TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Automatic real-time transcription of multimedia conference

Master thesis

Study programme:	N2612 – Electrical Engineering and Informatics
Study branch:	3906T001 – Mechatronics
Author:	**Bc. Anna Kamenskaia**
Supervisor:	Ing. Ondřej Smola

Liberec 2019

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Automatic real-time transcription of multimedia conference

Diplomová práce

Studijní program:	N2612 – Electrical Engineering and Informatics
Studijní obor:	3906T001 – Mechatronics
Autor práce:	**Bc. Anna Kamenskaia**
Vedoucí práce:	Ing. Ondřej Smola

Liberec 2019

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics and Interdisciplinary Studies

Master Thesis Assignment Form

Automatic real-time transcription of multimedia conference

Name and Surname:	**Bc. Anna Kamenskaia**
Identification Number:	M18000224
Study Programme:	N2612 Electrical Engineering and Informatics
Specialisation:	Mechatronics
Assigning Department:	Institute of Information Technology and Electronics
Academic Year:	**2018/2019**

Rules for Elaboration:

1. Describe the current status of open source technologies and solutions used in multimedia conferences.
2. Describe and discuss possible solutions for capturing live conference audio. Describe the current state of real-time audio speech transcription software and select at least one platform to be used as a speech recognition backend.
3. Implement demo conferencing room that will capture every attendee audio and transcribe it using modular speech recognition backend.
4. Integrate your solution with one chosen open source web conference platform.
5. Discuss scalability and deployment requirements of your solution.

Scope of Graphic Work:	Dle potřeby dokumentace
Scope of Report:	cca 40-50 stran
Thesis Form:	printed/electronic

List of Specialised Literature:

[1] ROY, Radhika Ranjan. Handbook of SDP for multimedia session negotiations: SIP and WebRTC IP telephony. Boca Raton, FL: CRC Press/Taylor & Francis Group, 2018. ISBN 9781138484498.

[2] JOHNSTON, Alan B. SIP: understanding the Session Initiation Protocol. 3rd ed. Boston: Artech House, c2009. ISBN 1607839954.

[3] GRIGORIK, Ilya. High-performance browser networking. Sebastopol, CA: O'Reilly, 2013. ISBN 1449344763.

Thesis Supervisor:	Ing. Ondřej Smola Institute of Information Technology and Electronics
Date of Thesis Assignment:	18 October 2018
Date of Thesis Submission:	30 April 2019

L. S.

prof. Ing. Zdeněk Plíva, Ph.D.
Dean

prof. Ing. Ondřej Novák, CSc.
head of institute

Liberec 18 October 2018

Declaration

Abstrakt

Cílem práce je řešení pro přepis multimediální konference založené na protokolu WebRTC v reálném čase za pomoci kombinace existujících technologií a řešení v oblasti konferencí, přenosu médií a rozpoznávání řeči. Aplikace je naprogramována v Javě. Pro signalizaci se používá protokol WebSocket a pro přenos audio dat protokol RTP. Součástí řešení je modulární transkripční back-end využívající rozhraní Google Cloud Speech-to-text API a řešení pro rozpoznávání řeči vyvinuté v Laboratoři počítačového zpracování řeči (SpeechLab) [1] na Technické univerzitě v Liberci. Přepisy jsou zobrazeny v prohlížečích účastníků v reálném čase a zároveň jsou zapisovány do souboru. Práce obsahuje příklady přepisovaných konverzací.

Klíčová slova: WebRTC, multimediální konference, rozpoznávání řeči v reálném čase, přepis řeči.

Abstract

This work focuses on performing real-time transcription of a multimedia conference based on WebRTC protocol by combining existing technologies and solutions in conferencing, media transmission and speech recognition in one application. The result application is written in Java. It uses WebSocket to communicate with a conferencing application, RTP for receiving audio data and suggests modular transcription back-ends with Google Cloud Speech-to-text API and speech recognition engine developed by the Laboratory of Computer Speech Processing (SpeechLab) [1] in Technical University of Liberec already successfully integrated. Transcripts are stored in files and also can be displayed in browsers in real-time. Examples of transcribed conversations are provided.

Key words: WebRTC, multimedia conference, real-time speech recognition, transcription.

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Smola for all his valuable advices, which helped me to solve the task and overcome the difficulties I have encountered.

Contents

List of abbreviations

API	Application Programming Interface
FIFO	First in, first out
IP	Internet Protocol
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JDK	Java Development Kit
JSON	JavaScript Object Notation
NAT	Network Address Translation
SDK	Software Development Kit
SRE	Speech Recognition Engine
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TUL	Technical University of Liberec
UDP	User Datagram Protocol
VoIP	Voice over Internet Protocol
XMPP	eXtensible Messaging and Presence Protocol

List of Figures

1 Introduction

Multimedia conferencing allows live communication for people residing in different locations. Widely known solutions such as Skype, Google Hangouts, Zoom, Discord and others join many people for communication in business, education and entertainment activities. The WebRTC (Web Real-Time Communications) project was introduced around 2011 and provided technologies and tools for building browser-based multimedia conferencing solutions, connecting browsers, mobile platforms and IoT devices with a common set of communication and data transmission protocols [2]. WebRTC allows participating in multimedia conferences without installing additional software except a web browser.

Speech recognition is the ability of computer to convert human spoken speech into text representation. Speech recognition software involves advanced methods and technologies of computer science such as big data, deep learning and neural networks. Speech recognition is widely used in all areas of human society: science, education, military, business, telephony and daily live. Increasing accuracy and power of speech recognition software allows more and more advanced applications [3].

Transcription is the process of representing speech in written form. A transcript is a written record of spoken language [4]. If an important meeting is held in a conference room, there can possibly be a special person - transcriber, who would literally transcribe everything spoken on this meeting by hand or using computer. Now, if a conference can be held online in the browser, why would not we perform transcription automatically using various means of web communication and data transmission together with latest achievements in the speech recognition field?

Offline transcriptions are not a rarity nowadays. An online meeting can be recorded and then sent to a speech recognition service. Youtube demonstrates a high accuracy of transcribed speech - one can upload a recording and the service will automatically generate subtitles. But there is not much information on performing live multimedia conference transcription. Some commercial solutions offer such functionality but technical details are not available. Community lacks open-source solutions. Skype introduced call transcription last year in December [5]. All these identifies that this field is relevant. Live multimedia conference transcription is a modern and demanded task which justifies the relevancy of this diploma thesis.

This work focuses on solving various tasks and issues that may be encountered while building a solution for real-time web conference transcription. The problem can be decomposed in three main parts: capturing live stream audio, performing speech recognition using third-party services and processing the results. All these must be done as close to real-time as possible. A lot of problems may arise while fulfilling this requirement. The processes of searching for solutions and their implementations are described in this thesis.

An overview of existing technologies and protocols used for building web applications with conferencing functionality is performed in chapter 2.

Chapter 3 is dedicated to solution architecture. It is one of the most important chapters as it describes selection of conferencing platform, speech recognition back-end, the rest of necessary technological stack and how all these elements are connected with each other.

Possible ways to capture live stream audio are described in chapter 4 as well as the particular implementation used in our solution.

In chapter 5 we would like to describe the details of writing client solutions for the chosen speech recognition services.

Chapter 6 describes problems that can be encountered while performing live speech recognition for multiple audio streams and their solution. The matters of persistent data storage and its live representation are also discussed in this chapter.

Deployment of the solution is explained in chapter 7. There are also example conversation transcripts provided for different spoken languages as well as suggestions for application scaling and its further development.

2 Technologies used in multimedia conferencing

2.1 WebRTC

Web Real-Time Communication (WebRTC) is a collection of standards, protocols, and JavaScript APIs, the combination of which enables peer-to-peer audio, video, and data sharing between browsers (peers). However, it is not limited to browser communication and can be integrated with VoIP systems and SIP clients. Instead of relying on third-party plug-ins or proprietary software, WebRTC turns real-time communication into a standard feature that any web application can leverage via a simple JavaScript API [6].

There are three major components of WebRTC API which provide all the complex functionality required for a browser to support peer-to-peer data exchange, audio and video processing, and required network protocols:

- *MediaStream*: access to user's media;
- *RTCPeerConnection*: exchange of audio and video data;
- *RTCDataChannel*: transfer of arbitrary data.

WebRTC uses UDP on the transport layer: latency and timeliness are more critical than reliability. Several transport protocols layered on top of UDP are used for transport:

- Datagram Transport Layer Security (DTLS) is used for secure transport of application data;
- Secure Real-Time Transport (SRTP) is used to transport media;
- Stream Control Transport Protocol (SCTP) is used to transport application data.

Current WebRTC implementations use two default codecs for the media: OPUS for the audio and VP8 for the video [6]. There are also optional iSAC, iLBC, PCMA, PCMU audio codecs and VP9 video codec [7].

How to establish a connection between two WebRTC peers if they probably reside in their own networks and behind NAT? Most likely, neither of peers can be reached directly. Moreover, unlike a server which is expected to be opened for connections, a WebRTC peer may be unreachable, busy, or unwilling to initiate connection. As a result, the following problems must solved to succesfully establish peer-to-peer connection:

1. A remote peer must be notified about opening connection so it would start listening for the incoming packets;

2. Potential routing paths must be identified on both sides of connection and shared between peers;

3. Peers must exchange necessary information about media parameters: protocols, encodings [6].

2.2 STUN, TURN and ICE

Session Traversal Utilities for NAT (STUN) allow a host to determine the public IP address and port allocated to it in presence of a network address translator. To do so, it must send a request to a STUN server residing in pubic network and it would reply with a public IP address and the port of the client as they are seen from the public network. Unfortunately, STUN is not sufficient to deal with all possible network topologies and in some cases UDP may be blocked by a firewall [6].

Whenever STUN fails, Traversal Using Relays around NAT (TURN) protocol comes as a fallback. It relies on some public relay which transfers data between peers, so the connection is not actually peer-to-peer. This approach is reliable but the cost is high as well - the relay peer must possess enough capacity to serve all data flows. For this reason it should be used only when direct connection fails to establish [6].

Interactive Connectivity Establishment (ICE) is a built-in mechanism of WebRTC framework which is responsile for discovering routes and check connectivity between peers. Each *RTCPeerConnection* has its own ICE agent. ICE agent obtains local IP addresses and port tuples from the operating system, queries a STUN server and appends a TURN server as a fallback candidate if they are configured. The application is notified via a callback function. Once this process is complete, an SDP offer can be generated and delivered to the other peer through signalling channel. Once the remote session description is set on the *RTCPeerConnection* object, which now contains a list of candidate IP and port tuples for the other peer, the ICE agent begins connectivity checks. If a STUN binding request is confirmed by the other peer then the routing path is established [6].

2.3 SDP

When initiating multimedia conferences, VoIP calls, streaming media or other sessions, it is necessary to convey media details, transport addresses and other session description metadata to the participants. Session Description Protocol (SDP) provides a standard representation for such information, irrespective of how that information is transported. SDP does not deliver any media by itself but is used between endpoints for negotiation of media type, format, and all associated properties. The set of properties and parameters are often called a session profile [8][9].

SDP include five major components: session metadata, stream, Quality of Service (QOS), network and security. Session metadata contains information about SDP protocol version, originator of the session and its duration. The stream description contains details about media (audio, video) transported within a session. The QOS description contains all performance parameters of media streams. The network parameters describe what kind of transport and network protocol is used. The security parameters may include encryption key, authentication, authorization, integrity [10].

SDP provides offer/answer communication model. In this model, one participant in the session generates an SDP offer - specification of the set of media streams and codecs the offerer wishes to use, along with the IP addresses and ports the offerer would like to use to receive the media. The offer is conveyed to the other participant (answerer) by some means of transport. The answerer generates an SDP answer responding to the provided offer. The answer has a matching media stream for each stream in the offer, indicating whether the stream is accepted or not, along with the codecs that will be used and the IP addresses and ports that the answerer wants to use to send and/or receive media [10]. Such communication model is used in Session Initiation Protocol (SIP).

