
        }
        GuiEvent::CallAccepted(preference) => {
            self.network_command_tx
                .send_blocking(NetworkCommand::CallAccept)
                .unwrap();
        }

        GuiEvent::CallRejected => {
            self.network_command_tx
                .send_blocking(NetworkCommand::CallReject)
```

```
                .unwrap();
        }
        GuiEvent::NameEntered(name) => {
            self.network_command_tx
                .send_blocking(NetworkCommand::Connect(name))
                .unwrap();
        }
    }
}
```

Listing 5.7: Metoda obsługująca zdarzenia GuiEvent

Zadanie połączenia

Zadanie połączenia emituje następujące zdarzenia (listing 5.8):

```
#[derive(Debug)]
pub enum NetworkEvent {
    UserlistReceived(Vec<(u32, String)>),
    CallReceived(String),
    CallAccepted,
    CallRejected(String),
    CallHangup(String),
}
```

Listing 5.8: Definicja pola wyliczeniowego NetworkEvent

oraz akceptuje następujące komendy (listing 5.9):

```
#[derive(Debug)]
pub enum NetworkCommand {
    CallStart(u32, String),
    CallAccept,
    CallReject,
    CallHangup,
    Connect(String),
}
```

Listing 5.9: Definicja pola wyliczeniowego NetworkCommand

Zdarzenia sieciowe są obsługiwane w następujący sposób:

```
async fn handle_network_event(&mut self, event: NetworkEvent) {
    info!("received network event: {event:?}");
    match event {
        NetworkEvent::UserlistReceived(userlist) => {
            self.window.set_contacts(userlist);
        }
        NetworkEvent::CallReceived(name) => {
            // display a dialog where user can accept/reject the message
            let dialog = adw::MessageDialog::new(
                Some(&self.window),
                Some(&format!("Incoming call from {name}")),
                Some(&format!(
                    "Receiving a call from {name}. Do you want to accept or reject this call?"
                )),
            );
```

```
            dialog.add_responses(&[("accept", "Accept"), ("reject", "Reject")]);
            dialog.set_response_appearance("accept", ResponseAppearance::Suggested);
            dialog.set_response_appearance("reject", ResponseAppearance::Destructive);
            let window = self.window.clone();

            dialog.run_async(None, move |_obj, response| match response {
                "accept" => {
                    window.accept_call();
                }
                "reject" => {
                    window.reject_call();
                }
                // here dialog got closed from outside, that means sender hung up
                _ => {
                    let dialog = adw::MessageDialog::new(
                        Some(&window),
                        Some(&format!("Caller hung up.")),
                        Some(&format!(
                            "The call from the user {name} terminated. Caller hung up."
                        )),
                    );

                    dialog.add_responses(&[("ok", "OK")]);
                    dialog.run_async(None, move |obj, response| {});
                }
            });
            self.current_dialog = Some(dialog);
        }
        NetworkEvent::CallHangup(name) => {
            info!("Received hangup");
            if let Some(dialog) = self.current_dialog.take() {
                dialog.close();
            }
        }
        NetworkEvent::CallAccepted => {
            if let Some(dialog) = self.current_dialog.take() {
                info!("CLOSING");
                dialog.close();
            }
        }
        NetworkEvent::CallRejected(name) => {
            if let Some(dialog) = self.current_dialog.take() {
                info!("CLOSING");
                dialog.close();
            }

            let dialog = adw::MessageDialog::new(
                Some(&self.window),
                Some(&format!("Call rejected")),
                Some(&format!("Recepient {name} rejected the call.")),
            );

            dialog.add_responses(&[("ok", "OK")]);
            dialog.run_async(None, move |obj, response| {});
        }
    }
}
```

Listing 5.10: Metoda obsługująca zdarzenia NetworkEvent

Zadanie GStreamer

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

Pipeline W konstruktorze obiektu z reprezentacji tekstowej tworzony jest pipeline kierujący lokalne strumienie wideo i audio do elementu webrtcbin (listing 5.11):

```
// 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 5.11: Reprezentacja tekstowa pipeline'u

Następnie do pipeline'u dołączane są zdalne strumienie mediów pojawiające się za pomocą sygnału `pad_added`. Przychodzące strumienie muszą najpierw zostać odkodowane za pomocą elementu decodebin (listing 5.12):

```
app.webrtcbin.connect_pad_added(move |_webrtc, pad| {
    let app = upgrade_weak!(app_clone);

    if let Err(err) = app.on_incoming_stream(pad) {
        gst::element_error!(
            app.pipeline,
            gst::LibraryError::Failed,
            ("Failed to handle incoming stream: {:?}", err)
        );
    }
});

...

// Whenever there's a new incoming, encoded stream from the peer create a new decodebin
fn on_incoming_stream(&self, pad: &gst::Pad) -> Result<(), anyhow::Error> {
    // Early return for the source pads we're adding ourselves
    if pad.direction() != gst::PadDirection::Src {
        return Ok(());
    }

    let decodebin = gst::ElementFactory::make("decodebin").build().unwrap();
    let app_clone = self.downgrade();
    decodebin.connect_pad_added(move |_decodebin, pad| {
        let app = upgrade_weak!(app_clone);

        if let Err(err) = app.on_incoming_decodebin_stream(pad) {
            gst::element_error!(
                app.pipeline,
                gst::LibraryError::Failed,
                ("Failed to handle decoded stream: {:?}", err)
            );
        }
    });

    self.pipeline.add(&decodebin).unwrap();
    decodebin.sync_state_with_parent().unwrap();
```

```
    let sinkpad = decodebin.static_pad("sink").unwrap();
    pad.link(&sinkpad).unwrap();

    Ok(())
}
```

