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

WROCŁAW, 2022

Streszczenie

Streszczenie w języku polskim powininno zmieścić się na połowie strony (drugą połowę powinien zająć abstract w języku angielskim).

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

WebRTC (ang. *Web Real Time Communications*)
ICE (ang. *Interactive Connectivity Establishment*)
STUN (ang. *Session Traversal Utilities for NAT*)
TURN (ang. *Traversal Using Relays around NAT*)

Rozdział 1

Wstęp

1.1. Wprowadzenie

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 oraz efektywność transmisji wideo stają się bardzo ważnymi problemami.

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 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

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

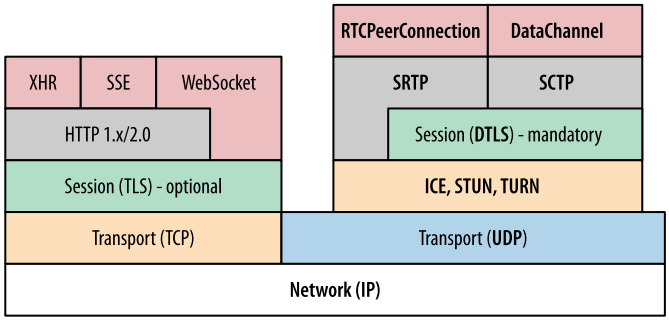
Rozdział 3

WebRTC

3.1. Co to jest, z czego się składa?

[1]

WebRTC jest projektem zapewniającym przeglądarkom możliwości komunikacji czasu rzeczywistego, 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 3.1.



Rys. 3.1: Stos protokołów w WebRTC - wzięty z HPBN - do zmiany

3.2. Sygnalizacja nawiązania połączenia

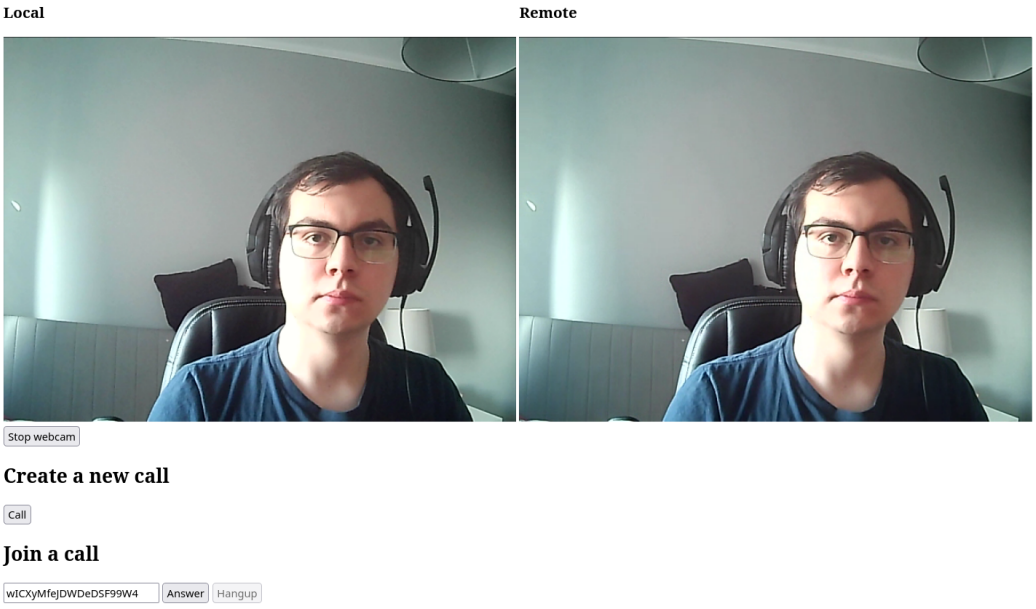
3.3. Interactive Connectivity Establishment (ICE)

3.4. Opis API WebRTC

Rozdział 4

Aplikacja webowa z użyciem WebRTC

W poniższym rozdziale zostanie omówiona aplikacja webowa wykorzystująca WebRTC.

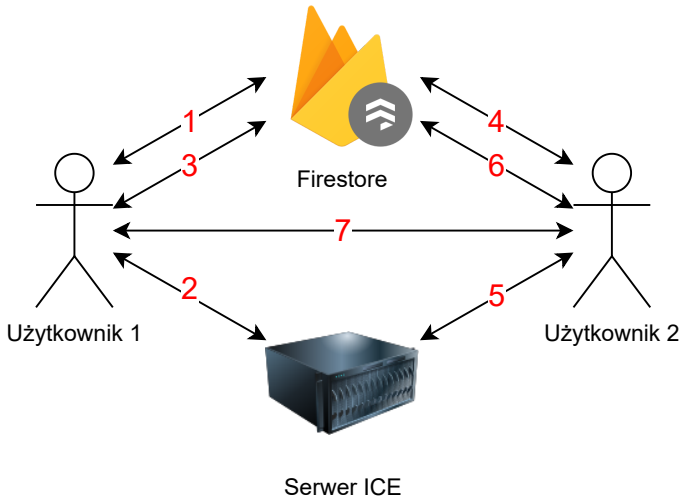


Rys. 4.1: Zrzut ekranu aplikacji WebRTC podczas połączenia na tym samym komputerze

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 4.2.



Rys. 4.2: 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).

4.1. 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

4.2. Wybrane fragmenty kodu

4.2.1. Przechwytywanie wideo i audio

Listing 4.1: 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;
});
```

4.2.2. 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 4.2: 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 4.3: 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);
      }
    })
  })
});
```

4.2.3. Dołączanie do połączenia

Listing 4.4: 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));
      }
    })
  });
});
```

4.3. Omówienie działania aplikacji na przykładzie

4.3.1. 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 4.5: Dokument połączenia po utworzeniu przez użytkownika 1

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

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

Listing 4.6: 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 4.7: 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.

4.3.2. Analiza pakietów protokołu SDP

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

Listing 4.8: 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 4.8, 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 5

Mechanizmy kompresji wideo

5.1. Koncepty kompresji

5.2. Kontenery

5.3. Kodeki

5.3.1. H.264

5.3.2. H.265

5.3.3. VP9

5.3.4. AV1

Rozdział 6

Architektura projektu Piperchat

6.1. Serwer

6.2. Klient

Rozdział 7

Aplikacja desktopowa w języku Rust

7.1. GUI framework

7.2. ffmpeg

Bibliografia

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

[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.