An example session description generated by the application developed within this diploma thesis:

```
v=0
o=Transcriber IN IP4 147.230.165.35
s=WebRTC conference transcription
c=IN IP4 147.230.165.35
t=0 0
a=recvonly
m=audio 49170 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

The "v=" field defines the version of the Session Description Protocol. The session is originated by *Transcriber* at IPv4 address 147.230.165.35. Session name is "WebRTC conference transcription". The connection address is equal to session origin address in this case. The "a=recvonly" line tells that this host is instructed only to receive media. Last two lines tell that host listens for incoming audio streams at port 49170 and specify the media format – RTP/AVP payload type 0 (defined in RFC 3551 as PCMU [11]) which is mapped to PCMU (µ-law encoded PCM audio) sampled at 8000 Hz.

2.4 SIP

Session Initiation Protocol (SIP) is an application layer signaling, presence, and instant messaging protocol which facilitates the creation of multimedia application services such as video conferencing [12]. SIP employs RTP over UDP for transport and SDP for session capabilities negotiation. Although SIP is usually mentioned from telephony perspective, it can be used to establish sessions having little in common with telephony. SIP infrastructure includes many types of client and server endpoints, the most important among them are described below:

- SIP user agent (UA) is a SIP-enabled end device which helps to establish connections with other UAs;
- A back-to-back user agent (B2BUA) is a type of SIP UA that receives a SIP request, then reformulates the request and sends it out as a new request. Can be used for organizing an anonymizer service to connect UAs without exposing any contact information;
- SIP gateway provides interface between a SIP network and another network utilizing different signalling protocol. Possible applications include topology hiding, media traffic management, media encryption, access control and more;
- SIP proxy server forwards SIP messages between user agents. Although UAs can communicate directly if the IP addresses are known but it is not a common situation. SIP proxy typically has access to user databases and can determine the route. SIP proxy has no media capabilities, does not generate requests (only responses to UAs) and relies only on SIP headers without parsing message bodies;
- SIP registrar server registers SIP user accounts. The user database can be used by other SIP servers (for example, proxy servers) within the same administrative domain [12].

SIP is not the only signalling protocol which can be used with WebRTC, for example, Jingle or ISDN User Part can be used as well. Actually, WebRTC standards defer the choice of signaling transport and protocol to the application, so custom implementation of signaling is acceptable [6].

2.4.1 Integration of WebRTC and SIP

Telecommunication solutions based on the SIP architecture and WebRTC solutions have a lot in common so the idea of building conferencing solutions available both for browser and regular SIP clients is quite natural. It can be achieved if there is some translation gateway which provides interface between SIP and signalling protocol implemented in WebRTC part. Another idea is based on RFC 7118 [13] which describes usage of WebSocket protocol as a transport between SIP infrastructures and web-oriented solutions. The components responsible for integration would be a WebRTC client with signalling functionalities implemented with WebSocket SIP API and a SIP proxy with WebSocket interface. As for media plane, some mandatory WebRTC protocols may not be supported by SIP clients so a media gateway may also be required [14].

2.5 RTP

The Real-time Transport Protocol provides delivery services for data with real-time charasteristics, such as audio and video. Those services include payload type identification, sequence numbering, timestamping and delivery monitoring. RTP runs over UDP and is usually used together with the RTP Control Protocol (RTCP). RTCP is used to monitor transmission statistics and quality of service (QoS) and aids synchronization of multiple streams [11]. RTP is widely used in communication and media systems that involve streaming media, such as telephony, television services and conferencing applications including WebRTC.

A complete specification of RTP for a particular application usage requires profile and payload format specifications. The profile defines the codecs used to encode the payload data and their mapping to payload format codes in the Payload Type (PT) field of the RTP header. Each profile is accompanied by several payload format specifications, each of which describes the transport of a particular encoded data. For this reason, RTP is accompanied by SDP confidentiality, message authentication, and replay protection to the RTP traffic [15].

Secure Real-time Transport Protocol (SRTP) is a profile of the RTP, which can provide confidentiality, message authentication, and replay protection to the RTP traffic and to the control traffic for RTP, the RTCP [16].

2.6 WebSocket

WebSocket enables bidirectional, message-oriented streaming of text and binary data between client and server. It is the closest API to a raw network socket in the browser. WebSocket is one of the most versatile and flexible transports available in the browser. The simple and minimal API enables us to layer and deliver arbitrary application protocols between client and server – anything from simple JSON payloads to custom binary message formats – in a streaming fashion, where either side can send data at any time. WebSocket provides low latency delivery of text and binary application data in both directions over the same TCP connection. The WebSocket resource URL uses its own custom scheme: ws for plain-text communication and wss when an encrypted channel (TCP+TLS) is required. WebSocket protocol is a fully functional, standalone protocol than can be used outside the browser. Its primary application is as a bidirectional transport for browser-based applications [6].

2.7 Conferencing platforms

The main advantage of the WebRTC technology is that it allows peer-to-peer, or, more precisely, browser-to-browser communication with little intervention of server, which is usually intended for signaling only. One-to-one connections are easy to manage and deploy: the peers talk directly to each other and no further optimization is required. However, this approach is sufficient only for creating very simple web applications. Features such as group calls, media stream recording and processing, media broadcasting are hard to implement on top of it. For example, in case of a group call a peer is required to send his video/audio stream to every other attendee while receiving a video/audio stream from each of them. This is quite resource-demanding and potentially leads to poor performance when increasing the number of participants in a call beyond two. As a result, multiparty applications should carefully consider the architecture of how the individual streams are aggregated and distributed between the peers. Possible ways to organize a multiparty architecture are illustrated on figure 2.1.

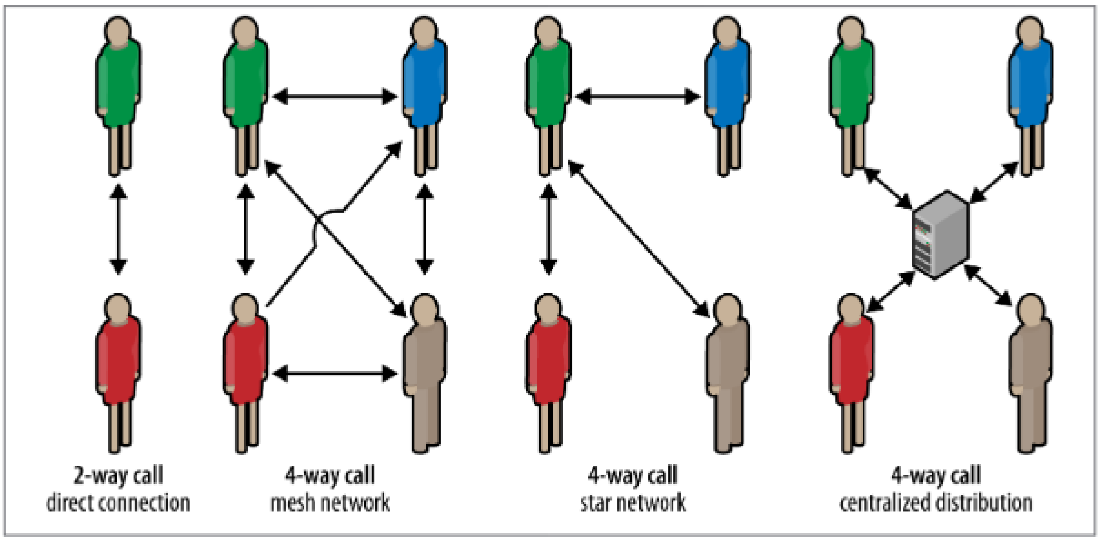


Figure 2.1: Distribution architecture for an N-way call [6]

.

While mesh networks are easy to set up, they are often inefficient for multiparty systems. It would be nice to reduce the number of streams a peer needs to send or even receive. To address this, an alternative strategy is to use a "star" topology instead, where the individual peers connect to a "supernode", which is then responsible for distributing the streams to all connected parties. This way only one peer has to pay the cost of handling and distributing N-1 streams, and everyone else talks directly to the super-node. A supernode can be another peer or it can be a dedicated service. WebRTC enables peer-to-peer communication but it does not mean that one should not consider a centralized infrastructure [6]. The concept of a WebRTC server needs to be introduced here. Basically, a WebRTC server acts as an intermediate node where media traffic goes through while moving between peers .

2.7.1 Types of WebRTC servers

There are two main types of WebRTC servers. If it only acts as a relay, it is called SFU (Selective Forwarding Unit), meaning its main purpose is forwarding media streams between clients [17]. Also, there is a concept of MCU (Multipoint Control Unit) which does not just forward media streams but operates on them and may modify them in some way: record, transcode, mix multiple streams into one and then send to the clients. MCU acts as a central entity every participant is talking to. It receives media from each participants, mixes into one stream, performs necessary operations and sends it to participants [18]. From browser perspective each participant is speaking only to one person. Unlike this, in case of using SFU each participant would have an uplink with his data and as many downlinks as there are people he is speaking with. There is no generalized opinion of what is better: SFU or MCU. The best selection depends on task.

2.7.2 Janus

Janus us a general purpose WebRTC server. Its core is designed to provide only the minimal functionality necessary to set up WebRTC communication. Any specific feaute needs to be implemented as a plugin. Example of such plugins can be implementations of applications like echo tests, conference bridges, media recorders, SIP gateways and the like [19]. Janus is lightweight and limited in basic installation but highly customizable.

2.7.3 Jitsi Videobridge

Jitsi is an open-source collection of VoIP and web conferencing oriented applications and libraries. The main projects are Jitsi Videobridge and Jitsi Meet. Jitsi Meet is a full conferencing application written in JavaScript working with Jitsi Videobridge. Jitsi Videobridge is a SFU, implements XMPP for signalling [20].

2.7.4 Kurento Media Server

Unlike Janus and Jitsi, Kurento is a WebRTC capable media server providing both SFU and MCU functionality. It is written in Java and combines the Mobicents/J-Boss application server and the GStreamer multimedia stack. Kurento can be controlled via API it exposes with the help of client implementations written for several programming languages. Kurento API has modular structure and relies on two basic concepts:

- *Media Element* - a functional unit performing specific action on a media stream. There are input/output elements responsible for injecting and tak-ing media streams out of pipeline, filters that are in charge of analyzing and modifying data and hubs managing multiple media streams in a pipeline;

- *Media Pipeline* - a graph formed by chains of *Media Elements* where he output stream generated by a source element is fed into one or more sink elements.

Kurento can be used in any type application where the signaling is based on SIP or HTTP and the media is represented and transported in any of the protocols and formats supported by GStreamer [21]. This makes Kurento a notable candidate for building advanced multimedia applications.

2.8 Existing solutions for conference transcription

A conference can be recorded and uploaded to Youtube that will generate transcripts or to some transcription service like Way With Words, but those are not real-time. Speaking about recently developed solutions, transcription feature was added to Skype [5]. But what about open-source solutions? There are not many.

The BaBL Project [22] is a simple conferencing application using Javascript Web Speech API available for Chrome browser for speech recognition. Transcription is performed and displayed separately for each speaker. Unfortunately, the project has not been updated since 2014.

Jigasi [23] is a part of Jitsi stack. It is a SIP gateway that allows regular SIP clients to join Jitsi Meet conferences hosted by Jitsi Videobridge which uses different signalling protocol (XMPP). Jigasi provides transcription capabilities since 2017 [24]. Jigasi can be invited to a Jitsi Meet conference as a silent attendee. It receives audio data from conference participants via RTP and uses Google Cloud Speech-to-text API for transcription. The problem is that one is obliged to use Jitsi stack as this software works only for Jitsi Meet. Also, there is no adequate documentation on this solution and only some inexplicit installation instructions so it is hard to setup and use (actually, attempts to configure Jitsi transcription within this project have failed.) However, this is the most consistent and relevant open-source solution for real-time transcription which can be found today.

3 Solution architecture

3.1 Application requirements

There are four main requirements to the transcribing application implied in the assignment. Fulfilling all these requirements was the main goal while designing the application architecture:

- It must somehow connect to the conference room and capture live stream audio of every participant in the room separately;
- It must support at least one modular transcription back-end. Modularity suggests designing the application in such a way that adding support of a new transcription back-end would not require changing of already existing logic;
- It should not be tied up to any particular conferencing platform so it would be possible to integrate it into various conferencing applications. In the scope of this diploma thesis, it must be integrated with one selected open-source conferencing platform.
- It should be easily deployable and scalable - capable of performing transcription for multiple conferences simultaneously.

3.2 Selecting conferencing platform

As development of conferencing application is not the focus of this work, there are no high or specific requirements to the central unit. It should be easy to deploy and control. Kurento turned out to be the most comfortable selection as it can be easily installed, provides good documentation and complete example applications. Its architecture also allows to easily implement new functionality.

3.2.1 Setting up demo conference room

Kurento provides various examples of how to use the media server for solving different tasks written in Java and Node.js. Among them there is a group call application which is sufficient to organize a demo conferencing room to test the transcription application. This example application is simple and limited but development of a production-ready conferencing solution was not an objective of this thesis, so the example application written in Java was used and modified as far as it was necessary to enable web conference transcription.

The interface allows a user to enter a room name and a nickname to use in the room. If such room already exists, the client will join to that room, otherwise a new room would be created. After entering the room and giving the browser permission to capture his media data, the user can see himself and other participants of the room if there are any as well as sort of a chat box where the transcripts will be displayed. The screenshots of the GUI can be found in the appendix.

3.3 Selecting speech recognition back-end

3.3.1 General requirements for speech recognition service

The main requirement towards a speech recognition service for solving the task of live web conference transcription is support of streaming speech recognition. We will analyse and select a transcription back-end according to the following criterions:

- Streaming speech recognition support;
- It must provide a comprehensible API to integrate with our application;
- Variety of supported languages, Slavic languages being the focus of our solution;
- Continuous speech recognition. As web conferences are usually held for a relatively long time (e.g. from 10 minutes to several hours) it is necessary for transcription back-end to maintain persistent connection with the client. If there are any limitations on the audio stream duration it should be possible to quickly reinitialize recognition.
- Accuracy of speech recognition. There is a difference between transcribing short audio and a long conversation with multiple participants where relatively low word error rate for each of them may accumulate and result in a nonsense final conversation log. However, high accuracy usually comes in the cost of higher latency which is critical for a real-time application so there must be some trade-off between these qualities.
- It should be available without purchasing subscription for a decent price;
- Java client libraries and detailed documentation are desirable but not obligatory;

There is plenty of commercial speech recognition software, both online and offline installations. Many of them are oriented towards enterprise usage, do not offer free trials and not many actually support live speech recognition (record audio and upload the file instead) and provide an API for developers. There are also open-source solutions such as CMUSphinx [25], but they are not considered because of significantly lower quality of speech recognition comparing to commercial solutions. We are not able to analyze all existing speech recognition software and it is not the purpose of this thesis, so we will look at only the most known and widely used solutions, applying the defined criterions.

3.3.2 Google Cloud Speech-to-text API

Google Cloud Speech-to-text API [26] is a well-documented API with client libraries available for many programming languages such as C#, Go, Java, Node.js, PHP, Python and Ruby. It supports a great variety of languages, actually, most of the languages spoken in the world. Cloud Speech-to-Text provides the following capabilities of transcribing audio:

- Synchronous speech recognition intended for transcription of short audio files (less than 1 minute);
- Asynchronous speech recognition allows transcribing audios longer than 1 minute but they have to be uploaded to Google Cloud Storage first. Recognition time depends on the length of the audio and can take minutes if the audio file is large;
- Streaming speech recognition allows streaming audio to Cloud Speech-to-Text and receiving results in real time. Unfortunately, the length of the audio is limited to 1 minute and if it exceeds this limit, an error will be returned. Reinitializing the recognition every one minute by sending configuration requests again seems to be the only way to overcome this limitation for now. Streaming speech recognition is available via gRPC (gRPC Remote Procedure Calls).

There are some features of Google Cloud Speech-to-Text worth mentioning: separation of different speakers, automatic detection of the spoken language, automatic punctuation, transcribing audio with multiple channels. It is stable against side noises in the audio. There are also special recognition training models such as phone call model (currently available for English only) which might be especially useful as web conferences are close to phone calls. Obviously, this API is not free. Actually, one can transcribe up to 60 minutes of audio for free and then it will cost $0.006 for every 15 seconds. It is important to keep in mind that every request will be rounded to the nearest increment of 15 seconds so, for example, 3 separate requests containing 7 seconds of audio will be billed as 45 seconds of audio.

Google Cloud Speech-to-Text recommends providing streaming audio captured with a sampling rate of 16 kHz or higher, encoded in FLAC or LINEAR16 codec and split into 100-milliseconds frames as a good trade-off between efficiency and latency [26].

3.3.3 IBM Watson Speech to Text

IBM Watson Speech to Text [27] supports far less languages comparing to Google Cloud Speech-to-Text: Arabic, English, Spanish, French, Brazilian Portuguese, Japanese, Korean, German, and Mandarin. There are many available SDKs: Android, Java, Node.js, Python, Ruby, JavaScript library, .NET etc. The service offers three speech recognition interfaces:

- Synchronous HTTP interface;
- Asynchronous HTTP interface;
- WebSocket interface. According to documentation, it is the preferred mechanism for speech recognition as it has a number of advantages over the HTTP interface such as full-duplex communication channel, establishing a single authenticated connection indefinitely (HTTP interfaces require to authenticate each call), reduced latency and network utilization and an event-driven model of communication;

The WebSocket and synchronous HTTP interfaces accept a maximum of 100 MB of audio data with a single request. Up to 1 GB of audio data can be send with a single asynchronous request. The WebSocket interface looks like an applicable option for real-time speech recognition. This recognition service is suitable for high-noise environments. IBM charge $0.02 per minute based on the actual length of audio sent.

3.3.4 Microsoft Speech-to-text

Microsoft Speech-to-text [28] is one of the Azure speech services previously available as Bing Speech API. The Bing Speech API is still functional but will stop working from 15.10.2019 so it is not considered in this thesis. This API supports more languages than IBM Watson but still less than Google Speech-to-text. There are SDKs available for C/C++, C#, Java, JavaScript/Node.js, Objective-C, Python. There are the following usage cases described in the documentation:

- Transcription of an audio recorded with a microphone;
- Speech recognition from an input file;
- Audio Input Stream API provides a way to recognize audio streams instead of microphone recordings or input files. The only audio format currently supported is PCM, single channel, sampled at 16 kHz, 16 bits per sample. However, the documentation does not provide a clear and detailed code sample of using this API.

The pricing looks more or less attractive. In case of one concurrent request at a time Speech-to-text services can be used for 5 hours free per month. Up to 20 concurrent requests will cost $1 per hour. Usage is billed in one-second increments.

3.3.5 Amazon Transcribe

Amazon Transcribe [29] is one of the machine learning services provided by Amazon. It supports transcription of streaming audio in real-time using HTTP/2 streams: client send a stream of audio and Amazon transcribe returns a stream of JSON objects containing the transcript. Unfortunately, streaming recognition is supported only for English and Spanish languages.

3.3.6 Yandex SpeechKit

Yandex SpeechKit [30] supports Russian, English and Turkish languages and provides streaming speech recognition via gRPC . Acceptable audio formats are LINEAR16 and OPUS. The maximum duration of transmitted audio for a single session is 5 minutes. To continue recognition, it is necessary to reconnect and send a new message with speech recognition settings. So, while being the cheapest among mentioned services, Yandex SpeechKit is relatively limited.

3.3.7 Speech recognition software developed in TUL

In the scope of this diploma thesis there also was an opportunity to try out cloud transcription platform based on the speech recognition engine developed by SpeechLab in the walls of Technical University of Liberec [1] which will be later referred to as TUL SpeechLab SRE (Speech Recognition Engine). It supports 18 languages (most of them are Slavic languages), provides streaming speech recognition capabilities using gRPC, event-driven model of client-server communication model and timestamps which is extremely useful not only for indexing but also for time synchronization between multiple audio streams being recognized simultaneouosly like in case of transcribing a web conference. The platform provides three APIs:

- HTTP File API to transcribe pre-recorded audio files;
- WebSocket API for browser based applications;
- gRPC API for non-web applications with fast response time requirements.

All three APIs support real-time speech recognition, which is important as it means that this platform was designed specially for real-time solutions.

3.3.8 Final selection of transcription back-ends

The solution from SpeechLab was selected as primary recognition back-end for its real-time intended features. As we can see, other suitable speech recognition services are provided mainly by huge IT companies as part of their various machine learning cloud solutions for business and development. They use different communication technologies to provide live speech recognition functionality: gRPC, WebSockets, HTTP/2 streams. According to some benchmarks and comparisons Google Speech-to-text tends to perform with a generally lower WER (word error rate). IBM

Watson, Yandex SpeechKit and Amazon Transcribe support a narrow variety of languages comparing to Microsoft Speech-to-text and Google Cloud Speech-to-text. The main disadvantage of Google Cloud Speech-to-text is the short accepted length of the audio stream which can probably be tricked and Microsoft Speech-to-text Audio Input Stream API is limited to a particular audio format which may introduce additional complications and provides relatively scant documentation. Any of the services can be better or worse depending on different conditions so modularity of the transcription back-end in our application is required for it to be flexible and adjustable for slightly different usage cases. For this reason within this diploma thesis several transcription back-ends were selected to compare results.

An important note must be made. There are no examples of streaming speech recognition performed for multiple audio streams simultaneously with synchronization of results provided in documentation of any of the mentioned transcription services. It means that unexpected difficulties may be encountered while using any speech recognition back-end. To find out whether selected speech recognition back-end is truly suitable for real-time web conference transcription and define general requirements is one of practical goals of this thesis.

Google Cloud Speech-to-text API has been chosen as the second possible transcription back-end for the application developed within this diploma thesis as the most widely-used and well-proven service. It supports a vast amount of languages and audio formats which makes it flexible, offers high accuracy of recognition, automatic punctuation and other potentionally useful features. Although there is a significant limitation to the audio stream duration, we will search for a solution of this problem which must definitely exist as many applications use this API – the mentioned solution from Jitsi foundation is not an exception.

3.4 Solution structure

The whole system consists of three main elements, as it is shown on figure 3.1: the conferencing application, the media server and the transcribing application.

Conferencing and transcribing applications use WebSocket connection to exchange information about ongoing conferences and transfer transcripts. Conferencing application uses Kurento Client library to control Kurento Media Server which handles the flow of media in a conference and streams each participant's audio data to the transcribing application which extracts encoded audio data from RTP packets and sends it to a selected speech recognition service. Results are returned to the conferencing application to be displayed in browser clients and also saved to a file.

Figure 3.2 represents main transcribing application components and their interaction.

The following set of classes is responsible for primary application logic:

- *Main* - the application starts in this class. It is responsible for WebSocket connection, message handling and controlling the active transcribed conferences;

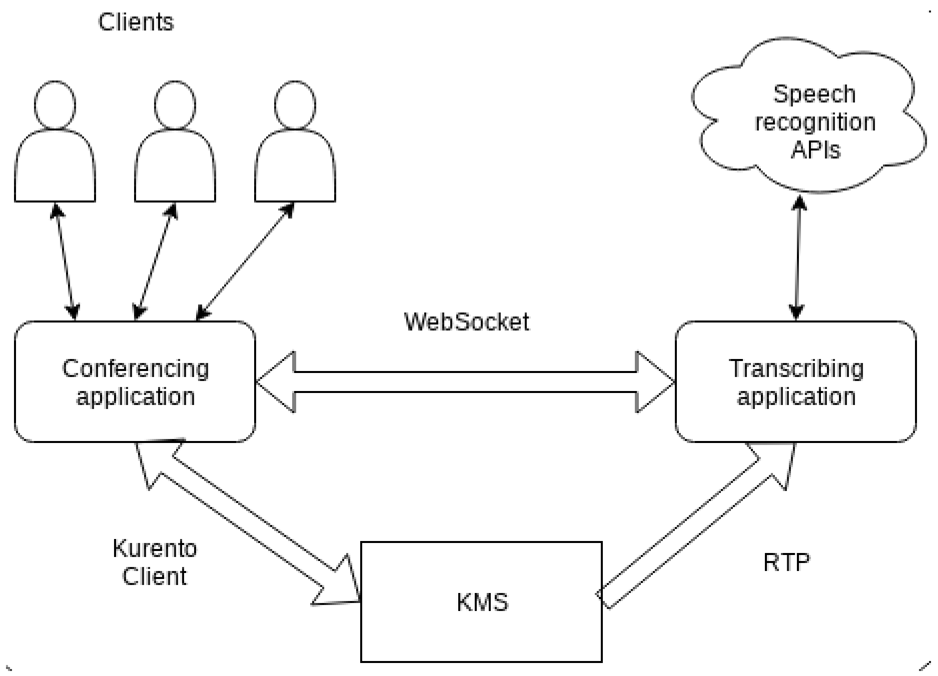


Figure 3.1: Scheme of the interaction between solution components.

- *Configuration* - a service class written as Singleton, stores configuration parameters loaded from a properties file provided on application start;
- *Conference* - all information about a conference is stored in an instance of this class. It stores data about participants, RTP receivers, transcribers and transcript processors working for a conference and controls creation and deletion of these objects;
- *RTPReceiver* - this class handles incoming RTP streams, discovering different participants in the incoming flow of RTP packets, extracting raw audio data and passing it to transcribers, starting new transcribers for each new source stream found. There is one RTPReceiver for every conference and it is running in a separate thread;
- *Transcriber* is an abstract class which must be inherited when adding transcription back-ends. Any transcriber has an id, a link to the conference it belongs to, a source stream and start/stop flags. Inherited classes must implement abstract methods *initialize()*, *startTranscription()*, *transribe()* and *stopTranscription*. Also, they must implement method *run()* from the interface Runnable as they are designed to run in a separate thread. Initialization suggests setting up connection and authentication parameters. Starting transcription is sending the start message to the transcription back-end. Method *transcribe()* should be called to send a chunk of data read from the source stream. Stopping transcription suggest closing connection to the speech recognition API;

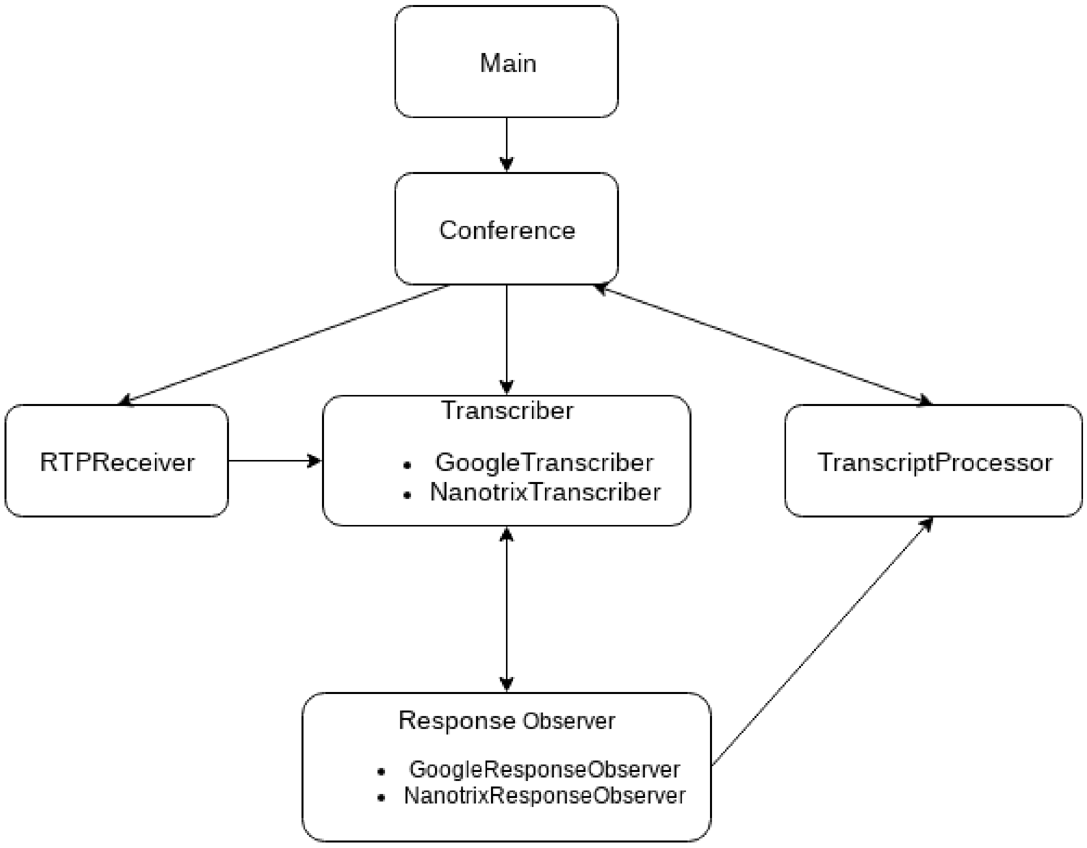


Figure 3.2: Application classes interaction.

- Response observers represented by *GoogleResponseObserver* and *NanotrixResponseObserver* handle responses from SREs, passing transcripts to transcript processing logic and control transcribers if necessary;

- *TranscriptProcessor* - transcripts received from the speech recognition back-end are processed here: sorted in correct order, serialized to JSON to be delivered back to conferencing application and saved to file for persistent storage. There is one TranscriptProcessor for a conference, running in a separate thread;

- *Transcript* - objects of this class contain pieces of transcribed speech and all necessary metadata: conference name, participant name, timestamp;

- *ObjectFactory* - a factory class providing static methods to create different instances of *Transcriber* and *TranscriptProcessor* depending on transcription back-end used in current instance of application.

3.4.1 Communication

The transcribing application uses WebSocket to communicate with conferencing application. As browser clients usually speak to conferencing back-end via WebSocket as well, it is relatively easy to integrate transcribing solution which acts like a client. Communication is performed by mutual exchange of JSON messages containing information about new and leaving participants, RTP session parameters and transcriptions of speech. Message type is specified in *id* field of a JSON message.

Types of messages sent by transcribing application:

- *transcriberRegister* - this message helps conferencing application to identify transcriber among other clients connecting to it via WebSocket. It is up to developer to implement acceptance/rejection logic but this message must be replied with *transcriberRegisterResponse*;
- *transcriberSdpOffer* - generated SDP offer. Such message also contains *conferenceId* and *participantId* fields identifying room and participant whose audio stream the transcription application wishes to obtain. The SDP offer must be processed by the part of conferencing application responsible for RTP streaming;
- *transcriptionStarted* - this message is used to notify the conferencing application that transcription has started for a particular user as it contains *conferenceId* and *participantId* fields. The conferencing application may then broadcast such message to the conference participants to let them know that transcription is active;
- *transcript* - such message contains a *transcript* of a piece of speech spoken by a person identified by *conferenceId* and *participantId* fields;
- *transcriptUrl* - contains a link to transcript served by HTTP server.

Types of messages handled by transcribing application:

- *transcriberRegisterResponse* - a response to *transcriberRegister* request. It must contain *response* field the value of which is *rejected* in case the conferencing application can not accept the transcriber for some reason (e.g. there is already registered transcribing application and only one is supposed by application logic). The rejection response must also containg *message* field specifying the reason of rejection;
- *newParticipant* - should be sent by conferencing application whenever a new conference is created or there is a new participant in an existing conference. It must contain *conferenceId*, *participantId* and *languageCode* fields referring to the room and participant for which this event has occured. If it is a new conference, the transcribing application will create new *Conference*, *RTPReceiver*, *TranscriptProcessor* objects and then generate an SDP offer. If it is a new

participant in an existing conference, only SDP negotiation will be performed to acquire new participant's RTP stream;

- *transcriberSdpAnswer* - an answer to the SDP offer sent by transcribing application. It must contain *conferenceId* and *participantId* fields. Participants are added to the conference description of the transcribing application only if SDP negotiation was successful;
- *quitParticipant* - contents are the same as in *newParticipant* message but should be sent if a user leaves conference room. The entities responsible for transcription for this user will be removed. If it was the last user in the conference (the call is finished) then RTP receiving and transcript processing will be stopped and the conference will be closed.

The following chapters provide detailed description of all stages of performing real-time conference transcription.

4 Capturing live audio streams of conference attendees

4.1 Possible approaches

The simplest way to capture audio is doing it on client side by getting direct access to the user's media device. It would limit the selection of transcription back-end as streaming speech recognition is not always available with JavaScript. Also, it would oblige to use this particular WebRTC client for someone who would like to use the solution.

Transcription must be performed by some external application to be flexible. This application can act as a silent participant. Such approach is used in Jigasi, where transcriber is a participant which can be invited to the conference by pressing special button. Transcriber is a SIP client.

The idea implemented is this work is generally similar but evades usage of SIP and necessary SIP servers and gateways, so there is no need to integrate SIP with WebRTC specially for transcription application. RTP streams are forwarded to the transcribing application by the media server and signalling is done in a custom way over WebSocket using SDP for session negotiation. From the perspective of the conferencing application clients, the transcription is performed by back-end, so there is no physical presence of some transcriber in the conference as a participant.

4.2 Streaming and receiving with RTP

There are not many implementations of RTP stack available for Java. *libjitsi* developed by Jitsi for their conferencing stack is the most advanced and relevant according to its description but unfortunately no success was achieved in attempts to use it in this project. Java Media Framework (JMF) [31] was used for quite a long time but it is extremely outdated (no updates since 2003) and therefore does not support modern audio formats and SRTP (secure version of RTP). Finally, the project was reconfigured for *jlibrtp* - a simple open-source library. It it is not tied up to any audio format like JMF which is definitely an advantage and is simple to integrate. However, it does not support SRTP.

4.2.1 Configuring Kurento to stream RTP

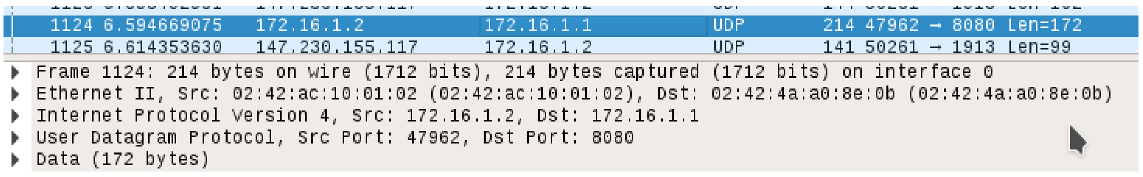
Kurento's pipeline concept and media server capabilities give an advantage in configuration of RTP streaming. Every participant registered in the application has an outgoing *WebRTCEndpoint* and a number of incoming endpoints equal to the number of participants in the conference, which is varying as users join and leave conference rooms. All outgoing media can be duplicated in a RTP stream by creating a *RTPEndpoint* and connecting it to the outgoing *WebRTCEndpoint* when initializing user session. Kurento will start RTP stream when it receives session description parameters. Kurento implements SDP offer/answer negotiation model for its *RTPEndpoint*. The stream starts as soon as the conferencing application receives an SDP offer from the transcribing application via WebSocket and passes it to the *RTPEndpoint* to process.

4.2.2 Audio format

WebRTC uses OPUS as default audio codec. Unfortunately, TUL SRE does not support this codec and Google Speech API supports only Ogg [32] containerized OPUS audio so it would be better to use another codec supported by both speech recognition back-ends. G.711, also known as Pulse Code Modulation (PCM), is a very commonly used waveform codec, primarily in telephony. There are two slightly different versions: µ-law, which is used primarily in North America and Japan, and A-law, which is in use in most other countries outside North America [33]. µ-law encoded PCM was selected for this diploma thesis, as it is the only codec supported by both engines.

4.2.3 Depacketizing RTP stream

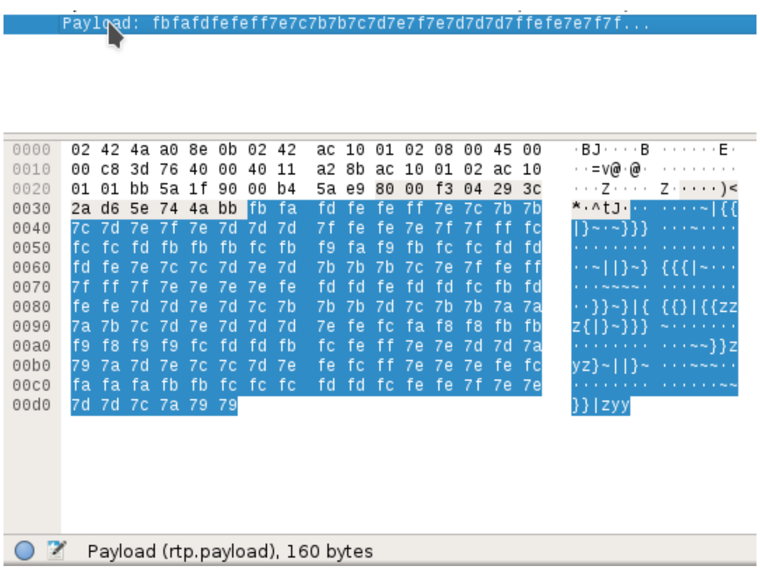
Before extracting raw audio data from the RTP frames we should analyze the incoming stream to make sure it works correctly and to figure out the correct way of extracting audio data. Wireshark is a widely used network protocol analyzer and it suits for this task just fine. At first the RTP packets can be seen coming from Kurento Media server to the transcription application listening for incoming RTP stream on port 8080 which is illustrated on figure 4.1 However, these packets are