Listing 5.12: Dekodowanie zdalnych strumieni

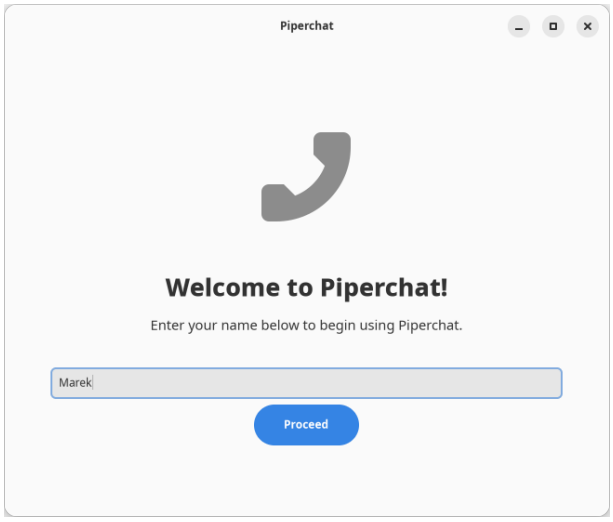
Następnie zdekodowane strumienie są dodawane do pipeline'u w funkcji `on_incoming_decodebin_stream(pad)` (listing 5.13):

```
// Handle a newly decoded decodebin stream and depending on its type, create the relevant
// elements or simply ignore it
fn on_incoming_decodebin_stream(&self, pad: &gst::Pad) -> Result<(), anyhow::Error> {
    let caps = pad.current_caps().unwrap();
    let name = caps.structure(0).unwrap().name();

    let sink = if name.starts_with("video/") {
        gst::parse_bin_from_description(
            "queue ! videoconvert ! videoscale ! autovideosink",
            true,
        )?
    } else if name.starts_with("audio/") {
        gst::parse_bin_from_description(
            "queue ! audioconvert ! audioresample ! autoaudiosink",
            true,
        )?
    } else {
        println!("Unknown pad {:?}, ignoring", pad);
        return Ok(());
    };

    self.pipeline.add(&sink).unwrap();
    sink.sync_state_with_parent()
        .with_context(|| format!("can't start sink for stream {:?}", caps))?;

    let sinkpad = sink.static_pad("sink").unwrap();
    pad.link(&sinkpad)
        .with_context(|| format!("can't link sink for stream {:?}", caps))?;

    Ok(())
}
```

Listing 5.13: Dodawanie zdekodowanych strumieni do pipeline'u

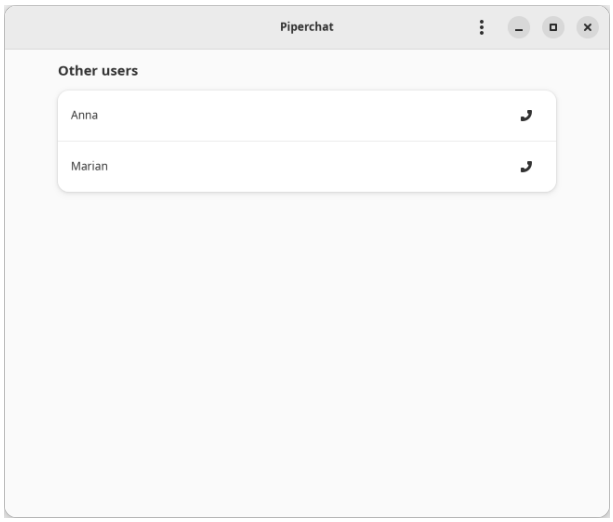
Interfejs użytkownika

W niniejszym rozdziale zaprezentowany zostanie interfejs użytkownika wykonany za pomocą frameworka GTK oraz biblioteki libadwaita.
Po uruchomieniu aplikacji prezentowany jest ekran lądowania który wyświetla wiadomość powitalną oraz prosi o podanie nazwy użytkownika (rys. 5.6).