```
1124 6.594669075   172.16.1.2        172.16.1.1        UDP        214 47962 → 8080 Len=172
1125 6.614353630   147.230.155.117   172.16.1.2        UDP        141 50261 → 1913 Len=99
▸ Frame 1124: 214 bytes on wire (1712 bits), 214 bytes captured (1712 bits) on interface 0
▸ Ethernet II, Src: 02:42:ac:10:01:02 (02:42:ac:10:01:02), Dst: 02:42:4a:a0:8e:0b (02:42:4a:a0:8e:0b)
▸ Internet Protocol Version 4, Src: 172.16.1.2, Dst: 172.16.1.1
▸ User Datagram Protocol, Src Port: 47962, Dst Port: 8080
▸ Data (172 bytes)
```

Figure 4.1: The flow of the packets between source and destination hosts.

considered as simple UDP frames. This is not a problem as Wireshark can be forced to decode those frames as RTP packets. The audio data is expected to be encoded with µ-law algorithm, sampled at 8000 Hz, 8 bits in each sample, mono. Now it can be clearly seen on figure 4.2 that RTP payload contains 160 bytes of PCMU audio, exactly 20 8-bit samples.



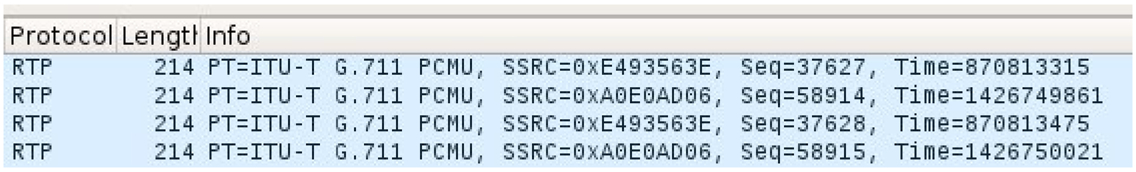
```
Payload: fbfafdfefeff7e7c7b7b7c7d7e7f7e7d7d7d7ffefe7e7f7f...

0000  02 42 4a a0 8e 0b 02 42  ac 10 01 02 08 00 45 00   ·BJ····B ······E·
0010  00 c8 3d 76 40 00 40 11  a2 8b ac 10 01 02 ac 10   ··=v@·@· ········
0020  01 01 bb 5a 1f 90 00 b4  5a e9 80 00 f3 04 29 3c   ···Z···· Z·····)<
0030  2a d6 5e 74 4a bb fb fa  fd fe fe ff 7e 7c 7b 7b   *·^tJ··· ····~|{{
0040  7c 7d 7e 7f 7e 7d 7d 7d  7f fe fe 7e 7f 7f ff fc   |}~·~}}} ···~····
0050  fc fc fd fb fb fb fc fb  f9 fa f9 fb fc fc fd fd   ········ ········
0060  fd fe 7e 7c 7c 7d 7e 7d  7b 7b 7b 7c 7e 7f fe ff   ··~||}~} {{{|~···
0070  7f ff 7f 7e 7e 7e 7e fe  fd fd fe fd fd fc fb fd   ···~~~~· ········
0080  fe fe 7d 7d 7e 7d 7c 7b  7b 7b 7d 7c 7b 7b 7a 7a   ··}}~}|{ {{}|{{zz
0090  7a 7b 7c 7d 7e 7d 7d 7d  7e fe fc fa f8 f8 fb fb   z{|}~}}} ~·······
00a0  f9 f8 f9 f9 fc fd fd fb  fc fe ff 7e 7e 7d 7d 7a   ········ ···~~}}z
00b0  79 7a 7d 7e 7c 7c 7d 7e  fe fc ff 7e 7e 7e fe fc   yz}~||}~ ···~~~··
00c0  fa fa fa fb fb fc fc fc  fd fd fc fe fe 7f 7e 7e   ········ ·······~~
00d0  7d 7d 7c 7a 79 79                                  }}|zyy