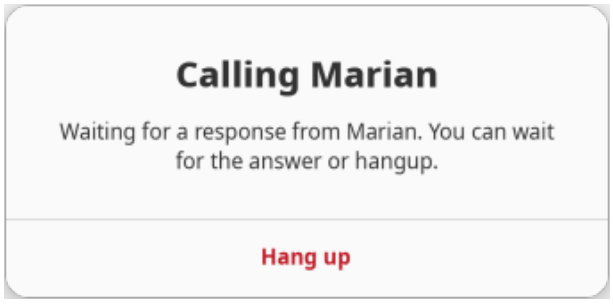
Rys. 5.6: Ekran lądowania

Po wprowadzeniu nazwy użytkownika aplikacja łączy się z serwerem, ogłasza dostępność użytkownika innym połączonym użytkownikom oraz wyświetla użytkownikowi listę innych połączonych użytkowników (rys. 5.7).



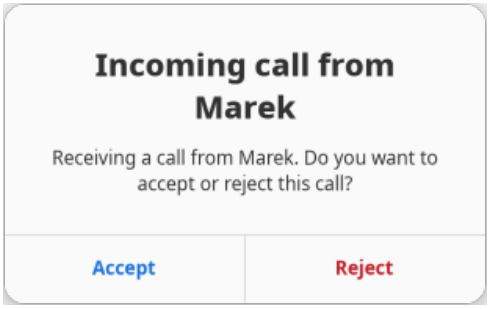
Rys. 5.7: Główny ekran aplikacji

Użytkownik może zadzwonić do drugiego użytkownika klikając na przycisk rozpoczęcia połączenia po prawej stronie nazwy innego użytkownika. Podczas oczekiwania na odpowiedź od odbiorcy, wyświetlane jest następujące okno dialogowe, gdzie nadawca może przerwać połączenie (rys. 5.8):



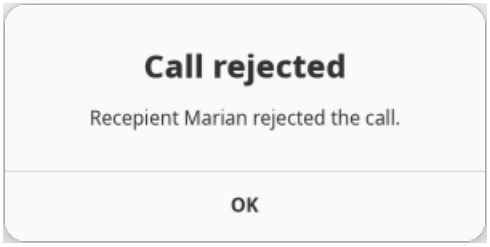
Rys. 5.8: Dialog wysłania połączenia

Tymczasem odbiorcy prezentowane jest okno dialogowe informujące o przychodzącym połączeniu. Odbiorca może odebrać lub odrzucić połączenie (rys. 5.9):



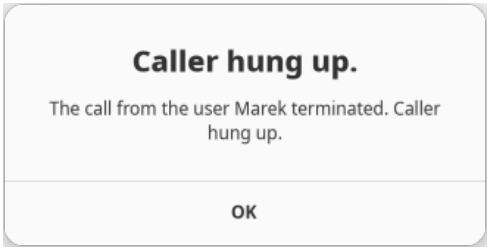
Rys. 5.9: Dialog otrzymania połączenia

Jeżeli odbiorca odrzuci połączenie, nadawcy wyświetla się okno dialogowe informujące o zdarzeniu (rys. 5.10):



Rys. 5.10: Dialog odrzucenia połączenia przez odbiorcę

Jeżeli połączenie zostanie przerwane przez nadawcę zanim nadawca zdąży odebrać albo odrzucić połączenie, okno dialogowe informujące o zdarzeniu wyświetlane jest odbiorcy (rys. 5.11):



Rys. 5.11: Dialog porzucenia połączenia przez nadawcę

Rozdział 6

Podsumowanie

6.1. Realizacja celu pracy

Zadanie utworzenia komunikatora prezentującego reguły prowadzenia połączeń peer-to-peer zostały zrealizowane. Wykonany komunikator realizuje postanowione wymagane wymagania funkcjonalne, jednak nie wykorzystuje kodeka AV1, ponieważ dostępne w GStreamer enkodery AV1 okazały się jeszcze niewystarczająco szybkie do kodowania wideo w czasie rzeczywistym. Modularna architektura frameworka GStreamer pozwoliła na bezproblemowe zastąpienie AV1 innym nowoczesnym kodekiem - VP8. W miarę rozwoju enkoderów programowych a także coraz większej dostępności enkoderów sprzętowych w nowoczesnych układach graficznych, użycie AV1 w komunikatorach wideo może jeszcze stać się praktyczne w niedalekiej przyszłości.

6.2. 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.
Ponadto, możliwe jest lepsze wykorzystanie systemu typów języka Rust do modelowania stanu aplikacji tak, by niepoprawne stany były nieosiągalne. Gdyby zaczynać projekt od nowa, autor na początku dążyłby do wykonania kompletnego modelu celem poprawnej separacji warstw.

6.3. 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] L. Moreira. “Digital video introduction”. (), adr.: https://github.com/leandromoreira/digital_video_introduction. (accessed: 20.01.2023).

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

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

[4] Wikipedia contributors. “GTK — Wikipedia, The Free Encyclopedia”. [Online; accessed 30-January-2023]. (2023), adr.: https://en.wikipedia.org/w/index.php?title=GTK&oldid=1136033946.

[5] “GUI development with Rust and GTK 4”. (), adr.: https://gtk-rs.org/gtk4-rs/stable/latest/book/#gui-development-with-rust-and-gtk-4. (accessed: 20.01.2023).

[6] 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.

Dodatek A

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,

Dodatek B

Opis załączonej płyty CD/DVD

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