Payload (rtp.payload), 160 bytes
```

Figure 4.2: RTP payload.

In the RTPReceiver class of our Java application receiving thread calls method *receive()* passing RTP packet as a parameter and the raw audio data can be accessed by calling method *getPayload()* which returns a byte array.

4.2.4 Discovering different source streams in incoming RTP packets and processing the data

When where are two or more participants in the conference we should expect Kurento to start multiple RTP streams. They can be distinguished by different values of SSRC field. An example is provided on figure 4.3



```
Protocol Length Info
RTP        214 PT=ITU-T G.711 PCMU, SSRC=0xE493563E, Seq=37627, Time=870813315
RTP        214 PT=ITU-T G.711 PCMU, SSRC=0xA0E0AD06, Seq=58914, Time=1426749861
RTP        214 PT=ITU-T G.711 PCMU, SSRC=0xE493563E, Seq=37628, Time=870813475
RTP        214 PT=ITU-T G.711 PCMU, SSRC=0xA0E0AD06, Seq=58915, Time=1426750021
```

Figure 4.3: Frames of two RTP streams observed in Wireshark.

Each participant's data needs to be processed individually by an instance of *Transcriber*. An intermediate buffer is required to store raw data - *RTPReceiver* will be writing to this buffer and *Transcriber* will be reading. Queues are data structures used for such purposes in this solution. If received frame belongs to a participant with previously unknown SSRC, a new queue is created and a transcriber is started. The wait-notify mechanism is used for synchronization. The queue acts as a monitor. When a new frame is received, *RTPReceiver* enters the monitor, adds chunk of data to queue and notifies transcriber about new data:

```java
public void receiveData(RtpPkt frame, Participant participant) {
  long ssrc = participant.getSSRC();
  byte[] data = frame.getPayload();
  if (sourceStreams.containsKey(ssrc)) {
    LinkedList sourceStream = sourceStreams.get(ssrc);
    synchronized (sourceStream) {
        sourceStream.add(data);
        sourceStream.notify();
    }
  }
  else {
    final LinkedList<byte []> participantStream = new LinkedList
       <>();
    sourceStreams.put(ssrc, participantStream);
    startTranscriber(participantStream, ssrc);
    logger.info("New participant: {}", ssrc);
  }
}
```

On the other side, *Transcriber* is waiting, being blocked on the same monitor. Transcriber thread wakes when *RTPReceiver* calls *notify* on the monitor and sends data to the speech recognition backend:

```java
while(!isFinished()) {
  synchronized (sourceStream) {
    try {
      sourceStream.wait();
      data = sourceStream.poll();
    }
    catch (InterruptedException e) {
      logger.debug("Thread interrupted");
    }
  }
  if ( data == null ) continue;
  System.arraycopy(data, 0, audio, offset, data.length);
  transcribe(audio);
}
```

5 Setting up transcription back-ends

The process of writing the client logic for both selected transcription services is described in this chapter. It ends with both services configured with default streaming speech recognition settings.

5.1 Writing a client for TUL SpeechLab SRE

5.1.1 gRPC

gRPC (gRPC Remote Procedure Calls) [34] is an open source remote procedure call (RPC) system initially developed at Google. In gRPC a client application can directly call methods on a server application which is deployed on a different machine just like local methods. Like it is usually done in RPC systems, gRPC uses service definition, specifying the methods that can be colled remotely. These methods must be implemented on the server side. Client has a stub that provides the same methods as the server. The interfaces are generated by a special compiler from service description for chosen programming languages. The main features of this RPC system are:

- Authentication;
- Bidirectional streaming with flow control;
- Synchronous and asynchronous method execution.

gRPC uses protocol buffers for service descriptions. gRPC services can be compiled and run in various environments. All these makes them extremely useful for building microservice style infrastructures.

Google Cloud Speech-to-text provides client SDK for Java built on top of gRPC so we will need to generate classes and interfaces only for TUL SpeechLab SRE.

5.1.2 Protocol buffers

Protocol buffers are a mechanism of serializing structured data. Google developed protocol buffers to use internally. They are platform and language independent - Google provides a code generator for multiple languages under open-source license. Developers define their services and data structures (called *messages*) in a special protocol buffer definition file (.proto) and compile them with code compiler provided by Google. Generated code is used to implement both server and client logic [35]. An example data structure description is provided as followed:

```
message Dog {
    required string name = 1;
    required int32 age = 2;
    required string owner = 3;
}
```

Creating an object of this type in Java can be done with the following lines of code

```
Dog dog = Dog.newBuilder()
     .setName("Rex")
     .setAge(5)
     .setOwner("Bob")
     .build();
```

5.1.3 Compiling service code with protoc

protoc is a comliler for protocol buffers definition files. For Java, the compiler does not generate the interfaces for communication with the server by default. To compile them, *protoc-gen-java* plugin must be installed first. The pre-compiled plugins available in the repositories may not work (like in our case). The best way to avoid both manual compilation of plugin and manual running *protoc* is to configure Maven to compile the code from *.proto* file by installing Protobuf Maven plugin and configuring it to use *protoc* with *protoc-gen-java*. After that the code can be easily generated by running *mvn compile* target.

5.1.4 Connection and authentication

Connection to the speech recognition engine is performed in several steps:

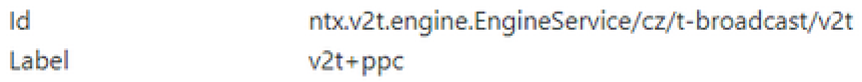
1. First, a Managed Channel is built to connect to the server. A custom thread executor with fixed thread pool is provided to handle responses. Replacing default executor is strictly recommended by gRPC documentation as it uses cached thread pool which behaves badly under heavy load spawning new threads while the rest are busy;

2. After that an asynchronous client stub can be created on the channel;

3. Next, an access token is obtained by performing HTTP POST request to the API with credentials provided;

4. Using the access token another request is sent to obtain task specific token. Each task is identified by unique label and id.

   As it can be seen in the figure 5.1, an example task definition configures transcription and text post-processing will be performed for Czech language.



Id	ntx.v2t.engine.EngineService/cz/t-broadcast/v2t
Label	v2t+ppc

Figure 5.1: An example id and label.

5. Task token is put into metadata together with additional *no-flow-control* flag set to true as disabling flow control is recommended for environments with varying level of latency and throughput.

6. Finally, a Response Observer and created and we use the stub to call the only available *streamingRecognize()* method on Response Observer to obtain Request Observer.

The Voice-to-text API for TUL SpeechLab SRE is implemented as bidirectional flow of *EngineStream* messages. The process starts by sending and receiving start message followed by bidirectional data transfer. The data transmitting session is terminated by sending and receiving end message [36]. Request Observer will be used to send control messages and audio data to the server and Response Observer will handle receiving of server responses.

5.1.5 Starting data transmission

The first message sent within a session must define *EngineContext* containing recognition settings (whether to apply post-proccesing, automatic punctuation, etc), specifications of audio format (codec, sampling frequency, channel layout). The server responds with a start message as well and browser clients are notified that transcription has started.

5.1.6 Handling server responses

Basically, there are two types of event contents being pushed by the server, that need to be taken care of: labels and timestamps. Details on timestamps are provided in section 5.1.8. Useful labels can be either *items* or *pluses*, the former being the words recognized by the engine and the latter are delimiters (e.g. whitespaces). Whenever a useful label is received, an instance of *Transcript* is created and passed to *TranscriptProcessor*.

5.1.7 Terminating the session

When transcriber's *isFinished* flag is set to true meaning participant has left and there is no more audio data coming, a special end message is sent and the client stream must be closed. However, the server still might be pushing some data followed by an end message. After an end message is received, server will not be pushing any more events.

5.1.8 Timestamps

TUL SRE provide speech recognition results along with timestamps measured in 100ns ticks. This is default behavior. These timestamps represent relative amount of time passed since the beginning of audio. In our case, it would the time when the first RTP packet is received for a participant. Knowing this absolute moment of time we can use timestamps to determine the moment of time when a piece of speech was spoken. This time will obviously be different from the moment when the transcript is received. An example extraction from application logs illustrating this difference is provided in figure 5.2:

```
Received item 'jak' at 2019-04-23 16:30:59.041
Transcript: 'jak', timestamp=50600000, spoken at 2019-04-23 16:30:54.426
```

Figure 5.2: Difference between speaking and receiving time.

For every word the time when it was spoken is computed as a sum of the start time and the timestamp a transcript belongs to. *System.currentTimeMillis()* method is called when the first RTP packet is received to acquire the most accurate time elapsed in milliseconds since the epoch. There is also *System.nanoTime()* method but it can be used only to acquire precise time relative to some arbitrary point. Thus, timestamps returned by SRE must be cast to milliseconds and it is done by division by 10000L.

5.2 Writing a client for Google Cloud Speech-to-text API

Generally, communication with Google Speech API performed in the same way as with TUL SRE. The Java client library allows to configure everything a little simpler. Also, this process is described well in the API documentation [26] so only brief description is mentioned here. To use API, one must first set up a project in Google Cloud Panel, enable Speech-to-text API for it, create a service account and, finally, download a JSON file containing the private key. After that speech recognition can be set up in the following steps:

1. Load the previously downloaded credentials;
2. Create a *SpeechClient* using provided settings;
3. Create a response observer. A custom response observer class was written for this speech recognition service as well.
4. Obtain the *ClientStream* by calling *splitCall()* on the response observer;
5. Send the start message. After that the application may start sending audio data.

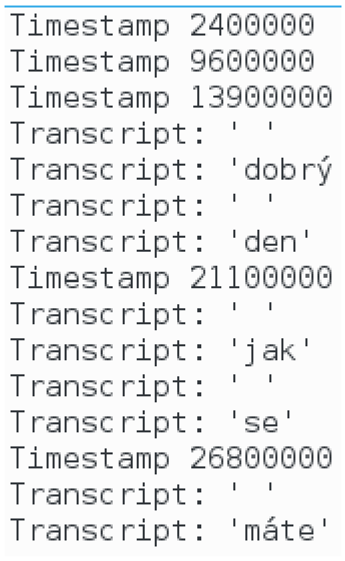
Google Speech API returns results as a list of *alternatives*. Unlike TUL SRE, there are usually complete sentences instead of individual words. The first alternative should be selected as it is likely the most accurate. *GoogleResponseObserver* is handling results and also controls *GoogleTranscriber* reinitialization when it is required.

6 Processing transcripts

6.1 Formulation of the problem

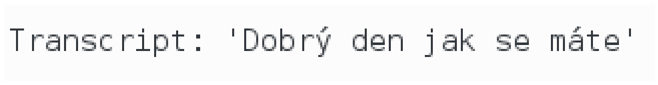
6.1.1 Differences in how transcription services return results

While TUL SpeechLab SRE API always uses real-time model of communication returning recognized text little by little as incoming audio data is processed, Google Cloud Speech-to-text API relies on detecting the silence in speech to determine when the speaker stops talking and sending optional intermediate results. The difference of these approaches is demonstrated on figures 6.1 and 6.2.



```
Timestamp 2400000
Timestamp 9600000
Timestamp 13900000
Transcript: ' '
Transcript: 'dobrý
Transcript: ' '
Transcript: 'den'
Timestamp 21100000
Transcript: ' '
Transcript: 'jak'
Transcript: ' '
Transcript: 'se'
Timestamp 26800000
Transcript: ' '
Transcript: 'máte'
```

Figure 6.1: Example transcription by TUL SpeechLab SRE API.



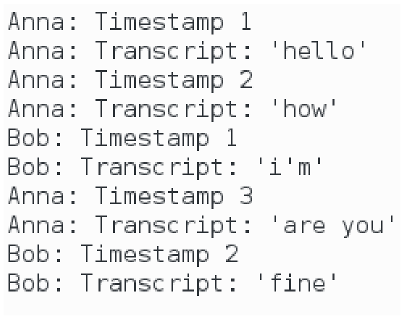
```
Transcript: 'Dobrý den jak se máte'
```

Figure 6.2: Example transcription by Google Speech API

Google Speech API works in such way that if you speak continuously you will probably receive results only after making a long pause. It is not recommended to speak continuously for a long time before a solution for 1-minute audio length restriction is found.

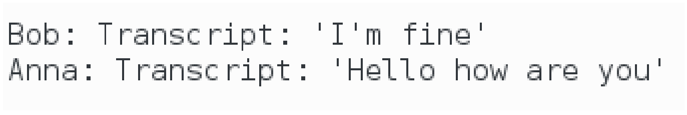
6.1.2 Problem of synchronization

Those differences do not affect anything as long as there is only one participant in a conference room. But when there is an ongoing conversation between several attendees the order of words in the transcribed text might be different from how they were actually spoken. For example, if one person asks *”Hello, how are you?”* and another replies *”I’m fine”*, it may result in the responses illustrated on figures 6.3a and 6.3b.



```
Anna: Timestamp 1
Anna: Transcript: 'hello'
Anna: Timestamp 2
Anna: Transcript: 'how'
Bob: Timestamp 1
Bob: Transcript: 'i'm'
Anna: Timestamp 3
Anna: Transcript: 'are you'
Bob: Timestamp 2
Bob: Transcript: 'fine'
```

(a) Responses from TUL SRE.



```
Bob: Transcript: 'I'm fine'
Anna: Transcript: 'Hello how are you'
```

(b) Responses from Google Speech API.

Figure 6.3: Examples of wrong transcript order.

The reasons for such behaviour may vary: network latency, delays on client side and peculiar properties of behaviour of the speech recognition engines, the latter being the primary one. Another unpleasant fact was discovered when performing transcription for languages other than English - Czech and Russian, to be precise: even if a speaker remained silent for a long time after speaking a short phrase, the service could respond in more that 30 seconds, usually closer to the duration limit. This identifies that default streaming speech recognition settings are not applicable for actual real-time recognition.

6.1.3 Solution for TUL SpeechLab SRE

Timestamps are the key to solving the problem of transcript synchronization for this engine as they help to determine the moment of time when a piece of speech was spoken independently of when the recognition result was actually received. Transcripts can then be sorted by time. Three methods of restoring the correct order of transcripts were suggested for this SRE:

1. Buffer all transcripts for a conference, sort them when it ends and then write to a file.

   Pros:

   - Easy to implement;
   - The order of transcripts will always be correct.

   Cons:

   - Some conversations may take a very long time. Therefore, storing many transcripts for such conversations in memory and sorting large amounts of data may turn out resource-demanding;
   - This method does not allow to deliver transcripts back to browser clients in correct order in real-time.

2. Buffer transcripts for some time, restore the correct order, then write to file and send to browsers. Basically, it would work like a jitter buffer.

   Pros:

   - Browser clients can view conversation log close to real-time;
   - Sorted collections may be used to sort the data as it arrives.

   Cons:

   - Additional delay;
   - There is no warranty that more late transcripts will not arrive after the contents of the buffer are processed, so we will still experience incorrect order at the border between two buffers.

3. Write messages for each participant in a separate collection, process transcripts only when where is at least one element in each collection. Even if there are no transcripts (the person is not speaking), timestamps are still coming. On each step we would take one element from each collection, compare them and process the oldest element if it is a transcript.

   Pros:

   - The order of transcripts will always be correct;
   - Real-time processing. Delays are possible but only for a few seconds.

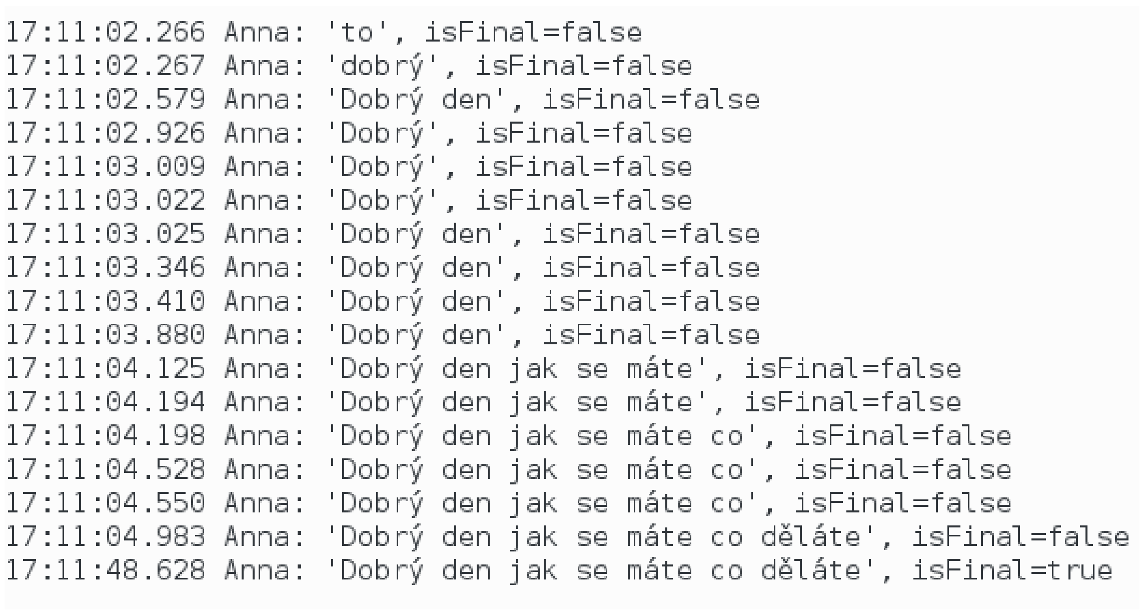
   Cons:

   - Slightly more complicated logic then in other variants.

The third method was chosen as it allows real-time processing and does not carry disadvantages present in other methods.

6.1.4 Solution for Google Cloud Speech-to-text API

First problem that must be solved is a relatively long time gap between last spoken word and server response. Actually, Google Speech API responds fast but not all responses have *isFinal* flag set to true. Sending of intermediate results can be enabled to receive hypothesises. These are engine's current guesses on what was said and final result might be different. An example transcript with intermediate results enabled is provided on figure 6.4. Although nothing new was said after 17:11:04, the server did not respond with final result until 17:11:48. The intermediate results



```
17:11:02.266 Anna: 'to', isFinal=false
17:11:02.267 Anna: 'dobrý', isFinal=false
17:11:02.579 Anna: 'Dobrý den', isFinal=false
17:11:02.926 Anna: 'Dobrý', isFinal=false
17:11:03.009 Anna: 'Dobrý', isFinal=false
17:11:03.022 Anna: 'Dobrý', isFinal=false
17:11:03.025 Anna: 'Dobrý den', isFinal=false
17:11:03.346 Anna: 'Dobrý den', isFinal=false
17:11:03.410 Anna: 'Dobrý den', isFinal=false
17:11:03.880 Anna: 'Dobrý den', isFinal=false
17:11:04.125 Anna: 'Dobrý den jak se máte', isFinal=false
17:11:04.194 Anna: 'Dobrý den jak se máte', isFinal=false
17:11:04.198 Anna: 'Dobrý den jak se máte co', isFinal=false
17:11:04.528 Anna: 'Dobrý den jak se máte co', isFinal=false
17:11:04.550 Anna: 'Dobrý den jak se máte co', isFinal=false
17:11:04.983 Anna: 'Dobrý den jak se máte co děláte', isFinal=false
17:11:48.628 Anna: 'Dobrý den jak se máte co děláte', isFinal=true
```

Figure 6.4: Example transcript with *interimResults* option enabled.

are provided for displaying them to the user while he or she is still speaking. This might be good enough in some cases, but we do not want the conference room to be over flooded with sometimes nonsense messages. Also, intermediate results can not be written to file which means that even if we choose to display intermediate results in browsers writing only final ones to a file at the same time, transcripts would still probably be in the wrong order.

Google Speech API response contains a *confidence* field in range between 0 and 1.0. Intermediate results could be used if their confidence is above some threshold value but, unfortunately, the server sets confidence of all non-final results to 0.

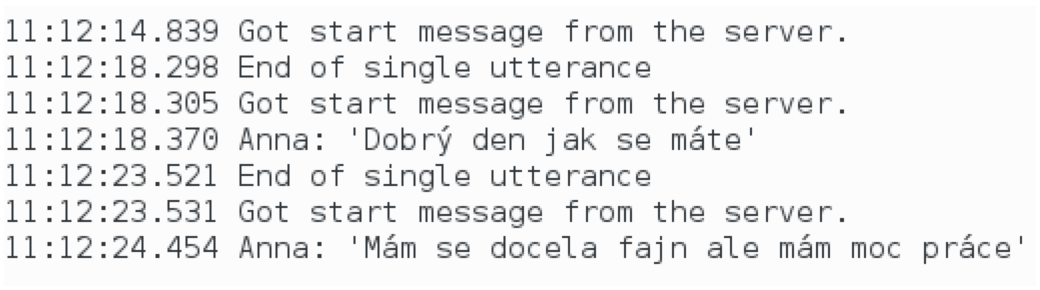
Google Cloud Speech-to-text can include word-level timestamps in response. Time offset values show the beginning and end in each spoken word in a recording. They represent time elapsed from the beginning of audio in increments of 100ms which is close to what is provided by TUL SRE but there is no way to force the server to send data in portions accompanied with time offsets. These timestamps are provided for analyzing long audio recordings where there may be a need to search for a particular word in the recognized text and locate it in the original audio.

Another important problem is the audio duration limit. A solution must be found, otherwise, even a logically correct transcription will hardly be of any use since it can not be used for conversations held longer than one minute.

When audio duration limit is violated, the *onError* method is called in the response observer and an exception with corresponding message is thrown. The simplest solution would be introducing a special flag to notify transcriber that recognition has failed and needs to be reinitialized. This is extremely simple but equally ineffective as significant losses of speech are inevitable.

Two overlapping communication sessions can be used to improve this idea. An initial session is opened when a person joins conference room and keeps opened for almost a minute. Then an additional session is opened, overlapping with the old one for a few seconds to capture speech if the user is in a middle of a sentence. Later, the new session becomes the old one and everything is repeated again. This might work, but causes double transcript and heavy impacts on accuracy.

There is *singleUtterance* configuration option that might solve the delay problem and partially solves the duration problem or, better to say, tricks it. An utterance is the smallest unit of speech. It is a continuous piece of speech beginning and ending with a clear pause. In the case of oral languages, it is generally but not always bounded by silence[37]. If enabled, the recognizer detects a single spoken utterance. If a user pauses for longer than a second or stops speaking, a special event indicating the end of utterance is returned and then server ceases recognition, half-closing the connection. After that a result of recognition may be returned. If nothing is spoken, a single utterance lasts for approximately 7 seconds. Again, we will use a flag to notify the transcriber about end of single utterance and it will reinitialize recognition by reopening client stream and sending initial message with configuration parameters. An example transcription process with such logic implemented is demonstrated on figure 6.5:



```
11:12:14.839 Got start message from the server.
11:12:18.298 End of single utterance
11:12:18.305 Got start message from the server.
11:12:18.370 Anna: 'Dobrý den jak se máte'
11:12:23.521 End of single utterance
11:12:23.531 Got start message from the server.
11:12:24.454 Anna: 'Mám se docela fajn ale mám moc práce'
```

Figure 6.5: An example transcript with *singleUtterance* option enabled.

It takes around 5-50ms to reinitialize recognition and response for an utterance is returned later than next session is initialized. For this reason the same response observer is used for all sessions opened for one participant. However, this solution is far from perfect as practical experiments showed that occasional loss of some parts of the speech happens. Also, this does not procure synchronized flow of transcripts according to what actually spoken - we can only rely on fast service response time which, however, is usually enough. All these leads us towards a conclusion that Google Speech API is not an ideal choice for continuous speech recognition in media conferences.

6.1.5 Input data flow optimisation

RTP packets are 20ms length and arrive every 20ms. They can be locally buffered for some time before sending to the speech recognition service. Buffering too many data for Google Speech API may result in worse speech recognition and occasional loss of the words, so buffering for 200 ms was kept as a trade-off solution. To the opposite, TUL SpeechLab SRE performance degrades with large input data chunks size so no local buffering is done for this SRE.

6.2 Defining a data structure to describe transcripts and their metadata

A special class was defined to store all data related to a single transcript. For every message containing transcribed data an object of this class is created. It is filled with all necessary data to process transcript:

- *id* - this field is a constant, required for serialization and processing messages by the conferencing application;
- *conferenceId* - name of the conference a transcript belongs to;
- *participantId* - name of the participant a transcript belongs to;
- *transcript* - transcribed piece of speech, usually a single word, delimiter or a punctuation mark; Can be null as some Transcript objects carry only timestamp data;
- *timestamp* - a timestamp representing **absolute** moment of time when the piece of speech was spoken;
- *isLast* - a boolean flag which is set to true if response observer received end message from the server. Used for synchronization of threads.

Every Transcript object containing transcript string is serialized to JSON for being delivered back to conferencing application. Therefore, some fields are declared as transient and not serialized.

Transcript class also implements interface Comparable. Transcripts are compared to each other by the value of timestamp in such way that transcript with older timestamp will always be before another transcript with newer timestamp. This logic is required for easy sorting of transcripts to natural order.

6.3 Restoring the logical flow of conversation

6.3.1 Selecting a data structure to store transcripts before processing

First, an appropriate data structure must be selected to store transcripts of each participant's speech. There are three main requirements listed below:

- It must implement FIFO method of processing elements, so the oldest transcripts will be processed first. The order of transcripts within an object of this structure will always be correct as it is impossible to speak to the past;

- The data structure must also be easily accessible both from the beginning and from end: transcripts will be inserted by transcribers to the end and retrieved for processing from the beginning;
- Native concurrency is highly desirable as the data will be accessed and modified by at least two threads: transcriber thread and transcript processing thread.

A double ended queue usually referred to as *Deque* would be the most appropriate collection satisfying these requirements. Java interface *Deque* is implemented by four collections: *ArrayDeque*, *ConcurrentLinkedDeque*, *LinkedBlockingDeque* and *LinkedList*[38]. *ArrayDeque* and *LinkedList* are not concurrent. *LinkedBlockingDeque* is likely the most befitting collection as it natively implements the logic suggested for our solution - if a thread attempts to retrieve an element from an empty queue it will be blocked until an element becomes available. It comes with a capacity bound which defines restrictions for writing to queue - an element can be inserted only if capacity restrictions are not violated. We do not actually want such restrictions and will set maximum capacity to maximum integer value which is very unlikely to be reached by this application.

Queues will be stored in a *HashMap* and retrieved with participant's name provided as a key.

6.3.2 Implementation of the algorithm of transcript sorting

The algorithm loops until there are no more transcripts coming to any queue meaning that the last participant has left the conference, receiving and transcribing audio is ceased and the SRE has returned all possible results. On the first step, it goes through all transcriber queues, retrieves one message from each of them, waiting for messages to come if necessary and stores them in a temporary array. There can actually be none as transcript processing thread may start before any transcriber was created. If there is only one participant in the conference, no sorting is required and the transcript can be immediately processed unless is has *isLast* flag set to true meaning no more transcripts are about to come or it is a service message carrying only time related data.

If there are more than one participant in the conference, transcripts are sorted. The oldest transcript will be the first element of the result array. If it is a last transcript, related transcriber queue is removed.

The remaining transcripts are returned back to their queues by inserting them to the beginning. The temporary array is cleared and next iteration starts. The full algorithm is listed below:

```
while(true) {
  if (transcriberQueues.isEmpty()) continue;
  for (Map.Entry<String, LinkedBlockingDeque<Transcript>> entry
      : transcriberQueues.entrySet()) {
    LinkedBlockingDeque<Transcript> transcriberQueue = entry.
        getValue();
    try {
```

```
                transcriptions.add(transcriberQueue.takeFirst());
            }
            catch (InterruptedException e) {
                logger.debug("'{}': Thread interrupted");
            }
        }
        if (transcriptions.size() == 0) continue;
        if (transcriptions.size() == 1) {
            Transcript transcript = transcriptions.get(0);
            if (transcript.isLast()) {
                // if last transcription was received before exiting loop
                transcriberQueues.get(transcript.getParticipantId()).
                    addFirst(transcript);
                break;
            }
            if (transcript.hasTranscript()) {
                processTranscription(transcript);
            }
        }
        else {
            // Search for the oldest transcription
            Collections.sort(transcriptions);
            Transcript oldestTranscription = transcriptions.get(0);
            // If this is the last message from the response observer,
                just remove the queue
            if (oldestTranscription.isLast()) {
                transcriberQueues.remove(oldestTranscription.
                    getParticipantId());
            }
            // If there is a transcription in the message (may not have
                if it is a timestamp message), process it
            else if (oldestTranscription.hasTranscript()) {
                processTranscription(oldestTranscription);
            }
            transcriptions.remove(0);
            // Return other transcriptions back to related queues
            for (Transcript transcription : transcriptions) {
                transcriberQueues.get(transcription.getParticipantId()).
                    addFirst(transcription);
            }
        }
        transcriptions.clear();
    }
```

6.3.3 Synchronizing response observers and transcript processors

Even if data transmission to the server is terminated there may still be final responses coming. transcript processing thread must process this data before terminating itself. The *isLast* flag was introduced to synchronize response observing and data processing threads. TranscriptProcessor will not terminate until the end message is retrieved from every participant's queue.

6.4 Persistent storage

A complete conversation log of a conference in *.txt* format is available in a unique directory within the root application data directory defined in configuration file. A simple logic was implemented to join separate words into sentences. For the first message we write timestamp and the name of the participant. If next message belongs to the same speaker, it is just appended to the current line. If it belongs to a different speaker, it will be written to a new line along with timestamp and the new speaker's name. For Google Cloud Speech-to-text all messages are written starting with a new line as they usually represent complete sentences. Transcripts are also sent back to the conference room unless the *isFinished* flag is set meaning that the conference room has already been closed and there is no need to deliver transcripts to browsers.

The transcribing application can be configured to start simple HTTP server to serve complete conversation logs. A link would be sent to the conferencing application and can be displayed in the browser clients. However, it is recommended for usage only in test and development environments. It is better to use complete web servers like Nginx or Apache2 in production.

6.5 Delivering transcripts back to browser clients

Every response message containing a transcript string is serialized to JSON and sent back to conferencing application via WebSocket. It is up to developers to decide how to handle them. In this solution a simple delivery model is implemented: transcripts are broadcast across all participants of a conference. A simple JavaScript function handles displaying transcripts on the client side:

```
let speaker = '';
let speakerMessage = '';
let div;
function showTranscript(message) {
  if (speaker!== message.participantId) {
  div = document.createElement('div');
  div.className = 'transcript';
  document.getElementById('transcriptWindow').appendChild(div);
  speakerMessage = message.transcript;
  speaker = message.participantId;
  div.innerText=`${speaker}: ${speakerMessage}`;
  return;
  };
  speakerMessage += message.transcript;
  div.innerText=`${speaker}: ${speakerMessage}`;
}
```

Transcript strings are put into *div* HTML elements and appended as child elements of a parent *div* element representing sort of a chat box.

7 Using the solution

7.1 Deployment

Environment requirements:

- Java (JDK) version 8 or higher;
- Latest version of Maven;
- Latest version of Docker (optional).

The whole solution can be deployed in three steps:

1. Install Kurento Media Server either from package manager or in a Docker container[39]. It is a good practice to run services in Docker containers. Docker is a container-based operating-system-level virtualization system which allows to run processes in isolated environments[40]. Benchmarks ran on Kurento in Docker and on a virtual machines showed that Docker containers have less overhead and demonstrate better performance than virtual machines[41].

2. Lauch demo conferencing room by downloading the source code and executing the following command in directory *kurento-group-call*:

   ```
   $ mvn clean spring-boot:run -Dkms.url=ws://<kurento-host>:<
       kurento-port>/kurento
   ```

3. Transcribing application. After downloading the source code, install the *jrtplib* which is not present in central Maven repository to a local repository and compile the application by running

   ```
   $ mvn install:install-file -Dfile=./lib/jrtplib-0.2.2.jar
   $ mvn clean compile
   ```

   Next, create configuration file. Configuration file contains key-value pair specifying application behaviour. The following configuration options are supported:

   - **websocket.url** - WebSocket endpoint of the conferencing application;
   - **ssl.certificate.nocheck** - if secure WebSocket is used but certificate is self-signed or expired, set to true. Usage in production environment is strictly not recommended;

- **transcription.backend** - speech recognition service for transcriptions. Currently supported values are *google* and *ntx*;
- **transcription.directory** - path to the directory where transcripts will be stored;
- **ip** - the IP address provided here will be used in SDP offer, to bind RTP session and to provide link for transcript download. Usually it is the external IP address of the server but can be different depending on if Kurento is in a Docker container or is on the same machine with transcribing application. HTTP server binds to all available interfaces by default. It is not possible to use localhost if Kurento is in a Docker container.;
- **http.server.enabled** - whether to start simple HTTP server. Not recommended in production environment;
- **http.server.port** - port for HTTP server to bind;
- **ntx.host** - hostname of the machine with TUL SpeechLab SRE installed;
- **ntx.port** - port TUL SpeechLab SRE API listens on;
- **ntx.ssl** - whether to use SSL when connecting to the API;
- **ntx.login** and **ntx.password** - credentials for TUL SpeechLab SRE.
- **google.creds** - JSON file with credentials to Google Speech API. If not specified, application will search for "google.json" file in the class path.

If no configuration file is specified, the application will use *target/classes/application.properties*. Start the application from its root directory by executing the following command:

```
$ mvn exec:java -Dconfig="path-to-config-file" -Dexec.
    mainClass="com.webconferencetranscriber.Main"
```

The conferencing application will be accessible at https://localhost:8443

7.2 Example transcripts

Example transcripts performed by both SREs for a conversation with two participants are provided in the appendix. Generally, Google Speech API performs faster and shows better accuracy, but TUL SRE is more reliable as there is no need to constantly reinitialize the connection risking to lose some data. Also, its accuracy is good for some languages, for example, Czech, but not for English. Speech recognition is very sensitive to the environment (noise), the quality of the microphone used by a speaker and accent. Examples were recorded in a pretty noisy environment using standard laptop microphones.

7.3 Scaling the solution

The application is designed to transcribe multiple conference rooms simultaneously. As the application is designed like a client application, horizontal scaling is possible by connecting multiple instances of the application to the conferencing application. The necessary logic of distributing conferences among transcribers can be implemented in many ways: round-robin, weight coefficients or custom rules. For example, the transcribing applications can be launched with different transcriptions back-ends. Conferences may be transcribed by one or another instance of application depending on the language spoken.

7.4 Further enhancements

There are some features that were not implemented in the scope of this diploma thesis but adding them would increase the variety of use cases and security:

- Adding SRTP support. In this project, all parts of the system were within the same local network or even on one host machine, so pure RTP was acceptable. Even in production environment it would be better to keep the media server and transcribing solution in a local network to reduce latency and possible packet loss. If it is not possible, security standards highly recommend to use secure version of RTP. Leaving RTP traffic unencrypted allows an attacker to sniff it;

- Continuing with the security measures, some kind of authorization in conferencing application must be implemented for transcribing application, for example, with JWT (JSON Web Token)[42].

- Translation. Transcribed text can be translated into desired language using some third-party services. This would allow people who speak different languages communicate and understand each over;

- It is also possible to save transcripts to JSON and store them not in files but in a document-oriented database such as MongoDB, CouchDB or ElasticSearch, the latter being a full-text search engine which can be very useful for indexing and searching transcripts.

8 Conclusion

An analysis on technologies used in modern multimedia conferencing and existing solutions for real-time conference transcription showed that solution of such task is actual and demanded. Community lacks descriptions of experience in solving this task, articles and source codes, so the task was decomposed into smaller parts which were solved step by step within this diploma thesis and joined together.

As a result of conducted research and programming work there was developed an application capable of capturing audio streams, transcribing them using third-party speech recognition services and returning back to browser clients in real-time or close to real-time as well as saving for long-time storage. Application uses WebSocket for communication which allows relatively simple integration with conferencing applications which usually employ the same technology for client-server application. Application architecture procures possibility to add more transcription back-ends without extensive intervention into existing classes. The audio capturing logic is not limited to any particular audio format. Also, custom transcript processing can be implemented on top of already existing classes. The solution is relatively simple to build and launch and does not require any additional software or servers to run except Java environment. It is not bound to Kurento media server used as a conferencing platform in this diploma thesis - to integrate with this transcription solution the conferencing application is only required to be capable of RTP streaming and handling SDP messages.

Speech recognition services can be adapted for real-time web conference transcription either if they provide a very fast response time (faster than it would take a person to say a few words) or if they send data in portions accompanied by timestamps for synchronization of multiple audio streams which are transcribed independently and simultaneously. Unfortunately, none of the existing speech recognition APIs can guarantee 100% accuracy of recognition but such technologies are developing rapidly as they are demanded in many areas of human society, so the efficiency of their usage should be expected to grow more and more soon.

References

1. SPEECHLAB. *SpeechLab - Laboratory of Computer Speech Processing* [online] [visited on 2019-04-25]. Available from: https://www.ite.tul.cz/speechlabe/.
2. WEBRTC.ORG. *WebRTC* [online] [visited on 2019-03-20]. Available from: https://webrtc.org/.
3. WIKIPEDIA. *Speech recognition* [online] [visited on 2019-03-20]. Available from: https://en.wikipedia.org/wiki/Speech_recognition.
4. WIKIPEDIA. *Transcript(law)* [online] [visited on 2019-03-20]. Available from: https://en.wikipedia.org/wiki/Transcript_(law).
5. TEAM, Skype. *Introducing live captions subtitles in Skype* [online] [visited on 2019-03-27]. Available from: https://blogs.skype.com/news/2018/12/03/introducing-live-captions-and-subtitles-in-skype/.
6. GRIGORIK, Ilya. *High-performance browser networking*. Sebastopol, CA: O'Reilly, 2013. ISBN 1449344763.
7. WEBRTC.ORG. *Frequent questions / WebRTC* [online] [visited on 2019-04-16]. Available from: https://webrtc.org/faq/.
8. M. HANDLEY V. Jacobson, C. Perkins. *SDP: Session Description Protocol*. RFC Editor, 2006. Available from DOI: 10.17487/RFC4566. RFC. RFC Editor.
9. WIKIPEDIA. *Session Description Protocol* [online] [visited on 2019-04-16]. Available from: https://en.wikipedia.org/wiki/Session_Description_Protocol.
10. ROY, Radhika Ranjan. *Handbook of SDP for multimedia session negotiations: SIP and WeRTC IP telephony*. Boca Raton, FL: CRC Press/Taylor Francis Group, 2018. ISBN 9781138484498.
11. H. SCHULZRINNE, S. Casner. *RTP Profile for Audio and Video Conferences with Minimal Control*. RFC Editor, 2003. Available from DOI: 10.17487/RFC3551. RFC. RFC Editor.
12. JOHNSTON, Alan B. *SIP: understanding the Session Initiation Protocol*. 3rd ed. Boston: Artech House, 2009. ISBN 1607839954.
13. I. BAZ CASTILLO J. Millan Villegas, V. Pascual. *The WebSocket Protocol as a Transport for the Session Initiation Protocol (SIP)*. RFC Editor, 2014. Available from DOI: 10.17487/RFC7118. RFC. RFC Editor.

14. SEGEČ, P.; PALÚCH, P.; PAPÁN, J.; KUBINA, M. The integration of WebRTC and SIP: Way of enhancing real-time, interactive multimedia communication. In: *2014 IEEE 12th IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 2014, pp. 437–442. Available from DOI: `10.1109/ICETA.2014.7107624`.

15. WIKIPEDIA. *Real-time Transport Protocol* [online] [visited on 2019-04-01]. Available from: `https://en.wikipedia.org/wiki/Real-time_Transport_Protocol`.

16. M. BAUGHER D. McGrew, M. Naslund. *The Secure Real-time Transport Protocol (SRTP)*. RFC Editor, 2004. Available from DOI: `10.17487/RFC3711`. RFC. RFC Editor.

17. *SFU(Selective Forwarding Unit* [online] [visited on 2019-04-15]. Available from: `https://webrtcglossary.com/sfu/`.

18. RODRIGUEZ, Pedro; CERVIÑO ARRIBA, Javier; TRAJKOVSKA, Irena; SALVACHUA, Joaquin. *Advanced Videoconferencing Services Based on WebRTC* [online] [visited on 2019-04-28]. Available from: `https://www.academia.edu/28680258/Advanced_Videoconferencing_Services_Based_on_WebRTC`.

19. MEETECHO. *Janus - General purpose WebRTC server* [online] [visited on 2019-04-23]. Available from: `https://janus.conf.meetecho.com/docs/`.

20. *Jitsi projects - free and open source video conferencing communications* [online] [visited on 2019-04-23]. Available from: `https://jitsi.org/projects/`.

21. *Kurento 6.10.0 documentation* [online] [visited on 2019-04-23]. Available from: `https://doc-kurento.readthedocs.io/en/6.10.0/`.

22. MUNOZ, Luis Villasenor. *The BaBL Project* [online] [visited on 2019-04-23]. Available from: `https://appliedtech.iit.edu/real-time-communications-lab-information-technology-and-management/projects/babl-project`.

23. JITSI.ORG. *Jigasi* [online] [visited on 2019-04-23]. Available from: `https://github.com/jitsi/jigasi`.

24. JITSI.ORG. *A speech-to-text prototype - Jitsi* [online] [visited on 2019-04-23]. Available from: `https://jitsi.org/news/a-speech-to-text-prototype/`.

25. *CMUSphinx Open Source Speech Recognition* [online] [visited on 2019-03-27]. Available from: `https://cmusphinx.github.io/`.

26. GOOGLE.COM. *Google Cloud Speech-to-Text documentation* [online] [visited on 2019-04-16]. Available from: `https://cloud.google.com/speech-to-text/docs/`.

27. IBM. *IBM Cloud Speech to text documentation* [online] [visited on 2019-04-16]. Available from: `https://console.bluemix.net/docs/services/speech-to-text/getting-started.html#gettingStarted`.

28. MICROSOFT.COM. *Speech to text API | Microsoft Azure* [online] [visited on 2019-04-21]. Available from: https://azure.microsoft.com/en-us/services/cognitive-services/speech-to-text/.
29. AMAZON. *Amazon Transcribe Documentation* [online] [visited on 2019-04-23]. Available from: https://docs.aws.amazon.com/transcribe/index.html.
30. YANDEX.RU. *Yandex SpeechKit documentation* [online] [visited on 2019-04-21]. Available from: https://cloud.yandex.com/docs/speechkit/.
31. ORACLE. *JMF 2.1.1 Software Documentation* [online] [visited on 2019-03-12]. Available from: https://www.oracle.com/technetwork/java/javase/documentation-138769.html.
32. WIIPEDIA. *Ogg* [online] [visited on 2019-04-15]. Available from: https://en.wikipedia.org/wiki/Ogg.
33. WIKIPEDIA. *G.711* [online] [visited on 2019-04-15]. Available from: https://en.wikipedia.org/wiki/Ogg.
34. GOOGLE. *gRPC* [online] [visited on 2019-04-12]. Available from: https://grpc.io/.
35. WIIPEDIA. *Protocol Buffers* [online] [visited on 2019-04-14]. Available from: https://en.wikipedia.org/wiki/Protocol_Buffers.
36. NANOTRIX. *Nanotrix Documentation* [online] [visited on 2019-04-22]. Available from: https://docs.nanotrix.cloud/.
37. WIKIPEDIA. *Utterance* [online] [visited on 2019-03-27]. Available from: https://en.wikipedia.org/wiki/Utterance.
38. ORACLE. *Deque (Java Platform SE 8)* [online] [visited on 2019-04-05]. Available from: https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Deque.html.
39. KURENTO.ORG. *Installation Guide - Kurento 6.10.0 documentation* [online] [visited on 2019-04-10]. Available from: https://doc-kurento.readthedocs.io/en/latest/user/installation.html.
40. WIKIPEDIA. *Docker(software)* [online] [visited on 2019-04-10]. Available from: https://en.wikipedia.org/wiki/Docker_(software).
41. SPOIALA, Cristian; CALINCIUC, Alin; TURCU, Cornel; FILOTE, Constantin. Performance comparison of a WebRTC server on Docker versus Virtual Machine. *13th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS, Suceava, Romania, May 19-21, 2016* [online]. 2016, pp. 295–298 [visited on 2019-04-10]. Available from: https://www.researchgate.net/publication/303659645_Performance_comparison_of_a_WebRTC_server_on_Docker_versus_Virtual_Machine.
42. JWT.IO. *JSON Web Token Introduction* [online] [visited on 2019-04-10]. Available from: https://jwt.io/introduction/.

A Enclosed files

- Source code of the demo conferencing application;
- Source code of the transcribing application.

B Example transcripts

B.1 English

Spoken text:

Anna: Hello, I have not seen you for a while, how are you doing?

Katja: I'm doing just fine, planning a vacation.

Anna: Great, where are you planning to go?

Katja: Italy. I always wanted to go there for the architecture and cuisine.

Anna: Sounds perfect. By the way, how is your cat? I remeber it was a little sick.

Katja: Nothing to worry about, it is absolutely healthy now, running and playing like crazy!

Anna: Glad to hear that. I must go now. Bye!

Katja: Bye!

B.2 Czech

Spoken text:

Katja: Čau, Moniko, jak se máš?

Anna: Čau, ujde to. A co ty?

Katja: Jo, mám se docela fajn, ale mám moc práce.

Anna: Tak máš taky peníze, ne?

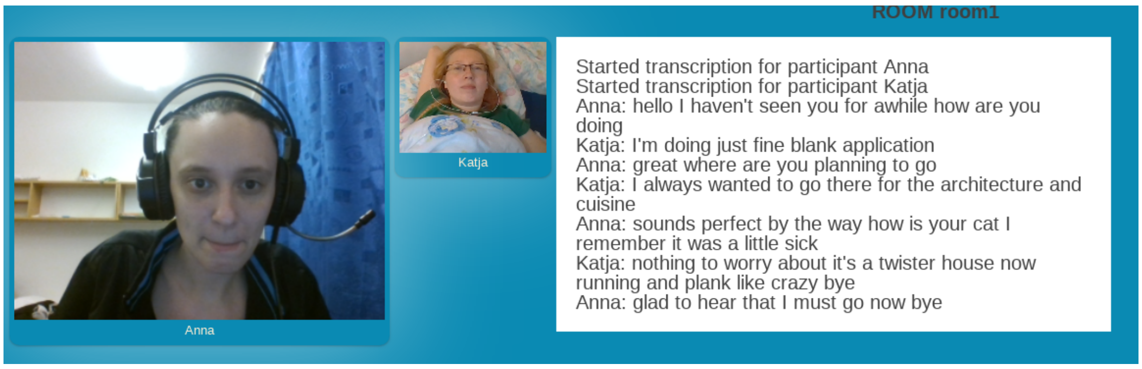
Katja: To jo, ale taky nemám vůbec čas. Ale ty ses nejaká smutná. Máš špatnou náladu?

Anna: Ne, jsem v pohode. Mám jen trochu rymu.

Katja: A nechces jít na kafe?

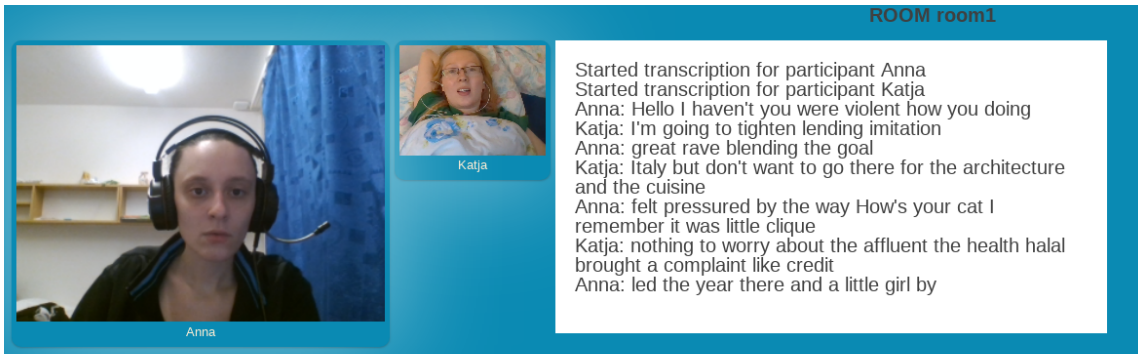
Anna: Ne, diky, nechci. Mám rande. Čau!

Katja: Ach jo, mám dneska fakt smůlu.



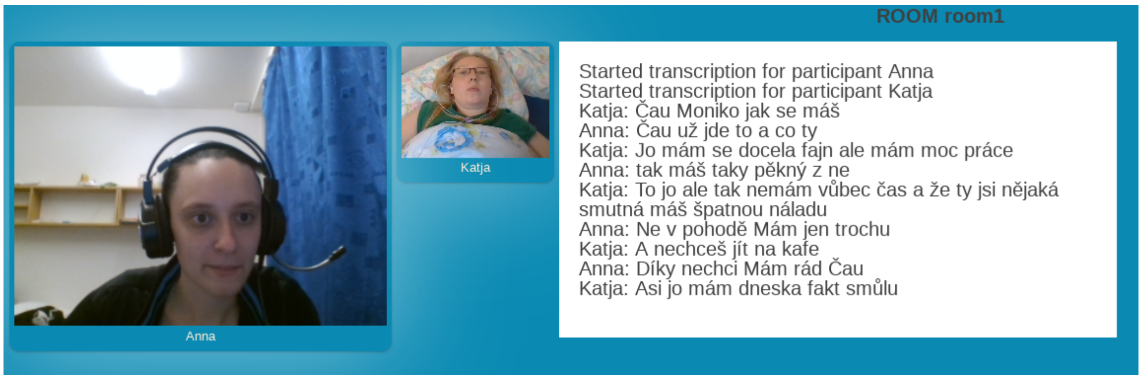
Started transcription for participant Anna
Started transcription for participant Katja
Anna: hello I haven't seen you for awhile how are you doing
Katja: I'm doing just fine blank application
Anna: great where are you planning to go
Katja: I always wanted to go there for the architecture and cuisine
Anna: sounds perfect by the way how is your cat I remember it was a little sick
Katja: nothing to worry about it's a twister house now running and plank like crazy bye
Anna: glad to hear that I must go now bye

Figure B.1: A conversation in English transcribed by Google Speech API.



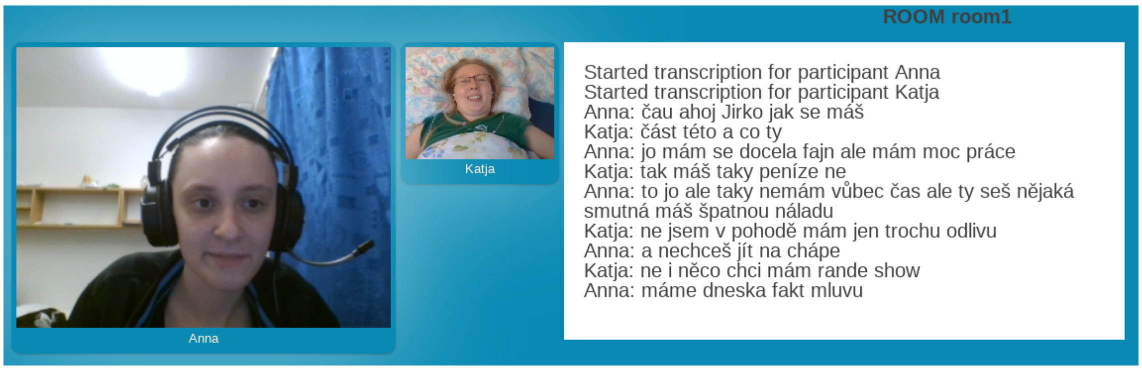
Started transcription for participant Anna
Started transcription for participant Katja
Anna: Hello I haven't you were violent how you doing
Katja: I'm going to tighten lending imitation
Anna: great rave blending the goal
Katja: Italy but don't want to go there for the architecture and the cuisine
Anna: felt pressured by the way How's your cat I remember it was little clique
Katja: nothing to worry about the affluent the health halal brought a complaint like credit
Anna: led the year there and a little girl by

Figure B.2: A conversation in English transcribed by TUL SRE.



Started transcription for participant Anna
Started transcription for participant Katja
Katja: Čau Moniko jak se máš
Anna: Čau už jde to a co ty
Katja: Jo mám se docela fajn ale mám moc práce
Anna: tak máš taky pěkný z ne
Katja: To jo ale tak nemám vůbec čas a že ty jsi nějaká smutná máš špatnou náladu
Anna: Ne v pohodě Mám jen trochu
Katja: A nechceš jít na kafe
Anna: Díky nechci Mám rád Čau
Katja: Asi jo mám dneska fakt smůlu

Figure B.3: A conversation in Czech transcribed by Google Speech API.



Started transcription for participant Anna
Started transcription for participant Katja
Anna: čau ahoj Jirko jak se máš
Katja: část této a co ty
Anna: jo mám se docela fajn ale mám moc práce
Katja: tak máš taky peníze ne
Anna: to jo ale taky nemám vůbec čas ale ty seš nějaká smutná máš špatnou náladu
Katja: ne jsem v pohodě mám jen trochu odlivu
Anna: a nechceš jít na chápe
Katja: ne i něco chci mám rande show
Anna: máme dneska fakt mluvu

Figure B.4: A conversation in Czech transcribed by TUL SRE.